KTH Computer Science
and Communication

On practical machine learning and data analysis

DANIEL GILLBLAD

Doctoral Thesis
Stockholm, Sweden 2008

TRITA-CSC-A 2008-11
ISSN-1653-5723 KTH School of Computer Science and Communication
ISRN-KTH/CSC/A--08/11--SE SE-100 44 Stockholm
ISBN-978-91-7178-993-3 SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framläges till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi onsdagen den 11 juni 2008 klockan 13.00 i Sal FD5, AlbaNova, Kungl Tekniska högskolan, Roslagstullsbacken 21, Stockholm.

Tryck: Universitetsservice US AB

Swedish Institute of Computer Science, SE–164 29 Kista, SWEDEN
SICS Dissertation Series 49
ISSN-1101-1335
ISRN SICS-D--49--SE

Abstract

This thesis discusses and addresses some of the difficulties associated with practical machine learning and data analysis. Introducing data driven methods in *e. g.* industrial and business applications can lead to large gains in productivity and efficiency, but the cost and complexity are often overwhelming. Creating machine learning applications in practise often involves a large amount of manual labour, which often needs to be performed by an experienced analyst without significant experience with the application area. We will here discuss some of the hurdles faced in a typical analysis project and suggest measures and methods to simplify the process.

One of the most important issues when applying machine learning methods to complex data, such as *e. g.* industrial applications, is that the processes generating the data are modelled in an appropriate way. Relevant aspects have to be formalised and represented in a way that allow us to perform our calculations in an efficient manner. We present a statistical modelling framework, *Hierarchical Graph Mixtures*, based on a combination of graphical models and mixture models. It allows us to create consistent, expressive statistical models that simplify the modelling of complex systems. Using a Bayesian approach, we allow for encoding of prior knowledge and make the models applicable in situations when relatively little data are available.

Detecting structures in data, such as clusters and dependency structure, is very important both for understanding an application area and for specifying the structure of *e. g.* a hierarchical graph mixture. We will discuss how this structure can be extracted for sequential data. By using the inherent dependency structure of sequential data we construct an information theoretical measure of correlation that does not suffer from the problems most common correlation measures have with this type of data.

In many diagnosis situations it is desirable to perform a classification in an iterative and interactive manner. The matter is often complicated by very limited amounts of knowledge and examples when a new system to be diagnosed is initially brought into use. We describe how to create an incremental classification system based on a statistical model that is trained from empirical data, and show how the limited available background information can still be used initially for a functioning diagnosis system.

To minimise the effort with which results are achieved within data analysis projects, we need to address not only the models used, but also the methodology and applications that can help simplify the process. We present a methodology for data preparation and a software library intended for rapid analysis, prototyping, and deployment.

Finally, we will study a few example applications, presenting tasks within classification, prediction and anomaly detection. The examples include demand prediction for supply chain management, approximating complex simulators for increased speed in parameter optimisation, and fraud detection and classification within a media-on-demand system.

Acknowledgements

This work has partly been performed in collaboration with others, most notably Anders Holst within the Hierarchical Graph Mixtures and applications thereof. The data preparation methodology and supporting software was developed together with Per Kreuger, and thorough testing of the incremental diagnosis model has been performed by Rebecca Steinert.

The larger part of the research leading up to this thesis was conducted at both the Adaptive Robust Computing (ARC) group and later the Industrial Applications and Methods (IAM) group at SICS, Swedish Institute of Computer Science, as well as in the Computational Biology and Neurocomputing (CBN) group of the School of Computer Science and Communication (CSC) at the Royal Institute of Technology (KTH). I would like to acknowledge all of the support and help from these institutions and the people working there.

I would like to express my gratitude to professor Anders Lansner, my advisor at KTH, for encouragement, support and for allowing me to join the SANS/CBN group at Nada.

I would also like to thank Anders Holst, without whom this thesis would not have been possible. His ingenuity and support lies behind most of the work presented here.

I am also grateful to Björn Levin, for friendship, support, and ideas during these years.

All the people that I have been working together with in various research projects deserve a special thank you, especially Diogo Ferreira, Per Kreuger, and Rebecca Steinert.

As a list of all the people who deserve my sincere thank you would extend the length of this thesis beyond control, I fear I will always leave someone out. However, I would like to thank all past and present members of the ARC and IAM groups at SICS, as well as the SANS/CBN group at CSC for a very pleasant atmosphere and inspiring discussions.

Finally, I thank Isabel. I could not, and most certainly would not, have done it without you.

Contents

Chapter 1

Introduction

1.1 Understanding and Modelling Complex Systems

The availability of fast and reliable digital computers has lead to significant new possibilities to understand complex systems, such as biochemical processes, sophisticated industrial production facilities and financial markets. The patterns arising in any such system are generally a consequence of structured hierarchical processes, such as the physical processes in a production plant. Finding this structure can lead to increased knowledge about the system and the possibility of creating, among other things, better control and decision support systems.

Here, we will concern ourselves with the study of such complex systems through examples. From historical data we can estimate and model the processes in the system at a level of abstraction that, although not able to provide a complete understanding of the inner workings, is detailed enough to provide useful information about dependencies and interconnections at a higher level. This, in turn, can allow us to *e. g.* classify new patterns or predict the future behaviour of the system.

The focus of this work is on artificial systems, or more specifically, man-made industrial and financial systems. However, the methods described are by no means limited to this areas and can be applied to a wide variety of both natural and artificial systems.

1.2 Data Analysis and Machine Learning

During the last decades, there has been an incredible growth in our capabilities of generating and storing data. In general, there is a competitive edge in being able to properly use the abundance of data that is being collected in industry and society today. Efficient analysis of collected data can provide significant increases in productivity through better business and production process understanding and highly useful applications for *e. g.* decision support, surveillance and diagnosis [Gillblad *et al.*, 2003].

The purpose of data analysis is to extract answers and useful patterns such as regularities and rules in data. These patterns can then be exploited in making predictions, diagnoses, classifications etc. Typical examples of working industrial and commercial applications are

- Virtual sensors, *i. e.* an indirect measurement of values computed from values that are easier to access.
- Predictive maintenance and weak point analysis through *e. g.* maintenance and warranty databases.
- Incremental step-wise diagnosis of equipment such as car engines or process plants.
- Intelligent alarm filtering and prioritisation of information to operators of complex systems.
- Fraud and fault detection in *e. g.* data communication systems and eBusiness.
- Sales and demand prediction, *e. g.* in power grids or retail.
- Speed-up through model approximation in control systems, *e. g.* replacing a slower simulator with a faster learning system approximation.
- Clustering and classification of customers, *e. g.* for targeted pricing and advertising, and identification of churners, *i. e.* customers likely to change provider.

With all data analysis and machine learning related applications running within industry, government, and homes, it is very hard to argue that the fields have not produced successful real world applications. However, there is still a definite gap between the development of advanced data analysis and machine learning techniques and their deployment in actual applications. There are several reasons for this.

Adapting and applying theoretical machine learning models to practical problems can be very difficult. Although it is often possible to achieve fair performance with a standard model formulation, we usually need a quite high degree of specialisation to achieve good performance and to satisfy constraints on *e. g.* computational complexity. Even if this is not necessary in certain situations, we usually still have to at least specify some model parameters or structure.

Understanding and preparing data for testing, validation and the actual application can be immensely time consuming. The data analyst trying to understand the data and the problem to be modelled is often not an expert in the application area, making acquisition of expert knowledge an important and time consuming task. Real-world data are also often notoriously dirty. It contains encoding errors (*e. g.* from errors during manual input) and ambiguities, severe levels of noise and outliers, and large numbers of irrelevant or redundant attributes. All of this

may cause severe problems in the modelling phase, and rectifying these problems is usually a very laborious task.

Deployment of data analysis or machine learning methods is difficult, and involves more than just developing a working model for *e. g.* prediction or classification. Creating interfaces for accessing data and user interaction is often much more labour intense than the actual model development, demanding a high level of commitment and belief that the system will perform as expected during its implementation.

1.3 Research Questions

In this thesis we will try to come to terms with some of these problems, and to at least in part bridge the gap between learning systems and their applications. We will introduce a flexible statistical modelling framework where detailed, robust models can easily be specified, reducing the complexity of the model specification phase.

To further reduce the need of manual modelling, we will discuss methods for learning the model structure automatically from data. The problem of data preparation and understanding will also be investigated, and a practical work flow and tools to support it are described. This will then be extended into modelling and validation, describing the implementation of a modelling library and interactive data analysis tool.

We will also discuss a number of practical applications of machine learning, such as demand prediction, anomaly detection and incremental diagnosis.

1.4 Overview of the Thesis

Chapter 2 gives an introduction to machine learning, data analysis and related issues. A number of common methods are described briefly, along with a description of their relative advantages and shortcomings in different situations. By no means a complete reference, it is intended to introduce the reader to common terminology and serve as an introductory overview of available methods.

In chapter 3, Hierarchical Graph Mixtures (HGMs) are introduced. They provide a flexible and efficient framework for statistical machine learning suitable for a large number of real-world applications. The framework generalises descriptions of distributions so we can, for example, define a mixture where each component is described by a separate graph. The factors of this graph can in turn be described by mixtures, and so on.

Chapter 4 discusses how to discover and describe structure in data, such as correlations and clusters. This is important not only to gain an understanding of an application area through data, but also for efficient statistical modelling through *e. g.* Hierarchical Graph Mixtures. Here, an entropy based measure of association

between time series is described, which can be used to find the edges of a graphicl model.

Using the HGM framework, chapter 5 describes an incremental diagnosis system, useful when information relevant to the diagnosis has to be acquired at a cost. The statistical model used and related calculations is presented along with results on various artificial and real-world data sets.

Even though flexible and effective models are vital for successful implementations of machine learning, they are by no means the only necessary component for creating applications of data driven methods. Chapter 6 discusses how to create efficient tools to enable rapid analysis of data in conjunction with the development of data driven, machine learning based applications. Methodology and implementation issues are discussed in more detail for the data preparation and modelling phases. An overview of an extensive example implementation covering a large number of data analysis aspects is also provided.

In chapter 7, a number of examples of machine learning and data analysis applications are presented along with brief problem descriptions and test results. The examples include demand prediction for supply chain management, future state prediction of complex industrial processes, and fraud detection within a telecommunication network. Although these examples do not provide a complete overview of data analysis applications, they serve as case studies of practical applications in which the methods presented in earlier chapters are applied and refined.

Finally, chapter 8 provides a discussion on the results in this thesis and provides directions for future research. These research directions are found both in the development of better data-driven algorithms, and in more practical matters such as how to facilitate rapid development and deployment of these methods.

1.5 Contributions

We introduce a statistical framework, *Hierarchical Graph Mixtures*, for efficient data-driven modelling of complex systems. We show that it is possible to construct arbitrary, hierarchical combinations of mixture models and graphical models. This is done by expressing a number of operations on graphical models and mixture models in such a way that it does not matter how the sub-distributions, *i. e.* the component densities in the case of mixture models and factors in the case of graphical models, are represented as long as they provide the same operations. This has to our knowledge never been shown before. As we discuss in chapter 3, this in turn allows us to create flexible and expressive statistical models that are often very computationally efficient compared to more common graphical model formulations using belief propagation.

We also introduce a framework for encoding previous knowledge, apart from what is allowed by specifying the structure of the Hierarchical Graph Mixture, based on Bayesian statistics. By noting that conjugate prior distributions used in the framework can be expressed on a parametric form similar to the posterior

distribution, we introduce a separate hierarchy for expressing previous knowledge or assumptions without introducing additional parameterisations or operations. The practical use of this is exemplified in chapter 5, where we create an incremental diagnosis system that both needs to incorporate previous knowledge and adapt to new data.

In chapter 4, we introduce a novel measure of correlation for sequential data that does not suffer from problems that other correlation measures show in this context. When creating correlograms from complex time series, actual correlations are drowned out by noise and slow moving trends in data, making it impossible to accurately determine delays between correlations in variables and to find the multiple correlation peaks that control loops and feedback in the system introduce. From the basic statement that we are dealing with sequential data we derive a new, much more sensitive and accurate measure of the dependencies in time series.

In chapter 5, we present a new approach to creating an incremental diagnosis system, addressing a number of critical practical concerns that have never before been consistently addressed together. The system is based on the Hierarchical Graph Mixtures of chapter 3, and makes use of both the possibility to create hierarchical combinations of graphs and mixtures, and the possibility to encode previous knowledge into priors hierarchically. By creating what essentially is a mixture of mixtures of graphical models, we introduce a model with low computational complexity suitable for interactive use while still performing very well on real-world data.

We show how previous knowledge encoded into prototypical data, a process that can make use of already available FMEA (Failure Mode and Effects Analysis) or fault diagrams, can be used in a statistical diagnosis model through careful Bayesian estimation using a sequence of priors. We also show that we can manage user errors by detecting inconsistencies in the input or answer sequence. How this can also be used to increase diagnosis performance is also demonstrated.

In chapter 6, we introduce a new methodological approach to data preparation that does not suffer from the limitations of earlier proposals when it comes to processing industrial data. We show how the methodology can be reduced to a small set of primitive operations, which allows us to introduce the concept of a "scripting language" that can manage the iterative nature of the process. We also show how both the data preparation and management operations as well as a complete modelling environment based on the Hierarchical Graph Mixtures can be implemented into a fast and portable library for the creation and deployment of applications. Coupled to this, we demonstrate a high-level interactive environment where these applications can be easily created.

In chapter 7, we present a number of practical applications of the methods discussed earlier. Among other examples, we describe an approach to demand prediction based on Bayesian statistics where we show that, by modelling the application appropriately, we can provide both good predictions and future demand and the uncertainty of the prediction. We also discuss how to perform future state prediction in complex system with learning systems, with the objective to replace a

complex simulator used by an optimiser to provide optimal production parameters. By using properties of the process itself and the cost function, we can reduce the problem to a one of manageable complexity. The applications provide examples both of graphical models, mixture models, Bayesian statistics, and combinations thereof, as well as pre-processing of data.

Chapter 2

Data Analysis and Machine Learning

2.1 Practical Data Analysis

Data analysis, in the sense that we will use the term here, is the process of finding useful answers from and patterns in data. These patterns may be used for *e. g.* classification, prediction, and detecting anomalies, or simply to better understand the processes from which the data were extracted. In practise we often do not have any real control over what data are collected. There is often little room for experiment design and selection of measurements that could be useful for the intended application. We have to work with whatever data are already available.

Fortunately, what data are already available is nowadays often quite a lot. Companies often store details on all business transactions indefinitely, and an industrial process may contain several thousands of sensors whose values are stored at least every minute. This gives us the opportunity to use these data to understand the processes and to create new data-driven applications that might not have been possible just a decade ago. However, the data sets are often huge and not structured in a way suitable for finding the patterns that are relevant for a certain application. In a sense, there is a gap between the generation of these massive amounts of data and the understanding of them.

By focusing on extracting knowledge and creating applications by analysing data already present in databases, we are essentially performing what is often referred to as *knowledge discovery* and *data mining*.

2.2 Machine Learning

To put it simply, one can say that machine learning is concerned with how to construct algorithms and computer programs that automatically improve with experience. We will however not be concerned with the deeper philosophical questions

here, such as what learning and knowledge actually are and whether they can be interpreted as computation or not. Instead, we will tie machine learning to performance rather than knowledge and the improvement of this performance rather than learning. These are a more objective kind of definitions, and we can test learning by observing a behaviour and comparing it to past behaviours. The field of machine learning draws on concepts from a wide variety of fields, such as philosophy, biology, traditional AI, cognitive science, statistics, information theory, control theory and signal processing. This varied background has resulted in a vast array of methods, although their differences quite often are skin-deep and a result of differences in notation and domain.

Here we will briefly present a few of the most important approaches and discuss their advantages, drawbacks and differences. For a more complete description, see *e. g.* [Russel and Norvig, 1995; Langley, 1996; Mitchell, 1997].

Issues and Terminology in Machine Learning

More formally, machine learning operates in two different types of spaces: A space X, consisting of data points, and space Θ, consisting of possible machine learning models. Based on a training set $\{x^{(\gamma)}\}_{\gamma=1}^{N} \subset X$, machine learning algorithms select a model $\theta \in \Theta$, where Θ is the space of all possible models in a selected model family. Learning here corresponds to selecting suitable values for the parameters θ in a machine learning model from the training set. How this selection is performed and what criteria is used to evaluate different models varies from case to case.

We have here made a distinction between *supervised* and *unsupervised* learning. In the former case, data are divided into *inputs* X and *targets* (or *outputs*) Y. The targets represent a function of the inputs. Supervised learning basically amounts to fitting a pre-defined function family to a given training set $\{(x^{(\gamma)}, y^{(\gamma)})\}_{\gamma=1}^{N} \subset X \times Y$, *i. e.* we want to find a function $y = f(x; \theta)$ where $\theta \in \Theta$. *Prediction* and *classification* are common applications for supervised learning algorithms.

In unsupervised learning, data are presented in an undivided form as just a set of examples $\{x^{(\gamma)}\}_{\gamma=1}^{N} \subset X$. The learning algorithm is then expected to uncover some structure in these data, perhaps just to memorise and be able to recall examples in the future, or to extract underlying features and patterns in the data set. Clustering data into a set of regions where examples could be considered to be "similar" by some measure is a typical application of unsupervised learning.

The type of parameterisation Θ and estimation procedure specifies how the model will *generalise*, *i. e.* how it will respond to examples not seen in the training data set. The generalisation performance is affected by the implicit assumptions the model makes about the parameter space Θ and data space X or $X \times Y$. The performance of the model, that is how close the models *output* or *target* variables are to the true values, is tested on a *validation* or *test data set* that should be different from the training data. This is done in order to evaluate the generalisation performance of the model, giving us an indication of how it would perform in practise.

Common Tasks in Machine Learning

Although we have already touched upon some of them, let us have a closer look at some of the most common tasks within machine learning. These tasks usually involve either predicting unknown or future attribute values based on other, known attributes in a pattern, or describing data in a human-interpretable or otherwise useful form.

Classification In this context, *classification* is the process of finding what class an example belongs to given the known values in the example [Hand, 1981; McLachlan, 1992]. In other words, it deals with learning a function that maps an example into a discrete set of pre-defined categories or classes. A wide range of tasks from many areas can be posed as classification problems, such as determining whether or not a client is credit worthy based on their financial history, or diagnosing a patient based on the symptoms. Other examples include classifying the cause of equipment malfunction, character recognition, and identification of items of interest in large databases. Automated classification is one of the most common applications of machine learning.

All deterministic classifiers divide the input space by a number of *decision surfaces*. They represent the decision boundaries between the different classes, and each resulting compartment in input space is associated with one class. The possible shapes of these decision surfaces vary with the classifier method. The perhaps most commonly applied shape is that of a hyperplane, which is the resulting decision surface for all linear methods.

Similar to classification, *categorisation* tries to assign class labels to examples where the exact type and number of categories are not known, which is directly related to clustering described below.

Regression and prediction Where classification uses a function that maps to a finite, discrete set, *regression* uses a function that maps an example to a real-valued prediction variable. As with classification, machine learning applications of regression and prediction are plentiful and well studied. They include time series prediction, where *e. g.* the future value of a stock is predicted based on its previous values; customer demand prediction based on historical sales and advertising expenditure [Gillblad and Holst, 2004b]; and predicting the future state of a production process [Gillblad *et al.*, 2005]. Other examples could be estimating the amount of a drug necessary to cure a patient based on measurable symptoms or the number of accidents on a road given its properties.

Anomaly Detection Anomaly detection can be defined as the separation of an often inhomogeneous minority of data, with characteristics that are difficult to describe, from a more regular majority of data by studying and characterising this majority. This can be done in a model based manner, where *e. g.* a physical model or simulator is used as a comparison to detect anomalous situations

(*e. g.* [Crowe, 1996; Venkatasubramanian *et al.*, 2003]), but also with a data-driven, machine learning approach, where a model representing normal situations is constructed from (normal) data and large deviations from this model is considered anomalous [Eskin, 2000; Lee and Xiang, 2001]. Closely related to this approach, *deviation detection* [Basseville and Nikiforov, 1993] focuses on detecting the most significant changes in data compared to previously measured values, regardless of whether this is to be considered normal or not.

Structure Description Finding and describing the properties and structure of a data set can give important insight into the processes generating the data and explain phenomena that are not yet understood. One of the most common structure description tasks is *clustering* [Duda and Hart, 1973; Jain and Dubes, 1988; McLachlan and Basford, 1988]. It tries to identify a finite set of categories or *clusters* that divide and describe the data set in a meaningful way. The clusters may be mutually exclusive, have a graded representation where a sample may belong in part to several clusters, or even have a more complex hierarchical structure.

Dependency derivation consists of finding a model that explains statistically significant dependencies between attributes in the data [Ziarko, 1991]. The resulting structure is highly useful for both for understanding the data and application area as well as for efficient modelling of data. For example, graphical models (see section 2.2) can make direct use of this dependency structure to efficiently represent the joint distribution of the data. In interactive and exploratory data analysis the quantitative level of the dependency structure, *i. e.* to what degree the attributes are dependent on each other or the strength of the correlations, is also highly useful.

Often used in exploratory data analysis and report generation, *summarisation* [Piatetsky-Shapiro and Matheus, 1992] of data involves creating compact descriptors for a data set for human interpretation. These can range from simple descriptive statistics such as measuring the mean and variance of attributes to more complex visualisation techniques.

Instance-Based Learning

One of the conceptually most straightforward approaches to machine learning is to simply store all encountered training data. When a new sample is presented, classification or prediction is performed by looking up the samples most similar to the presented one in the stored examples in order to assign values to the target variables. This is the foundation of *instance-based learning* [Aha *et al.*, 1991], which includes techniques such as nearest neighbour methods, locally weighted regression and case based methods. The approach often requires very little work during training, since the processing is delayed until a new sample arrives and the most similar of the stored samples have to be found. Because of this delayed processing, instance-based methods are sometimes also referred to as "lazy" learning methods.

The most basic instance-based learner is the *Nearest Neighbour* algorithm [Cover and Hart, 1967]. This uses the most similar sample in the stored data set to predict

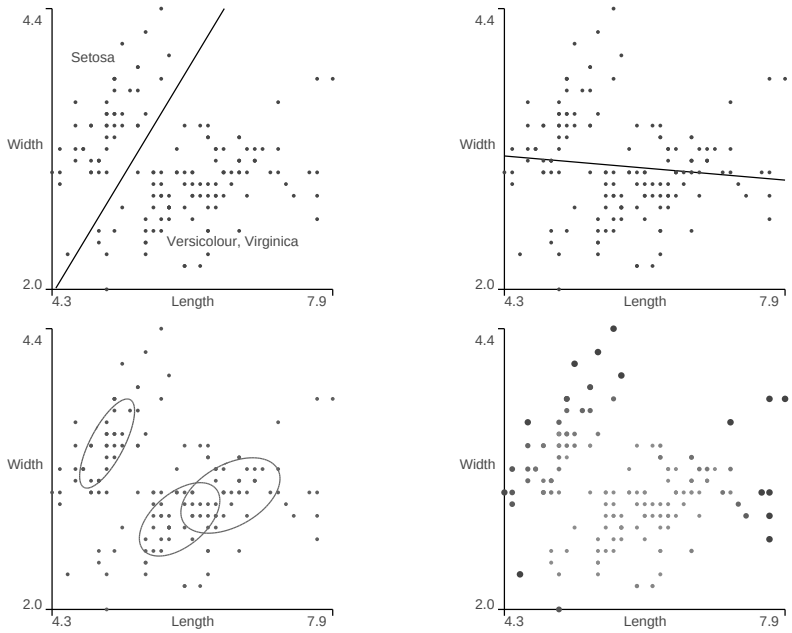

Figure 2.1: Examples of machine learning applications. All plots show the sepal length and width of plants from the one classical test data bases for machine learning, the Iris Plants Database [Fisher, 1936; D.J. Newman and Merz, 1998]. The upper left graph shows an example of a decision line between the *Setosa* class and the two other classes in the database, *Versicolour* and *Virginica*. The top right graph displays a simple linear regression line between length and width, which is rather flat due to the lack of correlation between length and width. The lower left graph shows a simple clustering of the data set into three clusters. The lower right graph shows an example of a simple anomaly detector, where the darkness and size of each data point represents how far from normal it is considered to be by a mixture of Gaussians (see chapter 3 for a closer description).

the target values. Since this procedure often is sensitive to noise and often does not generalise very well, it is usually extended to use the k nearest samples instead. This is referred to as a *k-Nearest Neighbour* method. The target values are usually combined using the most common value among the nearest samples in the case of discrete attributes, and the mean value in the case of continuous attributes. A natural refinement to the algorithm is to weigh the contribution of each sample by the distance, letting closer neighbours contribute more to the result [Dudani, 1976]. Using this approach, k can even be set to be the number of patterns in the stored data, *i. e.* all the stored patterns contribute to the result.

Common for all instance-based algorithms is that they require a metric on the sample space, typically chosen to be the Euclidean distance for continuous attributes and a Hamming distance, *i. e.* the number of differing attributes, for discrete attributes. In practise, the input attributes are usually not of the same or perhaps even comparable scale. The attributes are therefore often scaled or normalised to make them comparable.

A more complex, symbolic representation can also be used for the samples, which means that the methods used to find similar samples are also more elaborate. This is done in *Case-based reasoning* [Aamodt and Plaza, 1992]. Case-based reasoning does not assume a Euclidean space over the samples, but instead logical descriptions of the samples are typically used.

The main advantage of instance-based methods is that they can use local representations for complicated target functions, constructing a different approximation for each new classification or prediction. On the other hand, the most noticeable disadvantage of the approach is the high cost of classifying new instances. The methods may also show a high sensitivity to excess inputs, *i. e.* only a subset of the inputs is actually relevant to the target values, compared to other machine learning methods. The distance between neighbours is easily dominated by irrelevant attributes not contributing to the classification.

Logical Inference

One of the earliest approaches to machine learning, and for a long time the dominant theme within *Artificial Intelligence*, is to treat knowledge as relations between logical variables. Representing a problem within logical variables is straightforward if the measured attributes are binary or nominal, but requires some choice of representation if the attributes are numerical. The variables must be encoded with suitable predicates, such as treating a variable as "true" if it falls within a certain interval and "false" otherwise.

In some cases, logical inference systems can also be extended to deal with uncertainties in data, *e. g.* by the use of *fuzzy logic* [Zadeh, 1965]. While in normal, Boolean logic, everything is expressed in binary terms of true or false, fuzzy logic allows for representations of varying degrees of truth.

It is possible to learn logical representations directly from examples using *e. g.* a *Decision Tree* [Quinlan, 1983]. Decision trees are one of the most widely used representations for logical inference, and is capable of learning disjunctive expressions while being robust to noisy data. Decision trees classify instances by propagating them from the root down to some leaf node which provides the classification. Each node in the tree tests one predicate on one attribute, and the subtree is selected accordingly.

Finding a decision tree representation from data is typically done by a greedy search through the space of all possible decision trees. Each attribute is evaluated to determine how well it classifies the examples. The best attribute is chosen as a test at the root node, and a descendant of this node is created for each possible

outcome of the attribute. The process is then repeated for each descendant node using the training data associated with it, until all or most training examples belong to one class, at which point a leaf node is created.

Two commonly used variants of this basic approach are the ID3 algorithm [Quinlan, 1986] and the C4.5 algorithm [Quinlan, 1993]. While rather straightforward extensions to these algorithms make it possible to incorporate *e. g.* continuous-valued input attributes and training examples with missing attribute values, more substantial extensions are necessary to learning target functions with continuous values, and the application of decision trees in this setting is less common.

Note that the methods mentioned above usually use predicates that depend on only one attribute. More complex predicates that depend on more than one attribute can be used, but representation and learning becomes more difficult [Breiman *et al.*, 1984].

Artificial Neural Networks

The field of *Artificial Neural Networks* (see *e. g.* [Minsky and Papert, 1969; Kosko, 1993; Haykin, 1994]) includes a number of algorithms with quite different abilities, but they all share one basic property: The calculations are performed by a number of smaller computational units, connected by weighted links through which activation values are transmitted. The computation done by these units is usually rather simple, and may typically amount to summing the activation received on the input connections and then passing it through a transfer function. The transfer function is usually monotonous, non-linear and with a well defined output range, limiting the output of the unit.

When a neural network is used *e. g.* for prediction or classification, an input pattern is typically presented to a set of input units. These input units then propagate their resulting activation through the network as specified by the connections, until it arrives at a set of output units, whose outputs are interpreted as the networks prediction or classification. Training the network amounts to estimating the weights of the connections so that they minimise the error of the outputs.

Artificial neural networks are partly inspired by observations from biological systems, where neurons build intricate webs of connections. The simple computational unit in an Artificial neural network would then correspond to the neuron, and the weighted links their interconnections. However, although the algorithms discussed here follow this principle, they are only loosely based on biology and are in fact known to be inconsistent with actual neural systems.

The perhaps most popular and widely used neural network architecture is the *Multi-Layer Perceptron*. It is organised in layers of units, the activation propagating from the units in the input layer, through any number of hidden layers until it reaches the output layer. A network operating in this way, *i. e.* the activation propagates in one direction only, is usually referred to as a *feed-forward* network. The multi-layer perceptron can be trained in a number of different ways, but the most common method is probably the first training algorithm that was described

for the architecture, the *back-propagation* algorithm [Rumelhart *et al.*, 1986]. It is a gradient descent algorithm, attempting to minimise the squared error between the network output values and the true values for these outputs.

This kind of neural network is well suited to handle data that contain noise and errors, but may require a substantial amount of time to train. The evaluation of the learnt network however is usually very fast, and neural networks can perform very well in many practical problems. However, the opportunity to understand *why* the network performs well is unfortunately limited, as the learnt weights are usually difficult to interpret for humans.

By introducing a feedback loop, *i. e.* connections that feed the output of units in one layer back into the units of the same or a previous layer, we can create a network with quite different abilities compared to the feed-forward network discussed above. This type of network is usually referred to as a *recurrent* neural network. The recurrent network is typically not used by sending a pattern from the input units to the output units and make direct use of their output values, but rather by letting the signals cycle round the network until the activity stabilises.

An example of a recurrent neural network is the *Hopfield network* [Hopfield, 1982], a fully connected feedback network where the weights usually are constrained to be symmetric, *i. e.* the weight from neuron i to neuron j is the same as from j to i. This type of network has mainly two applications; as an associative memory and to solve optimisation problems. A Hopfield network is characterised by its *energy function*, which is a scalar function from the activity in the network. The energy function defines an *energy landscape*, in which the activity pattern strives to find a local minimum during the recall phase. These local minima constitute stable patterns of activity.

A related group of neural networks are the *competitive* neural networks, such as *Learning Vector Quantization* [Kohonen, 1990] and *Self-Organizing Maps* [Kohonen, 1982, 1989]. The units react in relation to how close they can be considered to be to an input pattern, and compete for activation. Usually the networks use a winner-takes-all strategy, the most active unit suppressing all other units. The units in the network can be considered to represent prototypical input patterns, making the approach somewhat similar to the instance based methods discussed earlier. Training the networks amounts to iteratively adapting the prototypical units towards the patterns that they respond to.

Evolutionary Computation

The term evolutionary computation is usually used to describe methods that use concepts working on populations to perform guided search within a well defined domain. In practise, the field is mainly concerned with combinatorial optimisation problems, and to a lesser degree self-organisation.

Optimisation related evolutionary computing can roughly be divided into *evolutionary algorithms* and *swarm intelligence* [Beni and Wang, 1989]. Swarm intelligence concerns itself with decentralised self-organising systems consisting of

populations of simple agents, where local interactions lead to the emergence of an organised global behaviour. This can be observed in nature in *e. g.* bird flocks and ant colonies. The algorithms are typically used to find approximate solutions to combinatorial optimisation problems.

This is true also for evolutionary algorithms, a large and varied field drawing inspiration mainly from concepts within evolution theory, such as mutation, recombination, reproduction and natural selection. As an example, *genetic algorithms* provide an approach to learning based on simulated evolution. Solution hypothesis are encoded as strings of numbers, usually binary, and their interpretation depends on a chosen encoding of the problem domain. A population of such strings is then evolved through mutating and combining a subset of the strings before selecting a subset of them according to some measure of fitness.

Similarly, *genetic programming* [Cramer, 1985; Koza, 1992] is a method to automatically find computer programs that performs a user-defined task well, and can be viewed as a variant of genetic algorithms where the individuals in the population are computer programs rather than bit strings (or where these strings indeed represent computer programs).

As the generation of new programs through genetic programming is very computer intensive, applications have quite often involved solving relative simple problems. However, with the increase in computing power, applications have become more sophisticated and their output can now rival programs by humans, *e. g.* in certain applications of sorting. Still, choosing what functional primitives to include in the search may be very difficult.

Statistical Methods for Machine Learning

Probabilistic methods have consistently gained ground within the learning systems community. They are widely considered to be one of the most promising foundations for practical machine learning, and both methods and applications are rapidly emerging. Here, we will instead mention some of the more common statistical methods and briefly discuss the basic assumptions behind them.

In essence, statistical methods represent data as the outcomes of a set of random variables, and tries to model the probability distribution over these variables. Historical data is used to estimate the probability distribution, in order to *e. g.* draw conclusions about the processes that generated the data or classify incomplete examples.

When it comes to how the probability distributions are represented, a distinction is often made between *parametric* and *non-parametric* models. In a parametric model, the general form and structure is already known, and only a relatively small number of parameters controlling the specific shape of the distribution are estimated from data. This could be *e. g.* the mean and variance of a Gaussian distribution or the shape and scale parameters of the gamma distribution.

In contrast, non-parametric methods try to impose very few restrictions on the shape of the distribution. The term non-parametric does not mean that the

methods completely lack parameters, but rather that the number and nature of the parameters, which may in fact be very high, are flexible and depend on the data. The *Parzen* or *kernel density* estimator is an example of a non-parametric method [Parzen, 1962]. Here, the distribution of the data is approximated by placing a kernel density function, typically a Gaussian with fixed covariance matrix, over each sample and adding them together. This makes it possible to extrapolate the density to the entire domain. Classification of new sample points can be performed by calculating the response from each kernel function and adding the response levels by class. This is then normalised and interpreted as the probability distribution over the class for the new sample. Similarly, prediction can be performed by calculating a weighted mean of the training samples based on their responses. The method works in the same way as a nearest neighbour model using a function of the distance for weighting all neighbours, which means that it also suffers from the same problems with excess input attributes and classification complexity.

A related model that perhaps is best described as *semi-parametric* is the *Mixture Model* [McLachlan and Basford, 1988]. The Mixture Model addresses the problem of representing a complex distribution by representing it through a finite, weighted sum of simpler distributions. There are different ways to fit the parameters of the sub-distributions and their weights in the sum to a set of data. One common method is the Expectation Maximisation algorithm [Dempster *et al.*, 1977]. This is an iterative method which alternately estimates the probability of each training sample coming from each sub-distribution, and the parameters for each sub-distribution from the samples given these probabilities.

A different approach is used by the *Naive Bayesian Classifier* [Good, 1950]. The underlying assumption of this model is that all input attributes are independent, or to be precise, conditionally independent given the class. This leads to a simple representation of the complete distribution, which basically can be written as a product over the marginal distributions of the attributes, including the class. Since a complete representation of the distribution over the domain would have vastly more parameters than all these marginal distributions together, stable estimation from data becomes much more tractable and the of overfitting is reduced. Although the independence assumption used may seem to simplistic at first, Naive Bayesian classifiers often perform surprisingly well in complex real-world situations [Hand and Yu, 2001].

Hidden Markov Models [Baum, 1997; Rabiner, 1989], or HMM for short, is an popular tool in sequence analysis. A HMM represents a stochastic process generated by an underlying Markov chain that is observed through a distribution of the possible output states. In the discrete case, a HMM is characterised by a set of states and an output symbol alphabet. Each state is described by an output symbol distribution and a state transition probability distribution. The stochastic process generates a sequence of output symbols by emitting an output according to the current state output distribution, and then continuing to another state using the transition probability distribution. The activity of the source is observed indirectly through the sequence of output symbols, and therefore the states are said to be

hidden. Given a sequence of output symbols, it is possible to make inferences about the HMM structure and probability distributions.

In essence, both Naive Bayes and Hidden Markov Models can be viewed as special cases of *Graphical Models* (see *e. g.* [Cowell *et al.*, 1999] for an introduction). These models exploit the fact that the joint distribution of a number of attributes often can be decomposed into a number of locally interacting factors. These factors can be viewed as a directed or undirected graph, hence the naming. Nodes represent attributes and arcs dependencies, or more precisely, the lack of arcs represent conditional independencies between variables. Decomposing the joint distribution in this way roughly serves the same purpose as in mixture models, *i. e.* to simplify the representation and estimation of the distribution. Examples of graphical models include *Factor Graphs* [Kschischang *et al.*, 2001], *Markov Random Fields* [Kindermann and Snell, 1980], and *Bayesian Belief Networks* [Pearl, 1988].

The Bayesian belief network uses a directed graph to represent the conditional independencies between the attributes. The nodes in the graph can be both directly observable in data or *hidden*, allowing representation of *e. g.* attributes that have significant impact on the model but that cannot be measured. The distributions associated with these variables are usually estimated through variants of the expectation maximisation algorithm. To calculate the marginal distributions of attributes in the network given known values of some of the attributes, the *belief propagation* algorithm is typically used. This is a message passing algorithm that leads to exact results in acyclic graphs, but it can perhaps surprisingly also be used to arrive at good approximate results for graphs that contain cycles. This is usually referred to as *loopy belief propagation*.

Although the graphical structure in many cases can be at least partially estimated from data, it is perhaps most often constructed manually, trying to encode *e. g.* known physical properties of a process. Bayesian Belief Networks therefore rest in between learning systems and knowledge based methods, and are highly suitable to problems were there is a large base of available knowledge and a relative lack of useful training examples compared to the complexity of the data.

Other Methods

There is a very large number of methods available within machine learning, and we will by no means try to cover the complete field here. However, there is a couple of methods not discussed in the earlier sections that deserve a mention.

Reinforcement learning is an approach to performing learning in an environment that can be explored, and that accommodates delayed or indirect feedback to an autonomous agent [Barto *et al.*, 1983; Sutton, 1984]. The agent senses and acts in its environment in an effort to learn efficient strategies to achieve a set of defined goals. The approach is related to supervised learning in that it has a trainer, that may provide positive or negative feedback to indicate how desirable the state resulting of an agent's action is.

However, the feedback signal only indicates how good or bad the state was, and

the agent does not receive any information on the correct action as in supervised learning. The goal of the agent is to learn to select those actions that maximise some function of the reward, *e. g.* the average or sum, over time. This is useful in *e. g.* robotics, software agents and when learning to play a game where it is only possible to know whether a whole sequence of moves were good or bad.

The *Support Vector Machine* [Schölkopf, 1997; Burgess, 1998] is a learning algorithm for classification and regression with its roots in statistical learning theory [Vapnik, 1995]. The basic idea of a Support Vector Machine classifier is to map training data non-linearly to a high dimensional *feature space*, and try to create a separating hyperplane there. The plane is positioned to maximise the minimum distance to any training data point, which is solved like an optimisation problem.

To perform support vector regression, a desired accuracy has to be specified beforehand. The support vector machine then tries to fit a "tube" formed by the space between two hyperplanes, of a width corresponding to this accuracy to the training data. The support vector machine does provide a separating hyperplane that is in a sense optimal. However, it is not necessarily obvious how the transformation into high dimensional space should be selected.

By combining several simpler models, it may be possible to arrive at a better classifier or predictor. This is done in *e. g. bagging*, or bootstrap aggregating [Breiman, 1994]. The bagging algorithm creates replicate training sets by sampling with replacement from the total training set, and each classifier is estimated on one of these data sets separately. Classification or prediction is then performed by voting or averaging amongst the models respectively, reducing the expected value of the mean square error.

Boosting is another general method for improving the accuracy of any given learning algorithm, and the most common variant is known as AdaBoost [Freund and Schapire, 1997]. This algorithm maintains a weight for each instance in the training data set, and the higher the weight the more the instance influences the classifier learnt. At each trial, the vector of weights is adjusted to reflect the performance of the corresponding classifier, and the weight of misclassified instances is increased. Boosting often produces classifiers that are significantly more correct than one single classifier estimated from the same data.

Other Issues Within Machine Learning

A common problem for all machine learning methods is how to *validate* that the model will perform well on yet unseen data, and how to measure this performance. In general, machine learning methods have a tendency of *over fitting* to the examples used for training, leading to a decreased ability to generalise. A very detailed model fitted very closely to the training examples may perform very badly on new examples presented to the model. The tendency to over fit to training data usually increases with the number of free parameters in the model, leading to the fact that rather simple models often are preferable for very complex data or where there is a relative lack of training data. This is related to what is often called the *curse of*

dimensionality [Bellman, 1961], referring to the fact that when the number of input dimensions increase, the number of possible input vectors increase exponentially.

Dividing the data into a separate training set and a test set, where model performance is evaluated on the test set, may lead to a good estimation of generalisation performance if the data are homogeneous and plentiful. However, to make better use of data and get a better estimate of generalisation performance, *cross-validation* [Stone, 1974] can be used. The data set is partitioned into a number of smaller pieces, and the model is estimated and evaluated several times. Each time, one partition is removed from the data set. The model is then estimated on the remaining parts and evaluated on the extracted part. The average performance over all parts represents a good approximation of the generalisation performance of the model. When using k data partitions, the procedure is usually referred to as *k-fold cross-validation*. In the limit case of using as many partitions as there are examples in the data set, the procedure is usually called *leave-one-out cross-validation*. This is also often the preferable method of evaluating generalisation performance if the time training the model as many times as there are examples in the data set is not prohibitive.

What performance measures to use for evaluating a models generalisation capabilities is of course highly dependent on the intended application. In classification, the most widely used measure is the *error rate*, or the fraction of misclassifications made by the classifier. However, this does not tell us much about how informative the classifier is. A classifier that always outputs the same class would have a low error rate if this class is indeed the most common one. It is however usually of limited use in practise. A more suitable measure than the error rate could be the *mutual information* (see chapter 4) between the true class and the classifiers output, or the use of ROC (Reciever Operating Chacteristic) curves (plotting the number of true positives against the number of false positives in a sample). There may also be different costs associated with misclassification for the different classes, in which case the measure needs to take that into account. For numeric prediction, the mean-squared error, correlation coefficient or relative squared error are common measures of performance, but usually none of them alone give a good picture of the performance of the predictor.

All machine learning methods make some kind of assumptions about the attribute space and the regularities in it in order to be able to generalise at all. Quite common is the assumption that nearby patterns in the sample space belong to the same class or are associated to similar real-valued outputs. This means, however, that how we choose to represent the patterns to the model is crucial for how well it will perform [Simon, 1986]. How to choose this representation in practise is still very much an exploratory task for the analyst. Although there are approaches to help automate the process, in general the search space of tractable data transformations is vast, meaning that the time complexity of finding suitable transformations is too high for any practical purposes.

The theoretical characterisation of the difficulty of different types of machine learning problems and the capabilities and limitations of machine learning methods

is dealt with within *computational learning theory*. It tries to answer questions such as under what conditions a certain algorithm will be able to learn successfully and under what conditions are learning at all possible. For an introduction to the field, see *e. g.* [Anthony and Biggs, 1992].

2.3 Related Fields

As it is described earlier, we here use the term data analysis in a rather wide sense. This is to some degree also true of our use of the term machine learning, and although there are differences, there are a number of related research fields that could be described in much the same way as we have done above. The difference between these fields often lie more in the type of application area or techniques used than in the ultimate goal of the processes that they describe.

Using similar methods to statistical data analysis, *exploratory data analysis* is an approach to analysing data that relies heavily on user interaction and visualisation [Tukey, 1977]. In practise, visualisation plays a very important role in most data analysis projects, regardless of approach or methods used.

As discussed earlier, *Data Mining* and *Knowledge Discovery in Databases* (KDD) [Fayyad *et al.*, 1996b; Frawley *et al.*, 1992] are highly related to the concepts of data analysis outlined above, and some of the introductory texts to the field do indeed read much like descriptions of applied machine learning [Witten and Frank, 1999]. However, the goal of data mining can be expressed shortly as extracting knowledge from data in the context of large databases. As a consequence, the field also concerns itself with issues in database theory, knowledge acquisition, visualisation and descriptions of the whole analysis process. These questions are of a more practical nature and are largely overlooked in the field of machine learning, however critical they may be for the effective deployment of the methods.

Directly related to data analysis, *data fusion* [Waltz and Llinas, 1990; Hall and Llinas, 1997] tries to combine data from multiple sensors and associated databases in an effort to maximise the useful information content. The data and knowledge is often multimodal, representing sensory data streams, images, textual situation descriptions etc. This is combined into one coherent view of a situation *e. g.* for decision making or classification. With applications such as pattern recognition and tracking, it is closely related to the concepts of data analysis and machine learning as described earlier. Typical application areas of data fusion include military, robotics, and medicine.

Chapter 3

Hierarchical Graph Mixtures

3.1 Introduction

An important issue when applying learning systems to complex data, *e. g.* in advanced industrial applications, is that the system generating the data is modelled in an appropriate way. This means that the relevant aspects of the system have to be formalised and efficiently represented in the computer in a way that makes it possible to perform calculations on. We could do this by constructing physical models or simulators of some detail, but this might require quite an effort. If we instead choose to work on a higher level of abstraction where we do not manually model all relations in the system, and estimate some or all of the parameters of the model from historical data, we can reduce this effort significantly. Therefore, machine learning approaches becomes attractive. In this context, statistical learning methods have consistently gained ground within the machine learning community as very flexible tools for practical application development.

Two very commonly used examples of statistical machine learning models are *graphical models* and *finite mixture models*. In this chapter, we will introduce a framework, the *Hierarchical Graph Mixtures*, or HGMs for short, that allows us to use hierarchical combinations of these models. Through this, we can express a wide variety of statistical models within a simple, consistent framework [Gillblad and Holst, 2004a].

We describe how to construct arbitrary, hierarchical combinations of mixture models and graphical models. This is possible through expressing a number of operations on finite mixture models and graphical models, such as calculating marginal and conditional distributions, in such a way that they are independent of the actual parameterisation of their sub-distributions. That is, as long as the sub-distributions (the component densities in the case of mixture models and factors in the case of graphical models) provide the same set of operations, it does not matter how these sub-distributions are represented. This allows us to create flexible and expressive statistical models, that we will see can be very computationally efficient compared to

more common graphical model formulations using belief propagation for inference. We will discuss the basic concepts and methodology involved, why this formulation provides additional modelling flexibility and simplicity, and give a few examples of how a number of common statistical methods can be described within the model.

We describe how to estimate the parameters of these models from data, given that we know the overall model structure. That is, whether we are considering a mixture of graphical models or a graph of mixture models *etc.* is assumed to be known. We are also not considering estimation of the number of components in a mixture or the graphical structure of a graph model, other than the generation of trees.

We also introduce a framework for encoding background knowledge, from *e. g.* experts in the area or available fault diagrams, based on Bayesian statistics. By noting that the conjugate prior distributions used in the framework can be expressed on a similar parametric form as the posterior distributions, we introduce a separate hierarchy for expressing background knowledge or assumptions without introducing additional parameterisations or operations. A good example of the practical use of this in combination with a hierarchy of graphs and mixtures can be found in chapter 5, where we create an incremental diagnosis system that both needs to incorporate previous knowledge and adapt to new data.

In this chapter, we will start by decribing some related work, followed by an introduction to statistical machine learning, mixture models, and graphical models in sections 3.3, 3.4, and 3.5 respectively. For readers already familiar with statistical machine learning and the concepts of mixture and graphical models, these sections are most likely not critical to the understanding of the following sections.

We then propose a way of combining graphical models and mixture models that allows us to create arbitrary hierarchical combinations of the two in section 3.6. In the following sections 3.7 and 3.8 we provide expressions necessary in this context for two leaf distributions, discrete and Gaussian, as well as a few examples of how some common statistical models and specific applications can be formulated within this general framework.

In section 3.9 we will describe the second part of the hierarchical framework, namely that of Bayesian parameter estimation and hierarchical priors. We first provide a brief introduction to Bayesian statistics, before going into the details on how we can assign priors hierarchically. Finally, we provide some concluding remarks and practical considerations.

3.2 Related Work

Most of the models related to the framework we present here have been suggested in order to manage multimodal data, in the sense that data represent samples from a number of distinct and different models. In relation to statistical models, this issue has been studied for both classification and density estimation tasks for some time. In the seminal work by Chow and Liu [Chow and Liu, 1968], a

classification method based on fitting a separate tree to the observed variables for each class is proposed. New data points are classified by simply choosing the class that has the maximum class-conditional probability under the corresponding tree model. In a similar approach, Friedman et al. starts with Naïve Bayesian classifiers instead of trees, and then consider additional dependencies between input attributes [Friedman *et al.*, 1997]. By then allowing for different dependency patterns for each class, their model is identical to Chow and Liu's proposition.

One extension to this model is to not directly identify the mixture component label with the class label, but to treat the class label as any other input variable. The mixture component variable remains hidden, leading to the use of one mixture model for each class and a more discriminative approach to classification. This is done in the Mixtures of Trees (MT) model [Meila and Jordan, 2000], showing good results on a number of data sets. In another generalization of the Chow and Liu algorithm, [Bach and Jordan, 2001] describe a methodology that utilises mixtures of thin junction trees. These thin junction trees allow more than two nodes, as used in normal trees, in each clique, while maintaining a structure in which exact inference is tractable.

In a density estimation or clustering setting, The Auto-Class model [Cheeseman and Stutz, 1996] uses a mixture of factorial distributions (a product of factors each of which depends on only one variable), often produces very good results on real data. Also related to density estimation, in [Thiesson *et al.*, 1997] the authors study learning simple Gaussian belief networks, superimposed with a mixture model to account for remaining dependencies. An EM parameter search is combined with a search for Bayesian belief models to find the parameters and structure of the model [Thiesson and C. Meek, 1999].

All the examples above are similar in the respect that they in essence specify one mixture layer of graphical models. Here, we will focus on building models with multiple levels in the hierarchy, such as mixtures of graphical models containing mixtures and so on, as this can greatly reduce the number of free parameters needed to efficiently model an application area. Most of the models discussed above also focus on only one type of variables, such as discrete variables for Mixtures of Trees and continuous (Gaussian) variables in [Thiesson *et al.*, 1997]. Auto-Class is a notable exception, as it uses products of discrete and Gaussian distributions in order to accept both discrete and continuous attributes. By introducing the possibility of using mixtures of continuous variables with little restriction in our models, we have the ability to effectively model data containing both discrete and continuous attributes, including joint distributions between the two.

It is possible to extend the notion of a mixture model by allowing the mixing coefficients themselves to be a functions of the input variables. These functions are then usually referred to as gating functions, the component densities as experts, and the complete model as a mixture of experts [Jacobs *et al.*, 1991]. The gating functions divide the input space into a number of regions, each represented by a different "expert" component density. Although this is already a very useful model, we can achieve even further flexibility by using a multilevel gating function

to give a hierarchical mixtures of experts [Jordan and Jacobs, 1994]. This model uses a mixture distribution in which each component density is itself a mixture distribution, possibly extending through several hierarchical layers. A Bayesian treatment of this model can be found in [Bishop and Svensen, 2003]. Within a neural network setting, the issue of multimodal data has also been studied in a manner similar to mixtures of experts in mixture density networks [Williams, 1996].

Similarly, in [Titsias and Likas, 2002], the authors use a three-level hierarchical mixture for classification, in which it is assumed that data is generated by a finite number of sources and in each source there is a number of class-labelled sources. It can be considered to be a special case of a mixture of experts classifier, but is also very similar to what we would refer to as an unsupervised mixture of supervised mixture models within our HGM framework.

Although many proposed mixtures of experts structures, such as the one above, are very similar to what we will be able to express using the models presented here, we will not allow the component label distribution to be a function of the input variables. This does put some constraints on the type of models we can express, but by being able to create arbitrary combinations of both graphical models and mixtures there is a wide variety of models that are equally difficult to express with a hierarchical mixture of experts. To some degree, the models presented here and hierarchical mixtures of experts are complementary approaches with similar aims, but the combination of them will not be further treated here.

The use of mixtures as factors in graphical models, as opposed to the use of graphs within mixtures as discussed above, is unusual. However, in [Sudderth *et al.*, 2003], mixtures of Gaussians are being used as factors in a graphical model to overcome the problem of inference intractability when continuous, non-Gaussian variables are being used within belief propagation. The model has been successfully applied to *e. g.* visual tracking tasks [Sudderth *et al.*, 2004] and node localisation in sensor networks [Ihler *et al.*, 2005].

A similar approach could easily be taken in the models we present here, the main difference being that we propose the use of several hierarchical levels of mixtures and graphs. We also try to avoid using Belief propagation explicitly in our models, as the number of component densities may grow exponentially when we repeatedly multiply mixtures models during the propagation. Avoiding this is made significantly easier due to the fact that we use mixtures rather than hidden nodes in our models. The use of mixtures make it easier to construct models for which propagation is not necessary for inference.

3.3 Statistical Methods

As we have already mentioned, there exists a vast variety of probabilistic methods in use today within a machine learning context. Still, it is perhaps not apparent at first glance why probabilistic models are needed within this area. Given a number of measurements, a classifier outputs a decision on *e. g.* whether or not a customers

credit application should be approved. It may not be obvious where the stochastic variables are in this situation, but rather the problem looks like one of function approximation: We need to determine the functional expression that calculates the class based on the inputs. Of course, this is not far from the truth. Probabilistic models do represent such a functional expression, expressed by operations on distribution functions. However, probabilistic models allow us to represent and reason about uncertainties in data in a very natural way.

So, let us now have a look at how some common machine learning tasks can be expressed within a statistical framework. Let us first consider the pattern classification problem. Assume that we can represent and calculate the probability distribution over a variable of interest y given a number of known variables $\boldsymbol{x} = \{x_1, x_2, \ldots, x_n\}$ as $P(y|\boldsymbol{x})$. To perform a classification we find the class $c \in y$ with the highest probability $P(y|\boldsymbol{x})$,

$$c_h = \underset{c \in y}{\operatorname{argmax}} P(y = c|\boldsymbol{x}) \tag{3.1}$$

This classification gives the smallest possible probability of error. Similarly, if we are predicting a real value and $P(y|\boldsymbol{x})$ is a continuous distribution, we can calculate the expected value v_h of y given $\boldsymbol{x}$,

$$v_h = E_y[P(y|\boldsymbol{x})] = \int_y yP(y|\boldsymbol{x})dy \tag{3.2}$$

This prediction gives the smallest possible expected squared error. However, in some cases this value may fall between several peaks in the conditional density, where the calculated expected value actually has a very low probability density. This means that it may still be in those cases more useful to compute the value v_h for which the density is the highest instead,

$$v_h = \underset{v \in y}{\operatorname{argmax}} P(y_v = v|\boldsymbol{x}) \tag{3.3}$$

Finally, if we want to detect anomalous behaviour using a statistical model M estimated from mainly normal data, we can determine whether the probability of a certain pattern given a model M is lower than a certain threshold κ,

$$P(\boldsymbol{x}|M) < \kappa \tag{3.4}$$

All patterns with a probability below κ are considered anomalous. κ may depend on the model and is selected so that a pattern generated by the model has a sufficiently small probability of being labelled as anomalous.

Note that all variables $\boldsymbol{x} = \{x_1, x_2, \ldots, x_n\}$ in the problems described above could either be observed in data, in which case we have a supervised learning problem, or hidden (non-observable), in which case we have an (at least partly) unsupervised learning problem.

Most of the problems discussed above are similar in the sense that they all rely on the calculation of marginal (*e. g.* $P(x) = \int_y P(x, y)$) and conditional (*e. g.* $P(y|x)$) distributions of a more complicated joint distribution. Here, we will often talk about these operations as *probabilistic inference*. Basically, if we have the joint distribution $P(\boldsymbol{x})$ over $\boldsymbol{x} = \{x_1, x_2, \ldots, x_n\}$, $\boldsymbol{x}^A \subset \boldsymbol{x}$, and $\boldsymbol{x}^B \subset \boldsymbol{x}$, any operation $P(\boldsymbol{x}^A|\boldsymbol{x}^B)$ or decision based thereof will be referred to as probabilistic inference. This means that the classification, prediction, and anomaly detection scenarios discussed above all rely on similar kinds of probabilistic inference, and can to a degree be treated similarly.

So, given that we can describe the joint distributions of our variables of interest and perform calculations on them, we can perform a number of common machine learning tasks. However, a brute force approach to describing these distributions is usually computationally infeasible. Consider describing the joint distribution of ten variables, each with ten outcomes. If we represent this directly by the probability of each combination of outcomes, we need 10^{10} parameters. From noting that adding another variable with ten outcomes means that we need ten times more parameters, we can conclude that this approach scales very badly. Apart from the fact that this direct approach is unsuitable from a computational standpoint, if we want to reliably estimate the probabilities from data, we are going to need unrealistically large data sets.

It is obvious that we need to simplify our representation of the joint distribution somewhat in practise. One very common approach is to assume that all attributes $\boldsymbol{x}$ are independent given the class attribute y, and use Bayes rule to rewrite the conditional distribution we are interested in as

$$P(y|\boldsymbol{x}) = \frac{P(y)P(\boldsymbol{x}|y)}{P(\boldsymbol{x})} \propto P(y)P(\boldsymbol{x}|y) = P(y)\prod_{i=1}^{n} P(x_i|y) \tag{3.5}$$

The distributions for each attribute given a specific class, $P(x_i|y)$, is significantly easier to represent and estimate than in the brute force approach. The method based on this independence assumption is usually referred to as a *naïve Bayesian classifier* [Good, 1950], and often generates very good results on a wide variety of data in spite of the rather restrictive independence assumption not being completely fulfilled [Hand and Yu, 2001].

We will now have a look at a couple of more general approaches to describe complex joint distributions, *mixture models* and *graphical models*.

3.4 An Introduction to Mixture Models

Finite mixtures of distributions, or *mixture models* [McLachlan and Basford, 1988; McLachlan and Peel, 2000], have proved to be a very flexible approach to statistical modelling. In essence, a finite mixture model is a way to describe a complex distribution by a weighted sum of simpler distributions. A mixture model can be

written as

$$P(\boldsymbol{x}) = \sum_{i=1}^{n} P(v_i)P(\boldsymbol{x}|v_i) = \sum_{i=1}^{n} \pi_i f_i(\boldsymbol{x}, \theta_i) \tag{3.6}$$

where π_i are the *mixture proportions*, $f_i(\boldsymbol{x}, \theta_i)$ the *component distributions*, and θ_i are the component specific parameters. A finite mixture is typically used to model data that is assumed to have arisen from one of n distinct groups. The group associated with a sample is either observable in the data or hidden, and the number of components n may be unknown.

One of the most commonly used forms of component distributions are Gaussians or normal distributions, which can be both univariate and multivariate (see section 3.7 for a closer description). Figure 3.1 shows an example of a mixture of four univariate Gaussians, all with different means, variances and corresponding mixing proportions.

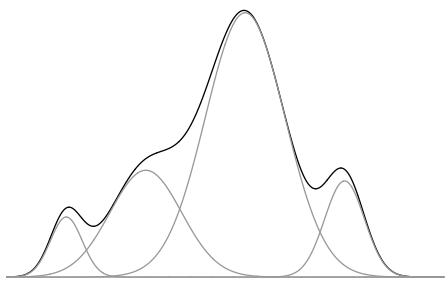

Figure 3.1: An example of a mixture of four Gaussian (normal) distributions. The black line represents the total mixture density, while the grey lines show the individual component densities.

Mixture models have been used for a wide range of applications [Titterington *et al.*, 1985] for quite some time. Already in the late 19th century, Pearson estimated the parameters of a mixture of two univariate Gaussian distributions using a method of moments [Pearson, 1894]. Today, with advances in methodology and computational resources, Bayesian or maximum likelihood methods are normally used for parameter estimation. We will return to parameter estimation shortly, but we will first introduce another useful interpretation of the mixture model.

In the *missing data* interpretation of the mixture model, each observation $\boldsymbol{x}^{(\gamma)}$ is assumed to arise from a specific but unknown component $z^{(\gamma)}$ of the mixture. We will refer to $z^{(\gamma)}$ as the *component-label* of γ. The mixture can be written in terms of these missing data, with $z^{(1)}, z^{(2)}, \ldots, z^{(n)}$ assumed to be realisations of independent and identically distributed discrete random variables $z_1, z_2, \ldots, z_n$ with the probability mass function

$$P(z^{(\gamma)} = i|\boldsymbol{\pi}, \boldsymbol{\theta}) = \pi_i \quad (\gamma = 1, \ldots, n;\, i = 1, \ldots, k) \tag{3.7}$$

If $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}$ are assumed to be independent observations given z_γ,

$$P(\boldsymbol{x}^{(\gamma)}|z_\gamma = i, \boldsymbol{\pi}, \boldsymbol{\theta}) = f_i(\boldsymbol{x}^{(\gamma)}; \theta_i) \tag{3.8}$$

Marginalising out the missing data $z_1, \ldots, z_n$ gives us the model

$$\begin{aligned} P(\boldsymbol{x}^{(\gamma)}|\boldsymbol{\pi}, \boldsymbol{\theta}) &= \sum_{i=1}^{k} P(z_\gamma = i|\boldsymbol{\pi}, \boldsymbol{\theta}) P(\boldsymbol{x}^{(\gamma)}|z_\gamma = i, \boldsymbol{\pi}, \boldsymbol{\theta}) \\ &= \sum_{i=1}^{k} \pi_i f_i(\boldsymbol{x}^{(\gamma)}, \theta_i) \end{aligned} \tag{3.9}$$

When there is a real and useful interpretation of the components of the mixture model, inferring the distribution over z itself may be of interest. This can be calculated through the proportionality

$$\begin{aligned} P(z_\gamma = i|\boldsymbol{x}^{(\gamma)}, \boldsymbol{\pi}, \boldsymbol{\theta}) &\propto P(z_\gamma = i|\boldsymbol{\pi}, \boldsymbol{\theta}) P(\boldsymbol{x}^{(\gamma)}|z_\gamma = i, \boldsymbol{\pi}, \boldsymbol{\theta}) \\ &\propto \pi_i f_i(\boldsymbol{x}^{(\gamma)}, \theta_i) \end{aligned} \tag{3.10}$$

which gives the normalised expression

$$P(z_\gamma = i|\boldsymbol{x}^{(\gamma)}, \boldsymbol{\pi}, \boldsymbol{\theta}) = \frac{\pi_i f_i(\boldsymbol{x}^{(\gamma)}, \theta_i)}{\sum_{l=1}^{k} \pi_l f_l(\boldsymbol{x}^{(\gamma)}, \theta_l)} \tag{3.11}$$

Even when there is no useful interpretation of the components of the mixture model, which may very well be the case if it is purely used to approximate a complex density, this formulation is still useful for notation and computation purposes.

Estimating the parameters of a mixture model can be done in several ways. If the component assignments can be observed in data, the estimation procedure can be reduced to estimating each component distribution from the samples that are labelled accordingly. The component distribution z itself can also be estimated from the observed component labels. However, if the component assignment cannot be observed in data, the procedure becomes somewhat more complicated.

The perhaps most common method for estimating the parameters in a mixture model where the component assignments cannot be observed in data is based on maximum likelihood principles. It is an iterative method usually referred to as *Expectation Maximisation* [Dempster *et al.*, 1977], or EM for short. It strives to maximise the likelihood of the model, *i. e.* the probability of the training data D given the parameters of the model M, $P(D|M)$. Generally, we assume that all samples are independent and identically distributed, meaning that the probability of getting all data samples is simply the product of the probability of each individual sample. The expression we want to maximise therefore becomes

$$P(D|\boldsymbol{\theta}) = \prod_{\gamma} P(\boldsymbol{x}^{(\gamma)}|\boldsymbol{\theta}) \tag{3.12}$$

where $\boldsymbol{x}^{(\gamma)}$ are samples of the training data set. The Expectation Maximisation algorithm maximises this expression by iterating a component probability calculation step with an estimation step. First, the algorithm is initialised by setting the parameters of all component distribution to different values, usually randomised in a way that is suitable for the domain and type of parameterisations. Then equation 3.11 is used to calculate how much each sample belongs to each component, given the initial parameters. This is referred to as the *expectation* step. Thereafter, the parameters of each component distribution are estimated where each sample of the data set is weighed by the probability of the sample being generated by the component, as calculated in the expectation step. This procedure is referred to as the *maximisation* step. These steps are repeated until all parameters have converged.

Instead of initialising the parameters of the model to random values, it is common to initialise the component distributions by estimating them from a randomly selected subset of the data [Holst, 1997]. This saves us from describing suitable domains for each parameter to randomise, and allows us to describe and implement the complete algorithm without knowing the actual parameterisations of the component distributions. This is, as we will see further on, a very useful property.

Although the convergence may be quite slow, it can be proven that the parameters will converge eventually to a local maximum [Dempster *et al.*, 1977]. There have been a number of methods proposed to make finding the actual global maximum more likely, and to speed up the convergence of the procedure [Louis, 1982]. These may be highly usable in situations where models need to be repeatedly re-estimated, but will not be further treated here.

Mixture models are related to and can in some cases represent a number of other common methods within machine learning. One rather obvious similarity is that to *radial basis function* neural networks, which are at least partially represented by a sum of localised functions, very similar to the representation of mixture models. With a number of components localised at different places in the input space, the mixture model can also be used to provide a soft interval coding not too different from *fuzzy sets* [Zadeh, 1965]. All data points will have a probability of being generated by each component, analogous to the degree of fuzzy membership.

Also, if we allow one component in a mixture for each data point in the training data set, and assign each component equal probability, we have a *Parzen density estimator* [Parzen, 1962]. The Parzen density estimator commonly assumes Gaussian component distributions with a fixed, pre-assigned variance, which in turn makes it similar to instance based methods using Gaussian, or for that matter any kernel function, when calculating distances between patterns. This similarity will be used and further explained in chapter 5.

3.5 An Introduction to Graphical Models

Another approach to describing complex distributions is through *graphical models*. Somewhat simplified, a graphical model represents dependencies among random

variables as a graph, in which nodes represent variables and edges dependency relations. Let us now have a brief look at a few common types of these models.

Types of Graphical Models

Let us start with the perhaps most common graphical model, the *Bayesian network* [Pearl, 1988]. The Bayesian network can be described as a directed acyclic graph, where the nodes represent variables and the directed edges represent statistical dependence relations (see figure 3.2 for a simple example). The variables may both be observed through measurements or not, and in the latter case we would refer to the variable as *hidden* or *latent*. The joint probability distribution over the random variables $\boldsymbol{x} = x_1, x_2, \ldots, x_N$ can be written as

$$P(\boldsymbol{x}) = \prod_{k=1}^{N} P(x_k|\boldsymbol{w}_k) \tag{3.13}$$

where $\boldsymbol{w}_k$ are the *parents* of x_k, *i. e.* the variables that have directed edges connected to x_k. This expression is derived from the defining property of a Bayesian network, which is that each variable x_k is conditionally independent of any combination of its *nondescendats* $\boldsymbol{n}_k$ given its parents. More formally, we can write this as

$$P(x_k|\boldsymbol{w}_k, \boldsymbol{u}) = P(x_k|\boldsymbol{w}_k) \qquad \forall \quad \boldsymbol{u} \subseteq \boldsymbol{n}_k \tag{3.14}$$

The nondescendants $\boldsymbol{n}_k$ are all variables that are not descendants of x_k, *i. e.* variables that are not its children, its children's cilhdren and so on. In a sense, the Bayesian network can be thought of as a generalisation of a *Markov chain*, where each variable x_{k+1} is assumed to depend only on its immediate predecessor x_k,

$$P(\boldsymbol{x}) = P(x_1)P(x_2|x_1)P(x_3|x_2)P(x_4|x_3)\ldots \tag{3.15}$$

To fully specify a Bayesian network, it is necessary to specify for each variable x_k the probability distribution for x_k conditional on its parents $\boldsymbol{w}_k$, $P(x_k|\boldsymbol{w}_k)$. These distributions may have any form, but are usually assumed to be either discrete or Gaussian, since this simplifies calculations. Sometimes all the parameters of these distributions are known, but usually, they have to be estimated from data. This is rather straightforward if there are no latent variables. If latent variables are present, estimation becomes rather more complicated. A common approach to estimation in this situation is to use the expectation maximisation algorithm, in a similar manner as for the mixture models we discussed earlier.

A *Markov Random Field* (MRF) is an undirected graphical model which is defined by a local Markov property saying that, given its neighbours, a variable is independent of all other variables in the graph. If we let $\boldsymbol{l}_i$ define the neighbours of variable x_i, we can write this as

$$P(x_i|\boldsymbol{x} \setminus x_i) = P(x_i|\boldsymbol{l}_i) \tag{3.16}$$

where $\boldsymbol{x} \setminus x_i$ denotes all variables in $\boldsymbol{x}$ except x_i. The joint distribution over all variables in a MRF can be written as a product of *clique potentials*, where a clique is a fully connected subgraph such that for every two vertices in the subgraph, there exists an edge connecting the two. The clique potential is then a function over the variables of this clique. Although the clique potentials may have a probabilistic interpretation, such as marginals over the included variables included in the clique, they are generally not restricted to this. If there are M cliques, the distribution can be written as

$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{m=1}^{M} \psi_m(\boldsymbol{x}_m) \tag{3.17}$$

where $\psi_j(\boldsymbol{x}_j)$ is the potential for the clique j. Z is a normalising constant specific for the each MRF, where

$$Z = \sum_{x \in \boldsymbol{x}} \prod_{m=1}^{M} \psi_m(\boldsymbol{x}_m) \tag{3.18}$$

A MRF can express some dependencies that a Bayesian network cannot, such as some cyclic dependencies, but it is still not possible to express all factorisations of the probability density. See *e. g.* [Smyth *et al.*, 1997] for examples of what dependency structures that can be expressed by directed graphical models but not by undirected graphical models and vice versa.

A more general type of graphical model which can represent a larger number of factorisations is the *factor graph*. A factor graph explicitly indicates how a joint distribution function factors into a product of functions of a smaller subset of variables. We write the probability distributions over a set of variables $\boldsymbol{x}$ as

$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{m=1}^{M} f_m(\boldsymbol{x}_m) \tag{3.19}$$

where Z is a normalising constant, just as in a Markov random field. $\boldsymbol{x}_m$ is the set of variables that a *local function* $f_m(\boldsymbol{x}_m)$ depends on. These local functions may be marginal or conditional distributions, or potentials as in a Markov random field. A factor graph is described by a bipartite graph, *i. e.* a graph with two kinds of vertices where edges can only connect vertices of different type. One set of vertices represent the variables $\boldsymbol{x}$, while another set represents the local functions $f_1(\boldsymbol{x}_1), f_2(\boldsymbol{x}_2), \ldots, f_M(\boldsymbol{x}_M)$. In figure 3.2, one directed and one undirected graph are shown together with examples of their representation as a factor graph. The round nodes represent variables and the square nodes the factors.

A factor graph may have directed edges to indicate conditional dependence similar to a Bayesian network. Factor graphs are more general compared to both Bayesian networks and Markov random fields in terms of expressing factorisations of the complete joint distribution.

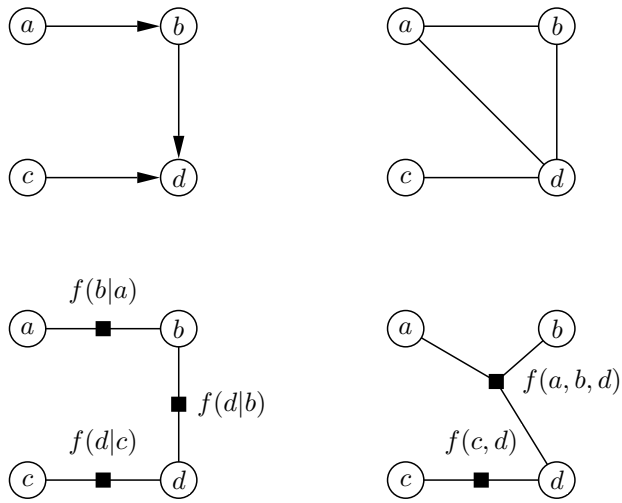

Figure 3.2: Different types of graphical models. The upper left graph shows a simple directed Bayesian network, with a factor graph representation of the same graph directly below. The upper right graph shows a markov random field, again with a factor graph representation below.

Inference in Graphical Models

Exact inference in a graphical model is usually performed using *message passing* algorithms, where simple messages are passed locally along the edges of the graph. They were first described for singly connected models [Pearl, 1986, 1988; Spiegelhalter, 1986; Lauritzen and Spiegelhalter, 1988], and are commonly referred to as the (generalised) *forward-backward algorithm*. This algorithm can be relaxed somewhat to cover a large number of graphical models into the *sum-product algorithm*, which we will have a look at here. It is in itself a special case of the *generalised distributive law* [Aji and McEliece, 2000], a general message passing algorithm of which a large number of common algorithms are special cases.

First, let us assume that we can write the distribution as a product of M factors as in equation 3.19. We can now view the sum-product algorithm as that it re-expresses the factorisation of equation 3.19 as another function with $M + N$ factors,

$$P(\boldsymbol{x}) = \prod_{m=1}^{M} \phi_m(\boldsymbol{x}_m) \prod_{n=1}^{N} \psi_n(x_n) \tag{3.20}$$

where each factor $\phi(\boldsymbol{x}_m)$ is associated with one factor node and each factor $\psi(x_n)$ is associated with one variable node. Initially, we set $\phi(\boldsymbol{x}_m) = f_m(\boldsymbol{x}_m)$ and $\psi(x_n) = 1$. We then update the factorisation according to

$$\psi_n(x_n) = \prod_{m \in S_n^f} \mu_{m \to n}(x_n) \tag{3.21}$$

$$\phi_m(\boldsymbol{x}_m) = \frac{f_m(\boldsymbol{x}_m)}{\prod_{n \in S_m^v} \mu_{m \to n}(x_n)} \tag{3.22}$$

where S_n^f is the set of factors in which variable n participates and S_m^v the set of variables in factor m. Each message $\mu_{m \to n}$ is calculated using

$$\mu_{m \to n} = \sum_{\boldsymbol{x}_m \setminus n} \left(\phi_m(\boldsymbol{x}_m) \prod_{n' \in S_m^v} \psi_{n'}(x_{n'}) \right) \tag{3.23}$$

where $\boldsymbol{x}_m \setminus n$ is the set of all variables in $\boldsymbol{x}_m$ with x_n excluded. In essence, equation 3.23 refers to a factor to variable message, while equation 3.21 refers to a variable to factor message. Messages are created in accordance with the above rules only if all the messages on which it depends are present. If the graph does not contain any loops, the expressions will converge after a number of steps equal to the diameter of the graph. When the graph does contain loops, the algorithm does not necessarily converge, and does not in general produce the correct marginal function. However, this type of *loopy belief propagation* is still of very great practical importance, as for many graphs it converges to usable approximations of the marginals relatively quickly.

It is worth noticing that in general, exact inference in multiple connected networks is NP-hard [Cooper, 1990]. In fact, even approximate inference to a certain precision in these networks is NP-hard [Dagum, 1993]. As exact inference can be difficult, it is often necessary to resort to approximate methods, such as a *Monte Carlo* approach. If we can draw random numbers for the unobserved variables $\boldsymbol{u}$ from the conditional distribution given the observed variables $\boldsymbol{v}$, $P(\boldsymbol{u}|\boldsymbol{v})$, then the relative frequencies of these random numbers can be used to approximate the distribution. Generating random numbers is relatively easy from a Bayesian network, as its lack of cycles mean that it is always possible to form an *ancestral ordering* $x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(N)}$ of the variables, where $x_{\pi(i)}$ is a permutation map such that the descendants of each variable come later in the ordering. Thus, as long as a value for x_k can be drawn according to $P(x_k|\boldsymbol{w}_k)$, we can simply perform random number generation for each variable using the ancestral ordering until the whole vector $\boldsymbol{x}$ has been drawn. In our brute force approach to Monte Carlo inference, we generate a large number of complete random vectors this way and compile a histogram over the variables $\boldsymbol{u}$. Unfortunately, the number of samples that we need to generate to arrive at a decent approximation of the distribution is, except for some unrealistically simple graphs, very large. Also, for many types of graphs, this

type of ancestral simulation is not possible, and the generation of each sample may be very time consuming.

In these cases, a Markov chain Monte Carlo (MCMC) method is often used. Given a number of known values $\boldsymbol{v}$, we can construct a temporal sequence $\boldsymbol{u}^1, \boldsymbol{u}^2, \ldots$ of the unknown variable values $\boldsymbol{u}$ whose stationary distribution is equal to $P(\boldsymbol{u}|\boldsymbol{v})$. By collecting the values in the sequence, we can find an approximation of the actual distribution *e. g.* through a histogram. Although it is desirable to run the simulation until we have reached the point where the stationary distribution can be considered to be sufficiently stable, it is more common to accept an approximation by terminating the calculation earlier so that computation time is kept reasonable.

As an example, we can take a simple MCMC algorithm, the *Gibbs sampler*. In this, only one variable $u_i \in \boldsymbol{u}$ is changed between successive states $\boldsymbol{u}^{s-1}$ and $\boldsymbol{u}^s$. The assumption is that although the full distribution $P(\boldsymbol{x})$ is intractable to calculate, the univariate conditional distributions $P(x_i|\{x_j\}_{j=1,j\neq i}^N)$ are not, as *e. g.* in a Bayesian network. Thus, if at state s we have decided to modify $x_i \in \boldsymbol{u}$, we draw a random value according to

$$P(x_i|\{x_j = x_j^{s-1}\}_{j=1,j\neq i}^N) \tag{3.24}$$

which often can be acquired with relative ease. However, although benefitting from the fact that there are no adjustable parameters, Gibbs sampling suffers from the fact that the space is explored through a slow random walk. This also applies to similar approaches, such as the *Metropolis* method [Metropolis *et al.*, 1953] and, to some degree, *slice sampling* (see *e. g.* [Neal, 1993; Frey, 1997] for closer descriptions).

Another important approximate inference method is *variational inference*. In contrast to Monte Carlo models, it does not use a stochastic sampling approach, but rather tries to approximate a marginal of interest with a simpler model that approximates the actual marginal sufficiently well. If we have a set of known variables $\boldsymbol{v}$, unknown variables of interest $\boldsymbol{u}$, and want to find $P(\boldsymbol{u}|\boldsymbol{v})$, we introduce a new distribution $Q(\boldsymbol{u}|\boldsymbol{v})$ that approximates $P(\boldsymbol{u}|\boldsymbol{v})$. We adjust the parameters of $Q(\boldsymbol{u}|\boldsymbol{v})$ to minimize the distance, usually taken as the Kullback-Leibler divergence, between the actual distribution and its approximation. This can be performed through a gradient search, such as conjugate gradient [Hestenes and Stiefel, 1952]. After this, $Q(\boldsymbol{u}|\boldsymbol{v})$ is used as a computationally more tractable approximation of $P(\boldsymbol{u}|\boldsymbol{v})$.

The form of the approximating distribution is usually chosen to be relatively simple, but naturally has a large effect on both the quality of the inference and the computational complexity. The choice of distance measure also depends on the problem, but can be crucial for the quality of the results. For a more detailed introduction to variational methods, see *e. g.* [Jordan *et al.*, 1999; Jaakkola, 2000].

3.6 Hierarchical Graph Mixtures

Let us now turn our attention to how we can combine mixture models and graph models into Hierarchical Graph Mixtures, in which we represent probability distributions by hierarchical combinations of graphs and mixture models. The key objective of these Hierarchical Graph Mixtures is naturally the same as both that of graph models and mixture models themselves: We want to effectively describe a complex probability distribution over a number of attributes. To quickly recapitulate earlier descriptions, in a mixture model, this is performed by representing this complex distribution as a sum of simpler distributions, and in a graph model as a product expansion consisting of joint distributions for some variables, assuming conditional independence between the rest. The benefits of combining these models relates to increased expressive power and ease of implementation while at the same time reducing the number of parameters.

The difference in approach towards describing the complete distribution makes the mixture model and graph model natural complements. In general, we could say that a mixture model groups samples, with the intention to simplify the description of the complete distribution, while graph models group combinations of attributes (see figure 3.3). If we are able to combine mixture models and graph models hierarchically, without limitation to the number of levels in this hierarchy, we have a very powerful modelling framework potentially capable of expressing very complicated distributions in a structured manner.

The method of combining graphs and mixtures allows us to easily and intuitively describe a large number of useful statistical models. Models such as mixtures of trees can easily be expressed and extended upon, as well as a wide variety of both similar and very different models.

Another important benefit is found in the implementation of the modelling framework. Since both the graph model and mixture model, our only higher order components, can be expressed independently of the parameterisation of the component distributions, more explicitly the component distributions of a mixture model and the factors of a graph model, implementation of a general and extensible system becomes relatively straightforward. Extending this system simply becomes a matter of implementing new basic distributions.

However, the perhaps most important aspect of the framework is that it allows us to easily express a phenomenon common in data extracted from complex systems, in that we can use different dependency structures for different clusters in data. Figure 3.4 shows a very simple example where in one cluster in data a certain graphical representation is suitable, and in another cluster another representation is more suitable. This differs in objective from modeling the actual (causal) relations in data. Instead, the aim of the exercise is, to be blunt, not to model the world perfectly, which would include finding the true causal dependencies between variables, but to model data and the dependencies that we can measure reliably. In this situation, being able to use a mixture of graphical structures allows for more efficient representations.

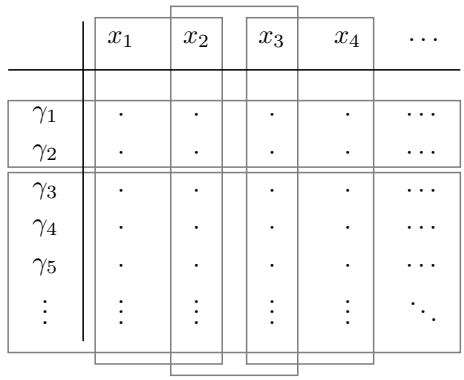

Figure 3.3: A simplified view of the complementary nature of mixture models and graph models. The columns $x_1, x_2, \ldots$ represent attributes, while the rows $\gamma_1, \gamma_2, \ldots$ represent samples. The graph model simplifies the distribution by grouping attributes, *e. g.* $\{x_1, x_2\}$ and $\{x_2, x_3\}$, and the mixture model by grouping samples, $\{\gamma_1, \gamma_2\}$ etc. By building hierarchical models, we can construct arbitrary groups of attributes and samples.

In practise, this type of situation with different dependence structures for different modes arise in real data *e. g.* for chemical production plants. Here, a chemical is often produced using different cost functions at different times. These different cost functions impose very different targets for the control system, effectively changing dependencies between measured attributes drastically. This is effectively modeled as a mixture of several different graphical structures. Although modelling the complete set of causal dependencies in one graph may be theoretically possible, the added complexity of doing so is often prohibitive, or will give degraded performance.

Using Mixture Models with Graphical Models

In the general graphical model setting, a mixture model is often described as a graph with a hidden node representing the mixture component label. Thus, it may not be directly apparent why introducing mixture models separately is useful. There are, however, a number of motivations to do this.

One important motivation for combining mixture models and graphical models is the opportunity to use, hierarchically, different graphical structures for different clusters in data. These hierarchies can be difficult to formulate and implement if we want to strictly use the graph model formulation. Introducing mixture models also allows us to write many models without using hidden nodes. Using *e. g.* a

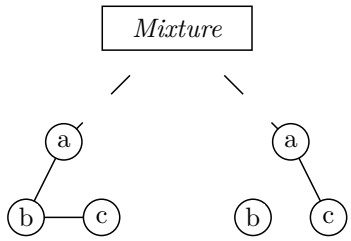

Figure 3.4: Using mixtures of graphical models, we can use different graph models for different clusters in data. Here two different graphs represent the dependencies in two different clusters. In the first, attributes a and c are conditionally independant given b, and in the second attribute b is independant of a and c. Creating a mixture of these graphical models allows for efficient representation of the joint distribution.

mixture of graphical structures instead of a hidden node within a network makes it easier to calculate marginals without resorting to Belief propagation, resulting in computationally more efficient models.

Mixture models also allow us to easily model joint distributions of *e. g.* discrete and continuous attributes, which are quite common when modelling real world data. These joint distributions are easily expressed as a mixture in which the discrete variables express the component label distribution, while the corresponding components are *e. g.* Gaussian distributions over the continuous variables.

Similarly, mixtures also allow us to easily express more complex factors within our graphical models. For example, compared to using Gaussians to describe the factors in a graphical model, the use of mixtures of Gaussians allow us to express much more complex, multimodal factor distributions without severily increasing model complexity.

Basic and Necessary Operations

A useful property, shared between both mixture models and graph models, is that most important operations can be expressed independently of the parameterisations of the sub-distributions that comprise them, *i. e.* the factors of a graph model and the component distributions of a mixture model. This includes estimation and calculation of marginals and conditionals. This relative independence of the sub-distributions make it possible to construct hierarchical combinations of these two

models without sacrificing functionality, *e. g.* graphs over mixtures and mixtures of graphs but also more elaborate variants like mixtures of graphs over mixtures.

Let us now go through the operations we would like to be able to perform on our distributions. Going back to section 3.3, it is apparent that we want to be able to calculate *conditional distributions* so that we are able to perform prediction and classification tasks. That is, we want to be able to calculate the distribution over one or several attributes given that we know the joint distribution and the values of a subset of the other attributes. Although not always trivial, calculation of conditionals turn out to be relatively straightforward in both mixtures and graphs.

Second, we would also like to be able to calculate *marginal distributons*, in order to study the distribution of a subset of the attributes given the joint distribution when we do not know the values of the rest of the attributes. This does not necessarily pose any real problems either for the mixture or the graph model.

Finally, to be useful in our data driven approach, there must be a way to *estimate* the parameters of a HGM from data, so that we then can use the estimated parameters to make statistical predictions. Naturally, there are several different levels of estimation to be considered, *e. g.* whether the graph structure of a graph model or the number of components in a mixture model should be estimated or considered given beforehand. We will focus on estimation of the distributions with a given graph structure and known number of components in the mixtures, but the models described are by no means limited to this scenario (see chapter 4 for a further discussion).

As we will see, though, the operations above are not quite sufficient in themselves. The mixture model and graph model each demands one additional operation of its components if we want to be able to perform the operations discussed above. The estimation and calculation of conditional distributions of mixture models demand that we are able to *calculate the probability density* for a certain sample. Also, calculation of marginals and conditionals in a graph model require that we are able to *multiply* different distributions over the same attributes. Note that here, by multiplication of distributions we refer to the combined operation of actually multiplying the distributions and then normalising the result.

Finite Mixture Models

We will start by generalising the notion of a mixture model that we introduced in section 3.4 somewhat to allow the component-label distribution $\boldsymbol{z}$ to be multivariate. This is important for consistency of the framework, and for the ability to describe joint distributions between *e. g.* several discrete and continuous attributes.

Assume that we have a multi-variate random variable $\boldsymbol{y}$ over the set of observed variables $S_y = \{y_1, y_2, \ldots, y_n\}$, and a multi-variate discrete distribution $\boldsymbol{z}$ over the set of variables $S_z = \{z_1, z_2, \ldots, z_m\}$ with number of possible outcomes of the marginals $k_1, k_2, \ldots, k_m$. The random variable $\boldsymbol{y}$ can be of any form, continuous or discrete, and represents the complex distribution we would like to describe by a sum of simpler distributions. The component-label variable $\boldsymbol{z}$, on the other hand,

must be a discrete variable. The probabilities of each individual outcome in $\boldsymbol{z}$ represent the weights in the weighted sum. The variables $z_1, z_2, \ldots, z_m$ can each be either observable in data or not, and in the latter case we will refer to the variable as *hidden*. To make notation easier, when we write a sum using index $\boldsymbol{i} \in S_z$ or similar, we will assume the sum to be over all indexes $\boldsymbol{i} = i_1, i_2, \ldots, i_m$ belonging to the m dimensional discrete distribution containing the variables S_z, $\sum_{\boldsymbol{i} \in S_z} \ldots = \sum_{i_1}^{k_1} \sum_{i_2}^{k_2} \ldots \sum_{i_m}^{k_m}$.

The basic definition of a finite mixture model over the random variables $\boldsymbol{y}$ and $\boldsymbol{z}$ can be written in shorthand as

$$f(\boldsymbol{y}, \boldsymbol{z}) = \sum_{\boldsymbol{i} \in S_z} \pi_{\boldsymbol{i}} f_{\boldsymbol{i}}(\boldsymbol{y} | \theta_{\boldsymbol{i}}) \tag{3.25}$$

where $f(\boldsymbol{y}, \boldsymbol{z})$ is the modelled density, $f_{\boldsymbol{i}}(\boldsymbol{y})$ the component densities, $\theta_{\boldsymbol{i}}$ the parameters for the component densities and $\pi_{\boldsymbol{i}}$ nonnegative quantities that sum to one and describe the *mixing proportions* or *weights* of the components. The mixing proportions can be viewed as individual probabilities of the outcomes of the general discrete distribution $\boldsymbol{z}$, and the total number of component densities amounts to the product of the number of outcomes in this discrete distribution, $\prod_{i=1}^{m} k_i$. In this context, it is easier to interpret the mixture model based on the missing data interpretation discussed in section 3.4, where each observation is assumed to arise from a specific component $\boldsymbol{z}$.

In most descriptions of mixture models, the component label distribution $\boldsymbol{z}$ is uni-variate, *i. e.* $m = 1$ with k_1 number of components, and is considered hidden. We will need this more general description though when we do general combinations of variables in graph models.

Supervised Mixture Models

Here we will make a distinction between *supervised* and *unsupervised* mixture models. In a supervised mixture model, the outcomes of the discrete distribution $\boldsymbol{z}$ are known from data, while they in the unsupervised model are hidden. We can of course also construct combinations of these two extremes, where some of the variables in the component label distribution are unknown and some not. We will refer to these models as *partially supervised*, and while their estimation will not be explicitly described here the formulations for the supervised and unsupervised models can be easily generalised to this case.

Estimation of a supervised mixture model simply amounts to estimating the component densities $f_{\boldsymbol{i}}(\boldsymbol{y}|\theta_{\boldsymbol{i}})$ on data points in which the outcome of the component label variable $\boldsymbol{z}$ has the corresponding outcome $z_{\boldsymbol{i}}$. Such a mixture model always has the same number of component densities as there are outcomes in the component label variable. There is no restriction on the dimensionality of the discrete variable $\boldsymbol{z}$. The mixture model just need to have the same number of component densities as the total number of outcomes in $\boldsymbol{z}$, *i. e.* in the worst case the product of the number of outcomes of all marginals.

The supervised mixture model comes in naturally in many kinds of modelling. For example, it is a convenient way to represent joint distributions of continuous and discrete variables. The component densities in this case are representations of joint distributions of the continuous attributes, and the mixing proportions describe the joint distribution of the discrete variables.

Unsupervised Mixture Models and the EM Algorithm

In the unsupervised mixture model we are left with the problem of estimating both the mixing proportion distribution $\boldsymbol{z}$ and the parameters of the individual components. As there is no closed form solution, this is usually performed using the *Expectation Maximisation* (EM) algorithm [Dempster *et al.*, 1977; McLachlan and Peel, 2000], trying to maximise the *likelihood* of the parameters, *i. e.* the probability of data given the model, $P(D|M)$. The EM algorithm is an iterative method consisting of two steps: The *expectation* step, where the probability of each sample $(\boldsymbol{y}^{(\gamma)}, \boldsymbol{z}^{(\gamma)})$ belonging to each component distribution given its parameters $f_{\boldsymbol{i}}(\boldsymbol{y}|\theta_{\boldsymbol{i}})$ is calculated, and the *maximisation* step, where new parameter settings for all components and the mixing proportion distribution $\boldsymbol{z}$ are estimated according to the probability of each sample belonging to each of the components. The probability of a sample $(\boldsymbol{y}^{(\gamma)}, \boldsymbol{z}^{(\gamma)})$ being generated by component density $v_{\boldsymbol{i}}$ where $\boldsymbol{z}^{(\gamma)}$ is unknown is given by

$$P(v_{\boldsymbol{i}}|\boldsymbol{y}^{(\gamma)}) = \frac{\pi_{\boldsymbol{i}} f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}|\theta_{\boldsymbol{i}})}{\sum_{\boldsymbol{j}\in S_z} \pi_{\boldsymbol{j}} f_{\boldsymbol{j}}(\boldsymbol{y}^{(\gamma)}|\theta_{\boldsymbol{j}})} \tag{3.26}$$

As we discussed earlier, the initialisation of the algorithm is often performed by randomising the parameters of the component distributions and then starting iterating at the expectation step. This is not a particularly good solution in this case, since we want the estimation procedure to be independent of the parameterisation of the component distributions. Instead, we can initialise the procedure by randomising how much each sample belongs to each component, and then starting the iteration at the maximisation step. By doing this, we can perform estimation of the mixture model regardless of the type and parameterisation of the component densities. See Appendix A for proof that this holds for relevant distributions.

Marginals, Conditionals and Multiplication of Mixtures

Calculating the marginal and the conditional of a mixture might result in either a discrete distribution, a distribution of the same type as the component distributions or a new mixture model. For the sake of clarity, we will describe all three cases for both the marginal and the conditional.

Let S_y^A and S_y^B be disjoint complementary subsets of S_y, $S_y^A \subset S_y$, $S_y^B \subset S_y$, $S_y^A \cap S_y^B = \varnothing$, and $S_y = S_y^A \cup S_y^B$. Similarly, let S_z^A and S_z^B be disjoint complementary subsets of S_z. Also assume that we want to calculate the marginal for variables in S_y^A and S_z^A. Calculation of the marginal involves integrating or

summing over all variables in S_y^B and S_z^B. Now, let us start with calculating the marginal of a mixture model for two special cases. First, if we are calculating the marginal only of attributes in S_z, *i. e.* $S_y^A = \varnothing$, the expressions simply become

$$f(S_z^A) = P(\boldsymbol{z}^A = \boldsymbol{i}) = \sum_{\boldsymbol{j} \in S_z^B} \pi_{\boldsymbol{i},\boldsymbol{j}} \tag{3.27}$$

which is the marginal of the discrete distribution. If on the other hand S_z^A is empty, *i. e.* the marginal is calculated over only a subset of the variables in S_y, the expression becomes

$$f(S_y^A) = \sum\nolimits_{\boldsymbol{i} \in S_z^B} \pi_{\boldsymbol{i}} \int_{S_y^B} f_{\boldsymbol{i}}(S_y^A, S_y^B | \theta_{\boldsymbol{i}}) \tag{3.28}$$

The result can be viewed as a new mixture with the same number of components as the original mixture, with one major difference: the outcomes of the component label distribution of the resulting mixture, although with exactly the same distribution as in S_z, can no longer be considered to correspond to any observable attributes in data. They must therefore now be viewed as hidden. Finally, the general expression for the marginal of a mixture model over the variables S_y^A and S_z^A can be written as

$$f(S_y^A, S_z^A) = \sum_{\boldsymbol{i} \in S_z^A} \left(\sum\nolimits_{\boldsymbol{k} \in S_z^B} \pi_{\boldsymbol{i},\boldsymbol{k}} \right) \left(\sum_{\boldsymbol{j} \in S_z^B} \frac{\pi_{\boldsymbol{i},\boldsymbol{j}}}{\sum_{\boldsymbol{l} \in S_z^B} \pi_{\boldsymbol{i},\boldsymbol{l}}} \int_{S_y^B} f_{\boldsymbol{i},\boldsymbol{j}}(S_y^A, S_y^B | \theta_{\boldsymbol{i}}) \right) \tag{3.29}$$

Comparing this expression with (3.25), we can see that it represents a mixture of mixture models, *i. e.* a new mixture model with mixture models as component distributions. The component densities of its mixture components correspond to the marginals $f_{\boldsymbol{i}}(S_y^A)$ of the component densities in the original mixture, and the integral over S_y^B represents the calculation of these marginals. Note that the same situation that occurs in equation 3.28, *i. e.* that we have to regard a set of discrete variables as hidden in the resulting model, may also occur in the general expression. All variables that no longer correspond to an observable attribute must be considered unknown in the resulting mixture model. This is also the reason why the result is a mixture of mixture models: The variables that used to be considered known must now be represented as a mixture.

Now, let us consider the conditional of a mixture model given the values of variables S_y^B and S_z^B. If we know the outcomes of all discrete variables, *i. e.* $S_z^A = \varnothing$, this conditional is simply

$$f(S_y^A | S_y^B, S_z^B) = f_{\boldsymbol{i}}(S_y^A | S_y^B, \theta_{\boldsymbol{i}}) \tag{3.30}$$

where $\boldsymbol{i}$ represents the outcome given by the known discrete variables and $f_{\boldsymbol{i}}(S_y^A | S_y^B, \theta_{\boldsymbol{i}})$ the conditional of the corresponding component distribution. The

result is on the form of this component distribution, and not a new mixture model. If $S_y^A = \varnothing$, all component distribution variables are known, we calculate the conditional using $P(\boldsymbol{y}|\boldsymbol{x}) \propto P(\boldsymbol{y})P(\boldsymbol{x}|\boldsymbol{y})$ and normalising,

$$f(S_z^A|S_y^B, S_z^B) = P(\boldsymbol{z}^A = \boldsymbol{i}|\boldsymbol{z}^B, \boldsymbol{y}^B) = \frac{\pi_{\boldsymbol{i},\boldsymbol{j}} f_{\boldsymbol{i},\boldsymbol{j}}(S_y^B|\theta_{\boldsymbol{i},\boldsymbol{j}})}{\sum_{\boldsymbol{k}\in S_z^A} \pi_{\boldsymbol{k},\boldsymbol{j}} f_{\boldsymbol{k},\boldsymbol{j}}(S_y^B|\theta_{\boldsymbol{k},\boldsymbol{j}})} \tag{3.31}$$

where $\boldsymbol{j}$ is the known outcome in S_z^B. This is simply a discrete distribution over the variables in S_z^A, weighted by the probability of the component distributions. The general expression for the conditional of a mixture model given the values of variables S_y^B and S_z^B can then be written as

$$f(S_y^A, S_z^A|S_y^B, S_z^B) = \sum_{\boldsymbol{i}\in S_z^A} \frac{\pi_{\boldsymbol{i},\boldsymbol{j}} \int_{S_y^A} f_{\boldsymbol{i},\boldsymbol{j}}(S_y^A, S_y^B|\theta_{\boldsymbol{i},\boldsymbol{j}})}{\sum_{\boldsymbol{k}\in S_z^A} \pi_{\boldsymbol{k},\boldsymbol{j}} \int_{S_y^A} f_{\boldsymbol{k},\boldsymbol{j}}(S_y^A, S_y^B|\theta_{\boldsymbol{k},\boldsymbol{j}})} f_{\boldsymbol{i},\boldsymbol{j}}(S_y^A|S_y^B, \theta_{\boldsymbol{i},\boldsymbol{j}}) \tag{3.32}$$

where $\boldsymbol{j}$ represents the known outcome in S_z^B. The result is interpreted as a new mixture model, where the component densities are the conditionals of the original component densities, $f_{\boldsymbol{i},\boldsymbol{j}}(S_y^A|S_y^B, \theta_{\boldsymbol{i},\boldsymbol{j}})$, for which the corresponding discrete variables are still unknown. The expression for the component probabilities is in fact simply the conditional of the discrete distribution compensated with the likelihoods of the components densities, and is normalised to produce a valid distribution.

Finally we will consider the multiplication operation, required by the graph models as we will see later. Let S_{zh} represent the set of hidden component variables and $S_z \setminus S_{zh}$ the component variables we can observe in data. Multiplying two mixture models $f^1(S_y, S_z)$ and $f^2(S_y, S_z)$ results in

$$\begin{aligned} f^1(S_y, S_z) \cdot f^2(S_y, S_z) \quad = \quad & \sum_{\boldsymbol{i}\in S_z\setminus S_{zh}} \frac{\pi_{\boldsymbol{i}}^1 \pi_{\boldsymbol{i}}^2}{\sum_{\boldsymbol{k}\in S_z} \pi_{\boldsymbol{k}}^1 \pi_{\boldsymbol{k}}^2} f_{\boldsymbol{i}}^1(S_y|\theta_{\boldsymbol{i}}^1) f_{\boldsymbol{i}}^2(S_y|\theta_{\boldsymbol{i}}^2) + \\ & \sum_{\boldsymbol{j}\in S_{zh}} \sum_{\boldsymbol{j}'\in S_{zh}} \frac{\pi_{\boldsymbol{j}}^1 \pi_{\boldsymbol{j}'}^2}{\sum_{\boldsymbol{k}\in S_z} \pi_{\boldsymbol{k}}^1 \pi_{\boldsymbol{k}}^2} f_{\boldsymbol{j}}^1(S_y|\theta_{\boldsymbol{j}}^1) f_{\boldsymbol{j}'}^2(S_y|\theta_{\boldsymbol{j}'}^2) \end{aligned} \tag{3.33}$$

which is a new mixture model. Studying this expression, we can see that multiplying mixtures can potentially cause some complexity problems. If the mixtures multiplied have h_1 and h_2 number of outcomes of the hidden variables in total, then the new mixture will have $h_1 \cdot h_2$ number of outcomes in total for the hidden variables. This might be cause for concern, since the dimensionality and complexity of the mixtures may get out of hand in more complex models.

Also, it is worth noting that a mixture model can in fact also easily be multiplied with a distribution that is not a mixture itself simply by multiplying this distribution with all the component distributions.

Graph Models

Let us now come back to graphical models. Graph models are used to compactly represent a joint probability distribution, using a graph structure to represent dependencies between variables. As before, a node in the graph usually represents a random variable. The arcs represent dependencies, or perhaps clearer, the lack of arcs represent conditional independence assumptions.

We have earlier made a distinction between *directed* and *undirected* graph models. In directed graph models, an arc implies a causal relationship. For example, an arc from a to b could be said to represent that "a causes b". Here we will focus on undirected hypergraphs [Geman and Geman, 1984; Buntine, 1996]. This is not as much of a restriction as it first might seem, since directed models with some limitations can be re-written as undirected graph models. From our highly data driven viewpoint, it may be even less of a restriction, since it is not necessarily very well defined what is meant by causal connections between random variables. All we can measure in a simple manner is correlation.

The undirected graph also describes how the included variables depend on each other: each node in the graph represents one variable, and each edge represents a dependency between two attributes. In general a *hyper graph* is required, which can contain edges that can each connect three or more nodes, thus representing higher order dependencies between the corresponding attributes.

The Graph Model and its Estimation

Let $\boldsymbol{x}$ be a multivariate random variable over the set of variables $S_x = \{x_1, x_2, \ldots, x_n\}$. Then the probability density function for the graph model over the random variable $\boldsymbol{x}$ can be written as

$$f(\boldsymbol{x}) = \prod_i \boldsymbol{\Psi}(u_i) \tag{3.34}$$

where $\boldsymbol{\Psi}(u_i)$ are *factors*, or joint probability distributions, over the variables $U_i \subset S_x$. Each variable U_i represents a set of variables that are dependent by their joint distribution, and does not keep track of any causal direction.

In this description, we will allow hyper-arcs that have a common set of nodes in the graph, provided that there is a separate arc connecting exactly these common nodes. The complete definition of the factors $\boldsymbol{\Psi}$ is then

$$\boldsymbol{\Psi(u_i)} = \frac{P(u_i)}{\prod_{U_j \subset U_i} \boldsymbol{\Psi}(u_j)} \tag{3.35}$$

where the product is taken over all "sub-arcs" to the hyper-arc U_i, and a sub-arc is required for every common set of nodes between two hyper-arcs. Note that here the primary attributes x_i are included among the U_i, although they are not strictly considered as "arcs" in the graph. This means that i ranges from 1 to $n + m$ here, where m is the number of complex factors incorporating more than one variable.

One interpretation of equation 3.35 is that for each direct dependency between a set of variables there is a factor $\mathbf{\Psi}(U_k)$ which will adjust the probability by multiplying with the joint distribution and dividing by the "previous" product expansion for that distribution.

If the graph structure is known, *i. e.* we have completely specified the sets U_i in the graph, estimation is simply a matter of estimating every sub-distribution $\mathbf{\Psi}$ represented in the graph. For some operations though, all marginals over all sub-distributions in the graph must be the same. If this cannot be guaranteed, we may get inconsistent results when calculating conditionals and marginals of the graph model. The problem typically arises if one variable is used in several mixture representations, or if one variable is part of several factors and Bayesian parameter estimates are used (see section 3.9).

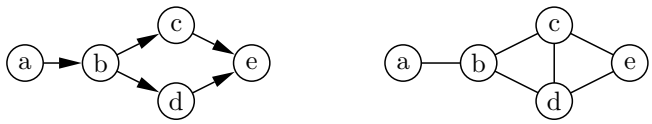

Figure 3.5: A directed dependency graph and its corresponding non-directed hypergraph.

As an example of how to write the probability distribution in a graph, consider figure 3.5. Starting with the directed graph, we sort the attributes into the chain rule for probabilities but keep only the direct dependencies according to the graph:

$$P(\boldsymbol{x}) = P(a)P(b|a)P(c|b)P(d|b)P(e|cd)$$

Then, write the conditional probabilities as fractions:

$$P(\boldsymbol{x}) = P(a)\frac{P(ab)}{P(a)}\frac{P(bc)}{P(b)}\frac{P(bd)}{P(b)}\frac{P(cde)}{P(cd)}$$

By noting the independencies between branches we can rewrite the expression to

$$\begin{aligned} P(\boldsymbol{x}) \quad &= \quad P(a)P(b)P(c)P(d)P(e)\left(\frac{P(ab)}{P(a)P(b)}\right)\left(\frac{P(bc)}{P(b)P(c)}\right)\left(\frac{P(bd)}{P(b)P(d)}\right)\cdot \\ &\cdot \quad \left(\frac{P(ce)}{P(c)P(e)}\right)\left(\frac{P(de)}{P(d)P(e)}\right)\left(\frac{P(cde)P(c)P(d)P(e)}{P(cd)P(ce)P(de)}\right) \end{aligned}$$

The result is now on the form of equation 3.34 and 3.35. It contains all first order factors as well as additional factors to compensate for the dependencies among the attributes.

Marginals, Conditionals and Multiplication of Graphs

The result of calculating the marginal and the conditional of a graph model is generally a new graph model over the attributes in interest. Similar to the description of the mixture model, let S_x^A and S_x^B be disjoint complementary subsets of S_x, $S_x^A \subset S_x$, $S_x^B \subset S_x$, $S_x^A \cap S_x^B = \varnothing$, and $S_x = S_x^A \cup S_x^B$. Then the marginal over the variables S_x^A and S_x^B of a graph model can be written as

$$f(S_x^A) = \prod_{i:U_i \subset S_x^A} \mathbf{\Psi}(u_i) \tag{3.36}$$

This should be interpreted as the product of all factors U_i that are composed of a subset of the variables in S_x^A. The expression can be seen as a new graph model over the attributes S_x^A. Formally, we may have to add arcs to the graph model if the resulting marginal consists of several sub-graphs that are not connected.

The conditional of a graph model given the values of the variables in S_x^B is given by

$$f(S_x^A|S_x^B) = \prod_i \mathbf{\Psi}(u_i|S_x^B) \tag{3.37}$$

The interpretation of this rather simplified notation is that all probabilities in the expression for $\mathbf{\Psi}$ are conditioned on S_x^B. If all variables in $\mathbf{\Psi}$ are known, *i. e.* they are all in S_x^B, then $\mathbf{\Psi}$ becomes constant and disappears from the expression. If no variables in $\mathbf{\Psi}$ are known from S_x^B then $\mathbf{\Psi}$ is left unchanged. We can interpret equation 3.37 as a new graph model on the form of equation 3.34.

Multiplying two graph models $g^1(S_x)$ and $g^2(S_x)$ can be expressed as

$$g^1(S_x) \cdot g^2(S_x) = \prod_i \mathbf{\Psi}^1(u_i) \cdot \mathbf{\Psi}^2(u_i) \tag{3.38}$$

The result is a new graph, where all probabilities in the expressions for $\mathbf{\Psi}^1$ are multiplied with the corresponding probability in the expression for $\mathbf{\Psi}^2$, where $\mathbf{\Psi}^1$ and $\mathbf{\Psi}^2$ are the factors in graph $g^1(S_x)$ and $g^2(S_x)$ respectively. The graph structures must be the same for both graphs for this expression to be valid, and it relies on the assumption that all corresponding sub-distributions can be multiplied themselves.

Modelling Considerations and Exact Calculation of Expressions

The graph modelling technique described is very similar to that used in *Bayesian belief networks* [Pearl, 1988], but the way it is used here is slightly different. In Bayesian belief networks the output, *i. e.* the variable to be predicted, is part of the graph. However, in many modelling situations, especially when the graph structure is automatically generated from data, the output attributes $\boldsymbol{w}$ can be kept "outside" of the graph, in the sense that all probabilities are conditional on the output variable $\boldsymbol{w}$. In our experience, it is often computationally advantageous to keep the output variables outside the graph in this manner, since there is then

no need to propagate distributions through the graph. It is also likely to be more robust, since distributions are calculated in "parallel" rather than in "series", and thus noise will cancel out rather than accumulate.

Also, note that we have here only been concerned with exact calculation of the expressions for marginals and conditionals. For very complex models and Bayesian approaches approximate calculation might become appealing or necessary, *e. g.* by sampling (Monte Carlo and Markov Chain Monte Carlo) methods [W. R. Gilks and Spiegelhalter, 1995], loopy belief propagation [Weiss, 2000] or variational methods [Jordan, 1999]. However, these methods will not be further treated in this text.

Combining Graph Models and Finite Mixtures

As we have seen, both graph models and general mixture models can be formulated in such a way that the most common operations are independent of the sub-distributions. This holds if both the graph model, the mixture model and all other distributions used provide a few standard operations. These operations are estimation, calculation of marginals and conditionals, calculation of the likelihood of a sample and multiplication of distributions. The need to calculate the likelihood of a sample arises is the estimation procedure used for mixture models, and the calculation of conditionals in graph models require us to be able to multiply distributions.

Let us now introduce some notation to easily describe these models and operations. We will refer to a general Hierarchical Graph Mixture, *i. e.* either a graph, a mixture or a simple distribution, over the variables $\{x_1, x_2, \ldots, x_n\}$ as $H(x_1, x_2, \ldots, x_n)$. A specific mixture model with component densities $H_1, H_2, \ldots, H_m$ will be written as $M(H_1, H_2, \ldots, H_m; S_z, S_z^h)$, where S_z is the set of observed discrete variables and S_z^h are the hidden discrete variables. Similarly, we will describe a graph model by all its factors, $G(H_1, H_2, \ldots, H_k)$ (see equation 3.34). Note that one factor may be the marginal of another. We do not need to state the resulting multiplicity for each of these factors, since this is implied by the complete set of factors.

The marginal over the attributes $x_1, \ldots, x_n$ of a model H will be expressed as $\mathrm{marg}(H; \{x_1, \ldots, x_n\})$, and the conditional of H given the values of these variables as $(H \mid \{x_1, \ldots, x_n\})$. Multiplying two models H_1 and H_2 will simply be expressed as $H_1 \cdot H_2$, and the probability of the sample $\boldsymbol{x}$ being drawn from model H as $\P(\boldsymbol{x}|H)$. Now we can write the expressions for marginals and conditionals of a mixture model M and graph model G in simplified form as

$$\begin{aligned}\mathrm{marg}(M(H_1, \ldots, H_n); \boldsymbol{x}) = M''(M_1'(\mathrm{marg}(H_1; \boldsymbol{x}), \ldots, \mathrm{marg}(H_n; \boldsymbol{x})), \\ \ldots, \\ M_k'(\mathrm{marg}(H_1; \boldsymbol{x}), \ldots, \mathrm{marg}(H_n; \boldsymbol{x})))\end{aligned} \tag{3.39}$$

$$(M(H_1, \ldots, H_n) \mid \boldsymbol{x}) = M'((H_1 \mid \boldsymbol{x}), \ldots, (H_n \mid \boldsymbol{x})) \tag{3.40}$$

$$\mathrm{marg}(G(H_1, \ldots, H_n); \boldsymbol{x}) = G'(\mathrm{marg}(H_1; \boldsymbol{x}), \ldots, \mathrm{marg}(H_n; \boldsymbol{x})) \tag{3.41}$$

$$(G(H_1, \ldots, H_n) \mid \boldsymbol{x}) = G'((H_1 \mid \boldsymbol{x}), \ldots, (H_n \mid \boldsymbol{x})) \tag{3.42}$$

These expressions correspond directly to the ones described earlier in this paper. The marginal of a mixture M is a new mixture M'' of k mixtures, where k is the total number of outcomes of the discrete variables in M'. The conditional of a mixture is a new mixture, and both the marginal and conditional of a graph model is a new graph.

3.7 Leaf Distributions

All hierarchies of graph models and mixture models will eventually terminate in leaf distributions, *i. e.* densities that are not expressed in terms of sub-distributions. Examples of these distributions are the discrete distribution and Gaussian distributions. In fact, we could model many situations to arbitrary precision using only these two leaf distributions with the help of mixture models. Unfortunately, this might require stable estimation of a very large number of mixture components, something that we most often do not have enough data to perform. Still, we will only mention discrete and Gaussian distributions here.

All leaf distributions must provide the same necessary operations as the mixture and graph model. For completeness, we will list how to perform these operations on the discrete and Gaussian distributions here.

The Discrete Distribution

Calculating marginals and distributions for a multivariate discrete distribution is very straightforward. For the following calculations, let us assume that the n-dimensional distribution is parameterised by the probabilities $P(\boldsymbol{x} = \boldsymbol{i}) = P(x_1 = i_1, x_2 = i_2, \ldots, x_n = i_n) = p_{\boldsymbol{i}}$, arranged in a hypercube of the same dimensionality as $\boldsymbol{x}$.

The Marginal Distribution

Calculating the marginal distribution simply amounts to summing over all variables that we are not interested in. If we let $P(\boldsymbol{x}_a, \boldsymbol{x}_b)$ be the joint distribution of the sets of variables $\boldsymbol{x}_a$ and $\boldsymbol{x}_b$, the marginal over $\boldsymbol{x}_a$ is simply calculated by summing over all variables in $\boldsymbol{x}_b$ as

$$P(\boldsymbol{x}_a = \boldsymbol{i}_a) = \sum_{\boldsymbol{i}_b \in \boldsymbol{x}_b} P(\boldsymbol{x}_a = \boldsymbol{i}_a, \boldsymbol{x}_b = \boldsymbol{i}_b) \tag{3.43}$$

That is, we calculate each parameter of the resulting distribution by summing over all probabilities which correspond to the certain outcome the parameter is representing.

The Conditional Distribution

If, as above, $P(\boldsymbol{x}_a, \boldsymbol{x}_b)$ is the joint distribution of $\boldsymbol{x}_a$ and $\boldsymbol{x}_b$, the parameters of the conditional distribution $P(\boldsymbol{x}_a|\boldsymbol{x}_b)$ are

$$P(\boldsymbol{x}_a = \boldsymbol{i}_a|\boldsymbol{x}_b = \boldsymbol{i}_b) = \frac{P(\boldsymbol{x}_a = \boldsymbol{i}_a, \boldsymbol{x}_b = \boldsymbol{i}_b)}{P(\boldsymbol{x}_a = \boldsymbol{i}_a)} \tag{3.44}$$

If the outcome $\boldsymbol{i}_b$ is known, the resulting distribution over $\boldsymbol{x}_a$ then is

$$P(\boldsymbol{x}_a = \boldsymbol{i}_a)_{|\boldsymbol{i}_b} = \frac{P(\boldsymbol{x}_a = \boldsymbol{i}_a, \boldsymbol{x}_b = \boldsymbol{i}_b)}{\sum_{\boldsymbol{i}_a \in \boldsymbol{x}_a} P(\boldsymbol{x}_a = \boldsymbol{i}_a, \boldsymbol{x}_b = \boldsymbol{i}_b)} \tag{3.45}$$

This corresponds to finding the subcube of $P(\boldsymbol{x}_a, \boldsymbol{x}_b)$ for which $\boldsymbol{x}_b = \boldsymbol{i}_b$ and normalising the result.

Multiplying Discrete Distributions

If we let $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ represent two different distributions over the variables $\boldsymbol{x}$, the product of the distributions $P(\boldsymbol{x})^*$ can be written as

$$P(\boldsymbol{x} = \boldsymbol{i})^* = \frac{P(\boldsymbol{x}_1 = \boldsymbol{i})P(\boldsymbol{x}_2 = \boldsymbol{i})}{\sum_{\boldsymbol{i} \in \boldsymbol{x}} P(\boldsymbol{x}_1 = \boldsymbol{i})P(\boldsymbol{x}_2 = \boldsymbol{i})} \tag{3.46}$$

In practise, we just need multiply the distributions parameter wise, and calculate a normalising constant for the resulting distribution.

The Gaussian Distribution

The Gaussian distribution is a continuous function that, due to its many convenient properties, is very commonly used description of a distribution. Statisticians and mathematicians usually refer to it as the multivariate *normal* distribution, while the name Gaussian is perhaps more used by physicists (although this will not stop us from consequently referring to it as a Gaussian distribution throughout this text). Gaussians are often used to describe random variates with unknown distributions, which actually is not as dangerous as it may sound initially. The central limit theorem states that the mean of any set of variates with any distribution having a finite mean and variance tends to the normal distribution. This means that many common attributes roughly follow Gaussian distributions. Still, one should proceed with caution when invoking Gaussian distributions, as there of course are many situations where they simply are not applicable.

The multivariate Gaussian is characterised by its mean value vector $\boldsymbol{\mu}$ and covariance matrix Σ, and its probability density function is given by

$$f(\boldsymbol{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\boldsymbol{x} - \boldsymbol{\mu})\right) \tag{3.47}$$

The distribution function is sometimes written as $\mathcal{N}(\boldsymbol{\mu}, \Sigma)$ for short (where $\mathcal{N}$ referers to the normal distribution). To make notation easier, let us partition the state space into two parts, $\boldsymbol{x}_a \in \Re^{k\times 1}$ and $\boldsymbol{x}_b \in \Re^{l\times 1}$, where

$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{x}_a \\ \boldsymbol{x}_b \end{pmatrix}, \qquad \boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix}, \qquad \Sigma = \begin{pmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{pmatrix} \tag{3.48}$$

Now we can write the probability density function as

$$f(\boldsymbol{x}; \boldsymbol{\mu}, \Sigma) = \frac{1}{\sqrt{(2\pi)^d \left|\Sigma\right|}} \exp\left(-\frac{1}{2} \begin{pmatrix} \boldsymbol{x}_a - \boldsymbol{\mu}_a \\ \boldsymbol{x}_b - \boldsymbol{\mu}_b \end{pmatrix}^T \begin{pmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{pmatrix}^{-1} \begin{pmatrix} \boldsymbol{x}_a - \boldsymbol{\mu}_a \\ \boldsymbol{x}_b - \boldsymbol{\mu}_b \end{pmatrix} \right) \tag{3.49}$$

a useful form when we want to express the marginal and conditional of a multivariate Gaussian.

The Marginal Distribution

To calculate the marginal distribution $f(\boldsymbol{x}_a)$ from the joint distribution $f(\boldsymbol{x}_a, \boldsymbol{x}_b)$ we can use that

$$\begin{aligned} f(\boldsymbol{x}_a, \boldsymbol{x}_b) &\sim \mathcal{N}(\boldsymbol{\mu}, \Sigma) \\ &\sim \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu}_a \\ \boldsymbol{\mu}_b \end{pmatrix}, \begin{pmatrix} \Sigma_{aa} & \Sigma_{ab} \\ \Sigma_{ba} & \Sigma_{bb} \end{pmatrix} \right) \end{aligned}$$

and thus

$$f(\boldsymbol{x}_a) = \frac{1}{\sqrt{(2\pi)^k \left|\Sigma_{aa}\right|}} \exp\left(-\frac{1}{2} (\boldsymbol{x}_a - \boldsymbol{\mu}_a)^T \Sigma_{aa}^{-1} (\boldsymbol{x}_a - \boldsymbol{\mu}_a) \right) \tag{3.50}$$

The Conditional Distribution

The conditional distribution, a function of both $\boldsymbol{x}_s$ and $\boldsymbol{x}_b$, is again a Gaussian distribution, and can be written as

$$f(\boldsymbol{x}_a | \boldsymbol{x}_b) = C_{a|b} \exp\left(-\frac{1}{2} (\boldsymbol{x}_a - \boldsymbol{\mu}_{a|b}) \Sigma_{|\Sigma_{bb}} (\boldsymbol{x}_a - \boldsymbol{\mu}_{a|b}) \right) \tag{3.51}$$

where the adjusted mean $\boldsymbol{\mu}_{a|b}$ is expressed as

$$\boldsymbol{\mu}_{a|b} = \boldsymbol{\mu}_a + \Sigma_{ab} \Sigma_{bb}^{-1} (\boldsymbol{x}_b - \boldsymbol{\mu}_b) \tag{3.52}$$

The term $\Sigma_{|\Sigma_{bb}}$, the covariance matrix of the resulting distribution, is the Schur decomposition of Σ with respect to Σ_{bb},

$$\Sigma_{|\Sigma_{bb}} = \Sigma_{aa} - \Sigma_{ab} \Sigma_{bb}^{-1} \Sigma_{ba} \tag{3.53}$$

and the normalisation constant $C_{a|b}$ is given by

$$C_{a|b} = \frac{1}{\sqrt{(2\pi)^k \left|\Sigma_{|\Sigma_{bb}}\right|}} \tag{3.54}$$

where k is the dimensionality of the resulting distribution.

Multiplying Gaussian Distributions

When Gaussians multiply, precisions add. Multiplying n Gaussians with means $\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_n$ and covariance matrices $\Sigma_1, \ldots, \Sigma_n$ results in a Gaussian distribution with

$$\boldsymbol{\mu} = \left(\sum_{i=1}^{n} \Sigma_i^{-1} \right)^{-1} \left(\sum_{j=1}^{n} \Sigma_i^{-1} \boldsymbol{\mu}_j \right) \tag{3.55}$$

$$\Sigma = \left(\sum_{i=1}^{n} \Sigma_i^{-1} \right)^{-1} \tag{3.56}$$

where $\boldsymbol{\mu}$ and Σ are the parameters of the resulting distribution and Σ_i^{-1} the inverse of the covariance matrix.

3.8 Examples of Models

Here we will present a few examples of common statistical models formulated as Hierarchical Graph Mixtures and discuss a few issues related to the construction of HGMs, as well as examples of the practical use of the framework. Some of the examples are very simple, but serve to provide a basic understanding to how the models are formulated.

Naive Bayes

Assume that we want to predict the distribution of a variable y, given the values of a set of variables $\boldsymbol{x} = \{x_1, x_2, \ldots, x_n\}$. One straightforward way of modelling this situation is to assume that y depends on all variables in $\boldsymbol{x}$, but that all variables in $\boldsymbol{x}$ are conditionally independent given y as shown in figure 3.6. This is usually referred to as a *naive Bayesian* model, and the complete distribution of y given $\boldsymbol{x}$ can be expressed as a product of the distributions of the individual variables, or $P(y|\boldsymbol{x}) \propto P(y)P(\boldsymbol{x}|y) = P(y) \prod_{i=1}^{n} P(x_i|y)$.

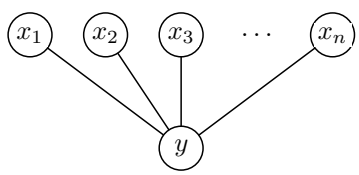

Figure 3.6: The naive Bayes model.

There are two ways of expressing the naive Bayes model with HGMs. The first is by using a graph model which we can write as

$G(H(x_1, y), H(x_2, y), \ldots, H(x_n, y), H(y))$. We can leave out the factors representing $H(x_i)$ in the graph since we do not intend to calculate the marginal or conditional for these variables. The probability density function can be written as $P(\boldsymbol{x}, y) = \prod_{i=1}^{n} P(x_i, y)/P(y)^{n-1}$.

The second way of expressing the naive Bayes model is by a mixture, $M(H_1(\boldsymbol{x}), \ldots, H_k(\boldsymbol{x}); \{y\}, \{\varnothing\})$. All component densities in the mixture are product models, expressed as graphs without dependencies between the attributes, $H_i = G(H(x_1), \ldots, H(x_n))$. In this formulation y needs to be a discrete variable with a finite number of outcomes since we have limited ourselves to this kind of mixture here. The number of outcomes in y is assumed to be k, and therefore we have k component densities in the mixture.

The fact that we can formulate the naive Bayes model both as a graph and a mixture points to the overlap between these two formulations. Returning to figure 3.3, it is apparent that by using one or several variables to label samples we can actually use the factors in a graph model to group samples, much like a mixture model. The opposite is of course also true, that by using labelled samples a mixture model can group attributes in a way that is similar to a graph model. This does not, however, mean that it is straightforward to implement a mixture of graphical models with different graphical structures directly within a graph. Including an attribute that labels samples according to cluster to replicate the functionality of a mixture model within a graph is possible but not in general desirable. If we do this, we will have to represent every dependency from every cluster within the graph, leading to very complex graphical structures.

Quadratic Classifiers

A quadratic classifier finds a quadratic discrimination function between classes, such as a parabola, a circle, or an ellipse in feature space. An example is the Gaussian density based quadratic classifier, which estimates one Gaussian distribution for each individual class. This can be expressed as a HGM as $M(H_1(\boldsymbol{x}), \ldots, H_k(\boldsymbol{x}); \{y\}, \{\varnothing\})$, where $\boldsymbol{x} = \{x_1, \ldots, x_n\}$ are the input variables, y is the class label with k different classes, and $H_1, \ldots, H_k$ are Gaussian distributions over all input variables $\boldsymbol{x}$.

Comparing this with the mixture model formulation of the naive Bayes classifier above, we can see that the expressions are very similar. The difference between the two models is the independence assumption made in the naive Bayes model, where all input attributes are assumed to be conditionally independent given the output. This quadratic classifier is also very similar to a linear classifier based on Gaussian densities. The difference is that the quadratic classifier calculates covariance matrices for each individual class, instead of assuming equal covariance matrices for all classes as the linear classifier.

Markov Models and Markov Random Fields

A Markov model is often used to describe sequential data, in which we can make the assumption that an attributes value at one specific position depends directly only on the values of the attribute in the n earlier positions. For example, if n is 1, the value of the attribute only depends on its previous value and we refer to its description as a first order Markov model. The probability for a sequence of length N can then be written as $P(\boldsymbol{x}) = P(x_N \mid x_{N-1})P(x_{N-1} \mid x_{N-2}) \dots P(x_2 \mid x_1)P(x_1)$. A graph representation of the model is shown on the left in figure 3.7.

A Markov Random Field can be seen as a generalisation to two or more dimensions of the Markov model, useful *e. g.* for modelling contextual or spatial dependencies. The assumptions are very similar to the one dimensional case, although we might now have to consider several versions of the neighbourhood system, *i. e.* how the sites are related to one another. A Markov Random Field with a simple first order neighbourhood is shown on the right in figure 3.7.

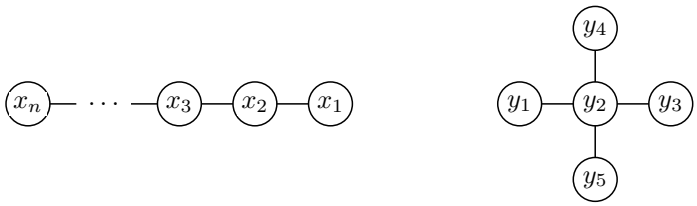

Figure 3.7: A Markov model of order n and a Markov random field with a first-order neighbourhood system.

The general Markov model can easily be described as a HGM, similar to $G(H(x_1, \dots, x_n), \dots, H(x_1, x_2), \dots, H(x_1))$, or easier in the case of a first order Markov model as $G(H(x_1, x_2), H(x_1), H(x_2))$. The Markov Random Field shown in figure 3.7 is similarly expressed as $G(H(y_2, y_1), \dots, H(y_2, y_5), H(y_2))$. Because of the close relationship between Markov Random Fields and the undirected graph models we use here, the Markov models are directly expressed as graphs. The component distributions of these graphs can be both mixtures and other graphs, allowing us to express more complicated variants of these simple Markov models, as well as constructing graphs or mixture models of the Markov models themselves. The latter approach is related, but not equivalent to Hidden Markov Models.

Probabilistic Dependency Trees

An effective way of approximating complex multivariate distributions is by using dependency trees, *i. e.* directed graph models where only second order distributions are used [Chow and Liu, 1968]. An optimal dependency tree representation, in

the sense that it maximises the likelihood function, can be easily generated using a greedy algorithm. The algorithm calculates the pairwise mutual information between all variables, and sorts the results in descending order. It then creates a tree by adding a branch between the attributes in the sorted mutual information list as long as this does not introduce a cycle, beginning with the strongest dependency in the list and continuing until all variables are included in the tree.

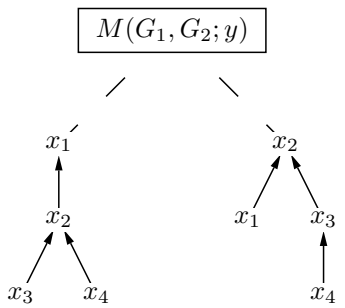

Figure 3.8: A simple example of a mixture of dependency trees. The mixture M contains two graph models G_1 and G_2. These graph models corresponds to the two outcomes of the discrete variable y, which can be either hidden or observed from data.

The resulting tree can easily be re-written to the undirected graph models described here, which means that we can create mixtures and graphs over such dependency trees. The fact that we can now create mixture models over dependency trees means that we can represent a complex distribution with different trees in different parts of feature space, or with different trees depending on the outcome of a discrete variable observable in data. These trees can be automatically generated using the algorithm described above, both if the discrete variable is hidden or not, and allows us to express a complex distribution in a very efficient way. Simply put, we can create mixtures with completely different graph structures for each outcome of an attribute, something that is difficult in *e. g.* Bayesian Belief Networks. The same structure also allow us to represent many of the models mentioned in section 3.2, such as [Friedman *et al.*, 1997] and [Meila and Jordan, 2000].

Example of a Complex Model Structure

Above, mixtures of graphs have been exemplified. As an example of a more complex model structure, let us consider a data fusion problem involving classification

based on three different modes of data; *e. g.* image, sound, and sensor measurements, which involves graphs of both mixtures and further graphs. We consider the measurements from these different modes to be independent given the class, a simplification that should in many cases hold relatively well in practise.

Thus, a suitable form of the model structure is, as shown in figure 3.9, a supervised mixture of graphical models, with one graphical model for each class. These graphs are in turn simple products of mixture models, with one mixture model for each mode of data. The mixtures consist of a number of graphical models, *e. g.* trees as generated as above, to compensate for clusters and dependencies within each mode of data. These graphs in turn would then consist of *e. g.* discrete distributions, Gaussian distributions, supervised mixtures for factors containing both continuous and discrete attributes, and possible mixtures of Gaussians to represent more complicated continuous marginals.

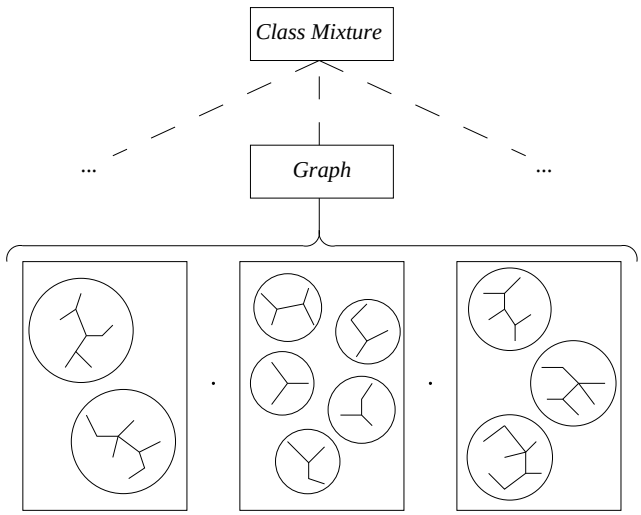

Figure 3.9: A classification model for multi-modal data. A supervised mixture consisting of one graphical model for each class is used for classification. These graphical models are simple products of models for each mode, which are drawn as three rectangles in the figure. Each of these graphical models is then specified as a mixture of graphical models, illustrated in the figure with two, five, and three circles respectively.

Let us denote the class y, and the variables within the three modes image, sound, and other measurements $\boldsymbol{x}^i$, $\boldsymbol{x}^s$, and $\boldsymbol{x}^m$ respectively. We can then write

the top levels of the model using our shorthand notation as

$$\begin{aligned} M(G(&M(G_1(\boldsymbol{x}_1^i),\ldots;\{\varnothing\},\{z^i\}),\\ &M(G_1(\boldsymbol{x}_1^s),\ldots;\{\varnothing\},\{z^s\}),\\ &M(G_1(\boldsymbol{x}_1^m),\ldots;\{\varnothing\},\{z^m\})),\ldots;\{y\},\{\varnothing\}) \end{aligned}$$

Using this model structure, we arrive at a model with a much lower complexity compared to using a single graphical model over all attributes in all modes of data. In the case of figure 3.9, the number of parameters would be in the order of $2+5+3$, compared to a single graphical models $2 \cdot 5 \cdot 3$.

A Practical Example

A practical example of where these hierarchical graph mixture models have been used is for the hot steel mill Outokumpu Stainless AB in Avesta. The task was to identify which steel coils are at risk of getting surface damages (or "slivers"). There were about 270 attributes to consider, some continuous valued and some discrete.

Furthermore, it turned out that different steel types were significantly differently sensitive for slivers. To effectively model the data, we had to use multiple layers in the hierarchical graph model: we built a mixture with one model for non-sliver cases and one model for sliver cases; within each of them we built a mixture model over each of eight different steel types; within each of these we modelled the data over the 270 attributes with a graph model; and finally, in the graph model we sometimes had to make a joint model over one continuous and one discrete attribute, which was again realized as a mixture (see figure 3.10). If we refer to the input attributes as $\boldsymbol{x}$, the steel type as y_t, and whether or not a sample has slivers as y_s, in shorthand notation the top levels of the model become

$$\begin{aligned} M(&M(G_1(\boldsymbol{x}),\ldots,G_8(\boldsymbol{x});\{y_t\},\{\varnothing\}),\\ &M(G_1(\boldsymbol{x}),\ldots,G_8(\boldsymbol{x});\{y_t\},\{\varnothing\});\{y_s\},\{\varnothing\}) \end{aligned}$$

In effect we had a mixture model of mixture models of graphs of mixture models. Each graph was built as a dependency tree, as described in section 3.8. This seemingly complicated model manages to 10-fold the accuracy when picking out which steel coils were at risk of getting slivers.

3.9 Encoding Prior Knowledge and Robust Parameter Estimation

When fitting complex models with many free parameters to limited amounts of data, the result is usually a model that performs very well on the data on which it was estimated from, but performs poorly when applied to previously unseen patterns. This phenomenon is known as *over-fitting*: adapting too much to the training data set on the expense of generalisation performance. To counter this within our

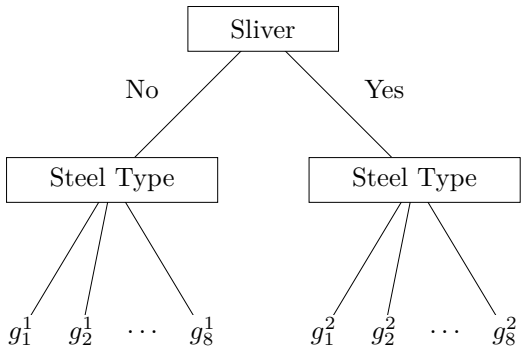

Figure 3.10: The model used for sliver risk identification. The rectangular boxes represent mixture models, and g_j^i individual graph models with different dependency structures. Note that some of the attributes in these graph models are themselves represented as mixtures.

statistical models, we can sometimes use simpler models that do not suffer as much from over-fitting. However, these may lack some of the expressive capabilities of more flexible models, compromising performance and practical usefulness. A more suitable approach might be to try to account for the uncertainty implied by the small data sets, and perhaps incorporate previous knowledge or hypothesis in the analysis. Here, we will use *Bayesian statistics* do just that.

We will also use these Bayesian estimates to introduce another type of hierarchy into the framework, not relating to how the problem is parameterised but rather how the parameters are estimated. We show that parameters can be estimated hierarchically, starting with a prior belief in what the distribution looks like and then modifying this belief based on observed data. The resulting distribution can then again be used to represent a prior belief which in turn is modified on another set of data, and so on. But before we go into the details, let us start with an introduction to Bayesian statistics.

A Brief Introduction to Bayesian Statistics

Mathematical statistics use two major paradigms. The first is the conventional, or *frequentist* paradigm. This considers probabilities as property of nature, which can be measured by repeating an experiment sufficiently many times for the required accuracy.

The second paradigm, *Bayesian statistics*, is based on an interpretation of prob-

abilities as rational, conditional measures of uncertainty, given the accepted assumptions and available information. This closely resembles the interpretation of the word "probability" in ordinary language, but differs from the frequentist view. In Bayesian statistics, a probability is always a function of at least two arguments, the event E whose uncertainty is being measured, and the conditions C under which the event is observed, which can be written as $P(E|C)$. In a typical application, we are interested in the probability of an event E given known data D, assumptions A, and other relevant knowledge K, as $P(E|D, A, K)$.

The main consequence of treating probability as a general measure of uncertainty is that we need to describe all uncertain parameters in a problem as probability distributions. All unknown parameters in probability models must have a probability distribution that describes what we know about their values. Parameters are thus treated as random variables, not to describe their variability but to express the uncertainty about their true values.

Now, let us see what this means in practise if we want to estimate the parameters of a probability distribution. Let us first start with a brief look at how this is performed classically through the *Maximum Likelihood* method. In this, we would like to find the values of the parameters M that maximise the probability of the data D, *i. e.* $P(D|M)$, where our "event" is the model M and we only rely mainly on known data D. In contrast, the Bayesian approach to parameter estimation aims not to maximise the probability of the data that we already have, but rather the probability of the parameters given the data, $P(M|D)$.

The relation between the two approaches becomes clearer if we note that

$$P(M|D) \propto P(D|M)P(M) \tag{3.57}$$

The distribution $P(M)$ in this expression is usually referred to as the *prior probability* of the model, or what we know or assume about the world before we have seen the data D, and $P(M|D)$ is the *posterior probability*, or what we believe after we have observed the data. The posterior distribution can then be used as a prior for a new data set generated by the same underlying process. As we can see, the relation between the classical way of parameter estimation and the Bayesian approach is essentially the prior distribution $P(M)$, expressing what we know or believe about the parameters before observing the data.

An Example of Bayesian Parameter Estimation

Consider estimating the probability of heads of a possibly unbalanced coin by tossing it a number of times and keeping track of the results. If we call our parameter, the probability of heads p, then the probability of getting c_H heads out of $c = c_H + c_T$ tosses is

$$P(D|M) = P(c_H|p) = \binom{c}{c_H} p^{c_H}(1-p)^{c_T} \tag{3.58}$$

If we want to find the maximum likelihood estimate, we differentiate this expression with respect to p to find its maximum,

$$\begin{aligned}\frac{d}{dp}P(c_H|p) &= \binom{c}{c_H}c_H p^{c_H-1}(1-p)^{c_T} - \binom{c}{c_H}c_T p^{c_H}(1-p)^{c_T-1} \\ &= (c_H(1-p) - c_T p)\binom{c}{c_H}p^{c_H-1}(1-p)^{c_T-1} \qquad (3.59)\end{aligned}$$

Equating with zero and ignoring the uninteresting solutions $p = 0$ and $1 - p = 0$ gives us

$$c_H(1-p) = c_T p \Rightarrow \hat{p} = \frac{c_H}{c_H + c_T} = \frac{c_H}{c} \qquad (3.60)$$

This is the maximum likelihood estimate of the parameter p of a Bernoulli distributed variable. This works quite well when we have lots of data, in this particular case meaning large numbers of recorded results of tossing the particular coin. However, for small sample sizes, the estimate might produce strange results. For example, if we make only two tosses of a balanced coin that both turn out to be tails, a not particularly unlikely result, our estimate of the probability of heads would be 0. As we in practise often want to be able to perform further statistical calculations based on our estimated parameters, and also be able to base them on sometimes very scarce data, this type of parameter estimation is simply not suitable.

Let us now consider a Bayesian approach. Consulting equation 3.57 indicates that we first need to specify a prior distribution $P(M)$ over p. If we assume that we do not know anything about the distribution over p beforehand, and that all values are equally likely, the prior distribution can simply be reduced to $P(M) = P(p) = 1$. The posterior distribution then becomes

$$P(M|D) = P(p|c_H, c_T) \propto \binom{c}{c_H}p^{c_H}(1-p)^{c_T} \qquad (3.61)$$

Although this looks like a binomial distribution, this is not quite the case. It should rather be interpreted as the distribution for the *parameter* p. As the expression is a proportion rather than an equality, it also needs to be normalised over all possible values of p to find the actual distribution. Integrating the right hand side, leaving out the actual calculations, gives us that

$$\int_0^1 p^{c_H}(1-p)^{c_T}dp = \frac{c_H!c_T!}{(c_H + c_T + 1)!} \qquad (3.62)$$

which in turn gives us the distribution for p as

$$P(p|c_H, c_T) = \frac{(c_H + c_T + 1)!}{c_H!c_T!}p^{c_H}(1-p)^{c_T} \qquad (3.63)$$

which is in fact a Beta distribution with the parameters $c_T + 1$ and $c_H + 1$. We are now not interested in the value of p which maximises this expression, but rather the expected value of p,

$$\begin{aligned} \hat{p} = E[p] &= \int pP(p|c_H, c_T)dp \\ &= \frac{(c_H + c_T + 1)!}{c_H!c_T!} \int_0^1 p^{c_H+1}(1-p)^{c_T} dp \\ &= \frac{(c_H + c_T + 1)!}{c_H!c_T!} \frac{(c_H + 1)!c_T!}{(c_H + c_T + 2)!} \\ &= \frac{c_H + 1}{c_H + c_T + 2} = \frac{c_H + 1}{c + 2} \end{aligned} \tag{3.64}$$

For large sample sizes, this estimate will converge to the same result as the maximum likelihood estimate of equation 3.60, but for small samples it will tend towards $1/2$. This is more in line with the intuitive way of estimating the parameter, where we do not blindly trust the available data if the sample size is small. Also note that for many applications of Bayesian methodology we do not necessarily want to calculate the expected value of the parameter, but rather use the distribution of p to answer the questions that are relevant for the application. Also note that we can also get to an estimate of the uncertainty of our parameter of estimation *e. g.* by calculating the variance of the distribution.

In most scenarios, the prior used above is a bit over simplified. The more common general form

$$P(p) \propto p^{\alpha-1}(1-p)^{\alpha-1} \tag{3.65}$$

instead gives us an estimate of p, based on calculations similar as those above, as

$$\hat{p} = \frac{c_H + \alpha}{c + 2\alpha} \tag{3.66}$$

If we are not being very strict, the parameter α can be thought of how much weight we put on the prior compared to the data. A common value of the parameter would be one, but a larger α may also be suitable, *e. g.* if data is very noisy.

Bayesian Parameter Estimation in Hierarchical Graph Mixtures

Now, equation 3.66 holds for a binary variable, but within the framework of hierarchical graph mixtures we would rather be interested in an expression for discrete distributions of N outcomes. Using a prior distribution on the form

$$P(\boldsymbol{p}) = \prod_{i=1}^{N} p_i^{\alpha-1} \tag{3.67}$$

makes the posterior distribution of the parameters $\boldsymbol{p}$ a multi-Beta distribution,

$$P(\boldsymbol{p}|\boldsymbol{c}) = \frac{\Gamma(c + N\alpha)}{\prod_i \Gamma(c_i + \alpha)} \prod_i p_i^{c_i+\alpha-1} \tag{3.68}$$

Finding the expected values of the parameters $\boldsymbol{p}$ is slightly involved, but the result is

$$\hat{p}_i = E[p_i] = \frac{c_i + \alpha}{c + N\alpha} \tag{3.69}$$

This expression is now directly usable for all estimations involving discrete distributions within our framework. If we now look at equation 3.67, it is apparent that it is possible to specify the prior distribution on the same parametric form as another discrete distribution plus a weight parameter α. If we assume that this "prior" has parameters p_i^p, we can write the expected value of our parameters $\hat{p}_i$ as

$$\hat{p}_i = \frac{c_i + \alpha p_i^p}{c + \alpha} \tag{3.70}$$

Let us now consider the case of continuous random variables represented by Gaussian distributions. The maximum likelihood estimates of the mean vector and covariance matrix of a multivariate Gaussian distribution are

$$\boldsymbol{\mu}^0 = \frac{1}{c}\sum_{j=1}^{c} \boldsymbol{x}_j \tag{3.71}$$

$$\Sigma^0 = \frac{1}{c}\sum_{j=1}^{c} (\boldsymbol{x}_j - \boldsymbol{\mu}^0)(\boldsymbol{x}_j - \boldsymbol{\mu}^0)^T \tag{3.72}$$

where $\boldsymbol{x}_j$ are samples from a data set of size c. If we want to make a Bayesian parameter estimation, we would start with the conjugate prior distribution (a prior distribution which results in a posterior from the same family) of the multivariate Gaussian distribution with unknown mean and covariance matrix, the Wishart-Gaussian distribution:

$$P(\boldsymbol{m}, Q|\boldsymbol{\mu}^*, \Sigma^*, \alpha) \propto |Q|^{(a-d-1)/2} \exp\left(-\frac{1}{2}\operatorname{tr}(\alpha Q \Sigma^*)\right) \exp\left(-\frac{1}{2}\operatorname{tr}(\alpha Q(\boldsymbol{\mu}^* - \boldsymbol{m})(\boldsymbol{\mu}^* - \boldsymbol{m})^T\right) \tag{3.73}$$

This is a joint density over the mean $\boldsymbol{m}$, and the inverse of the covariance matrix, Q. The expectation of $\boldsymbol{m}$ is $\boldsymbol{\mu}^*$, while the expectation of Q is Σ^{*-1}. If we calculate the posterior distribution over the mean and covariance, we find that it is itself a Wishart-Gaussian distribution, with the new parameters $\boldsymbol{\mu}$, Σ, and β,

$$\beta = c + \alpha \tag{3.74}$$

$$\boldsymbol{\mu} = \frac{c\boldsymbol{\mu}^0 + \alpha\boldsymbol{\mu}^*}{c + \alpha} \tag{3.75}$$

$$\Sigma = \frac{c\Sigma^0 + \alpha\Sigma^* + \frac{c\alpha}{c+\alpha}(\boldsymbol{\mu}^0 + \alpha\boldsymbol{\mu}^*)(\boldsymbol{\mu}^0 + \alpha\boldsymbol{\mu}^*)^T}{c + \alpha} \tag{3.76}$$

where c is the number of samples in the data set. The derivations of these expressions are somewhat lengthy but can be found in [Keehn, 1965; Holst, 1997]. We can interpret these expressions as that it is possible to find the Bayesian estimates of the parameters by combining maximum likelihood estimates $\boldsymbol{\mu}^0$ and Σ^0 with a weighted "prior" estimate $\boldsymbol{\mu}^*$ and Σ^*, corresponding to α samples.

Hierarchical Priors

Consider again expressions 3.70, 3.75 and 3.76. All these estimates are on a form that, in a loose sense, combines a maximum likelihood estimate with a prior belief expressed on the same parametric form. The significance of the prior is in all these cases specified by a parameter α, which can roughly be interpreted as how many data points our prior belief represents. For example, we can specify the prior of a Gaussian as another Gaussian along with a parameter specifying the relative weight of the prior. This allows us to easily represent and specify priors hierarchically, where one estimated distribution is used as prior for estimating another distribution, for both Gaussians and discrete distributions. An illustration of this procedure can be seen in figure 3.9.

Similarly, we can specify the prior of a graph model by specifying the prior, or hierarchy of priors, for each factor $\Psi(u_i)$ individually. Estimation of the graphical model then becomes a matter of estimating each factor individually as described before, regardless of whether this distribution is Gaussian, discrete, or a mixture.

We can use the same approach for mixture models, where we specify priors individually for each component. However, in a mixture we also need to specify the prior for the mixture proportions. As this is a discrete distribution, we can do this in the same manner as for all other discrete distributions. That is, for a mixture we specify one hierarchy of priors for the mixture proportions, and one hierarchy of priors for each component.

Note that this approach is not a completely Bayesian, as this would require us to integrate over all parameters of the complete model. This may however be prohibitively computationally expensive, since we in the general case will have to resort to *e. g.* Monte Carlo methods to calculate these integrals. Although the suggested approach of performing Bayesian parameter estimation component-wise may fail to faithfully describe the true posterior distribution, it does provide for easy encoding of prior knowledge and a much more robust way of estimating parameters, allowing us to reliably use much smaller data sets than otherwise possible.

3.10 Conclusions

Using the expressions described earlier, we can estimate the parameters, calculate conditionals, and find marginals for both graph models and mixture models independent of their component distributions. The only requirement on the component distributions is that a rather small set of operations can be performed on them. Since both the mixture model and the graph model fulfil this requirement, they

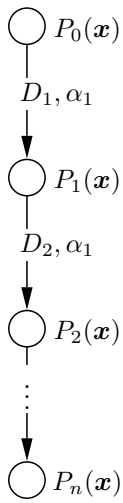

Figure 3.11: Hierarchical priors. To estimate the parameters of the distribution of interest, in this case $P_n(\boldsymbol{x})$, we first encode our prior knowledge in distribution $P_n(\boldsymbol{x})$. If we do not have any particular prior information, we may *e. g.* choose this in the discrete case as equally distributed. We then use this distribution, our prior weight α_1 and data set D_1 to estimate the parameters of distribution $P_1(\boldsymbol{x})$, which in turn is used to estimate $P_2(\boldsymbol{x})$ using data set D_2 etc.

can both be used as component distributions in both graphs and mixtures themselves. This allows us to create hierarchical combinations of the two models, such as graphs of mixtures of graphs and so on.

Using this, we can use different dependency structures for different clusters in data. This makes it possible to effectively model common penomena in complex industrial systems. The fact that we can easily create hierarchies of prior distributions also allows us to encode both prior knowlede and use different types of data sets for estimation, an example of which can be found in chapter 5, where we combine prototypical and case data for diagnosis.

The implementation of this modelling framework becomes, if not simple, at least very straightforward. General expressions are provided for the most important operations on the only two higher order models, *i. e.* the mixture model and the graph model. A complete system naturally includes some simpler distributions that do not consist of a number of component distributions, *e. g.* discrete distributions and Gaussian distributions exemplified here, to serve as the basic building blocks

when we construct hierarchies of mixtures and graph models. Again, implementing necessary functionality for these distributions does not in general pose any practical problem. We have implemented and tested the modelling framework on a wide range of data analysis problems, and have found it to be a powerful tool for practical statistical modelling.

The Hierarchical Graph Mixture modelling framework provides a simple and consistent way of describing a wide variety of statistical models. The implementation is straightforward, and provides a very flexible yet simple modelling tool for a wide variety of tasks.

Appendix A: Estimating the Parameters of Unsupervised Mixture Models

Let us now find the maximum likelihood estimates of the parameters, *i. e.* the parameters that maximise $P(D|M)$, for a mixture model. The derivations are not really complicated, and the main part of them can be found on some form in most introductory texts on mixture models and Expectation Maximisation. We include them as they are useful for understanding how mixtures can be estimated independently of the parameterisations of the component distributions.

Firts, we assume that all N samples $\boldsymbol{y}^{(\gamma)}$ of data set D are independently identically distributed, meaning that we the expression we would like to maximise is

$$P(D|\boldsymbol{\theta}) = \prod_{\gamma} P(\boldsymbol{y}^{\gamma}|\boldsymbol{\theta}) \tag{3.77}$$

where $\boldsymbol{\theta}$ denotes the parameters of model M. As this expression is difficult to differentiate, let us instead maximise the logarithm of this expression for a mixture,

$$\log P(D|\boldsymbol{\theta}) = \sum_{\gamma} \log P(\boldsymbol{y}^{\gamma}|\boldsymbol{\theta}) = \sum_{\gamma} \log \sum_{\boldsymbol{i}\in S_z} \pi_{\boldsymbol{i}} f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{i}}) \tag{3.78}$$

Now, us find the source proportions $\pi_{\boldsymbol{i}}$ that maximise this expression. They are a little tricky since the likelihood has to be maximized under the constraint that $\sum_{\boldsymbol{i}\in S_z} \pi_{\boldsymbol{i}} = 1$. This can be performed by introducing a Lagrange multiplier λ and instead maximizing the expression

$$\sum_{\gamma} \log \sum_{\boldsymbol{i}\in S_z} \pi_{\boldsymbol{i}} f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{i}}) + \lambda \left(\sum_{\boldsymbol{i}\in S_z} \pi_{\boldsymbol{i}} - 1 \right) \tag{3.79}$$

When the constraint is satisfied the second term is zero, so if we can find a λ for which the constraint is satisfied at the global maximum, this will also be a maximum for the original problem. Equating the partial derivative of the above expression to

zero gives us

$$\begin{aligned}
0 &= \frac{\partial}{\partial \pi_{\boldsymbol{i}}} \sum_{\gamma} \log \sum_{\boldsymbol{j} \in S_z} \pi_{\boldsymbol{j}} f_{\boldsymbol{j}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{j}}) + \lambda \left(\sum_{\boldsymbol{j} \in S_z} \pi_{\boldsymbol{j}} - 1 \right) \\
&= \sum_{\gamma} \frac{f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{i}})}{\sum_{\boldsymbol{j} \in S_z} \pi_{\boldsymbol{j}} f_{\boldsymbol{j}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{j}})} + \lambda \\
&= \sum_{\gamma} \left(\frac{\pi_{\boldsymbol{i}} f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{i}})}{\sum_{\boldsymbol{j} \in S_z} \pi_{\boldsymbol{j}} f_{\boldsymbol{j}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{j}})} \right) \frac{1}{\pi_{\boldsymbol{i}}} + \lambda \\
&= \sum_{\gamma} P(v_{\boldsymbol{i}} | \boldsymbol{y}^{(\gamma)}) \frac{1}{\pi_{\boldsymbol{i}}} + \lambda \qquad \Rightarrow \\
-\lambda \pi_{\boldsymbol{i}} &= \sum_{\gamma} P(v_{\boldsymbol{i}} | \boldsymbol{y}^{(\gamma)}) \qquad (3.80)
\end{aligned}$$

Inserting the λ for which the constraint is satisfied, *i. e.* $-\lambda = N$, gives us

$$\pi_{\boldsymbol{i}} = \frac{\sum_{\gamma} P(v_{\boldsymbol{i}} | \boldsymbol{y}^{(\gamma)})}{N} \qquad (3.81)$$

Thus, the parameters $\pi_{\boldsymbol{i}}$ can be estimated independently from the type of component distribusions used in the mixture.

To maximize the likelihood with respect to the parameters $\theta_{\boldsymbol{i}}$ of a component distribution $f_{\boldsymbol{i}}$, it i useful to rewrite the derivative as

$$\begin{aligned}
0 &= \frac{\partial}{\partial \theta_{\boldsymbol{i}}} \sum_{\gamma} \log \sum_{\boldsymbol{j} \in S_z} \pi_{\boldsymbol{j}} f_{\boldsymbol{j}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{j}}) \\
&= \sum_{\gamma} \frac{\pi_{\boldsymbol{i}} \frac{\partial}{\partial \theta_{\boldsymbol{i}}} f_{\boldsymbol{j}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{j}})}{\sum_{\boldsymbol{j} \in S_z} \pi_{\boldsymbol{j}} f_{\boldsymbol{j}}(\boldsymbol{y}^{(\gamma)})} \\
&= \sum_{\gamma} \left(\frac{\pi_{\boldsymbol{i}} f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{i}})}{\sum_{\boldsymbol{j} \in S_z} \pi_{\boldsymbol{j}} f_{\boldsymbol{j}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{j}})} \right) \frac{\frac{\partial}{\partial \theta_{\boldsymbol{i}}} f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{i}})}{f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{i}})} \\
&= \sum_{\gamma} P(v_{\boldsymbol{i}} | \boldsymbol{y}^{(\gamma)}) \frac{\partial}{\partial \theta_{\boldsymbol{i}}} \log f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{i}}) \qquad (3.82)
\end{aligned}$$

This equation suggests that the parameters of the component distributions can be found by using whatever expression that maximises its parameters given the data but where each sample is weighted by how much it belongs to a certain component. As, if we have more components than one, it is impossible to calculate these weights without knowing the parameters of the component distributions, we have to iterate these calculations within the Expectation Maximisation algorithm.

Although it is relatively easy to show that this weighted estimation does indeed hold for *e. g.* Gaussian and discrete distributions [Holst, 1997; McLachlan and Peel,

2000], we will not include these calculations here. If our component distributions are mixtures, estimation becomes no more difficult than the estimation of a single mixture model, as a mixture of mixture models is indeed again just a mixture. Instead, let us now turn to the second important distribution form apart from mixtures in our framework, the graphical model.

If we use a graphical model as the component distribution $f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{i}})$, the above expression can be written as

$$\begin{aligned} 0 &= \sum_{\gamma} P(v_{\boldsymbol{i}}|\boldsymbol{y}^{(\gamma)}) \frac{\partial}{\partial \theta_{\boldsymbol{i}}} \log f_{\boldsymbol{i}}(\boldsymbol{y}^{(\gamma)}; \theta_{\boldsymbol{i}}) \\ &= \sum_{\gamma} P(v_{\boldsymbol{i}}|\boldsymbol{y}^{(\gamma)}) \frac{\partial}{\partial \theta_{\boldsymbol{i}}} \log \prod_{i} \boldsymbol{\Psi}(u_i) \\ &= \sum_{i} \left(\sum_{\gamma} P(v_{\boldsymbol{i}}|\boldsymbol{y}^{(\gamma)}) \frac{\partial}{\partial \theta_{\boldsymbol{i}}} \log \boldsymbol{\Psi}(u_i) \right) \end{aligned} \tag{3.83}$$

One solution to the above expression can then be found through finding the parameters for which

$$0 = \sum_{\gamma} P(v_{\boldsymbol{i}}|\boldsymbol{y}^{(\gamma)}) \frac{\partial}{\partial \theta_{\boldsymbol{i}}} \log \boldsymbol{\Psi}(u_i) \tag{3.84}$$

for all factors $\boldsymbol{\Psi}(u_i)$. Now, this expression is again on the same form as equation 3.82. As we have already noted, this means that we can use weighted estimation for finding the parameters of $\boldsymbol{\Psi}(u_i)$, at least in the case of Gaussians and discrete distributions. Thus, it follows that estimating the parameters of a graphical model can be performed in the same manner, estimating each factor separately using a weight that determines how much the graphical model belongs to the sample. In other words, we can perform the parameter estimation of our mixture model independent of the specific parameterisation of the graphical model.

Acknowledgements

The construction and testing of the sliver detection model of section 3.8 was performed by Anders Holst. The work of preparing the data for the tests was performed by Anders Holst and Per Kreuger.

Chapter 4

Structure Learning

4.1 Approaches to Structure Learning

Learning the statistical *structure* of a process that generates data, expressed by *e. g.* descriptions of dependencies between attributes or clusters in data, is highly useful both for understanding and answering questions about the underlying generating process, and for automatically specifying the structure of a statistical model such as the Hierarchical Graph Mixtures.

When trying to automatically find the structure specification for a statistical model, there are several possible approaches:

- Using external measures and methods that do not depend on the type of model or the application, such as determining the dependency structure using common correlation measures.

- Choosing the model structure based on the performance of the model in the intended application, such as choosing the dependency structure of a graphical model based on its prediction or classification performance on a test data set.

- Not firmly specifying the structure at all, and instead sample from many or all possible model structures using *e. g.* a Markov Chain Monte Carlo method.

Here, we will focus on deriving the dependency structure of a data set through external measures for sequential data. There are several ways of determining correlation between series, most of them suffering from specific problems when applied to real-world data. We will introduce a new measure of interdependency for sequential data that does not suffer from many of the issues regular correlations suffer from within this context, that also allow us to determine the most significant delays between the correlations. This measure of pairwise dependency can then be used to create a graphical representation, *e. g.* using Chow and Liu's algorithm (see section 3.8).

4.2 Dependency Derivation

Finding dependencies and the creation of dependency graphs from data can increase the understanding of the system significantly. It is also a common step for the creation of graphical models, who rely on this information for partitioning the joint distribution of the attributes into simpler factors. We will here discuss how to measure these dependencies, without considering the details or estimated generalisation performance of a model based on them.

Correlation Measures

Correlation between attributes can be measured in a number of different ways, the perhaps most commonly used being Pearson's correlation coefficient and the covariance. In particular, the correlation coefficient can be written as

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}} \tag{4.1}$$

where x_i and y_i denote samples of X and Y respectively and $\bar{x}$ and $\bar{y}$ denotes the mean of the variables. It is essentially the covariance between X and Y, normalised to the region -1 to 1, and measures the amount and direction of linear dependence between X and Y. This is a very useful, robust measure that often gives a good indication about the dependency structure of the sequences. If, for some reason, we can be sure that there are only linear dependencies and independent samples in the data, it is also an optimal measure. When non-linear dependencies are present or samples are not independent, as in a time series, the measure might be fooled into either giving a low value of the correlation for two highly correlated variables, or to significantly overestimate the actual correlation.

The measure only calculates the linear correlation, which means that it might not detect obvious dependencies simply because they do not point in the same direction for all values. A typical example is a number of data points positioned on a circle. The points x and y coordinates are clearly dependent on each other, but the correlation coefficient is zero. The measure also only works for continuous data, and will not provide any useful information when measuring the correlation between nominal attributes.

Another problem with these measures is that there is no natural generalisation for measuring correlation between multivariate distributions. The use of *information theory* [Ash, 1967; Cover and Thomas, 1991], or more specifically the concept of *mutual information*, can provide a solution [Li, 1990].

Entropy and Mutual Information

Before introducing the mutual information and the mutual information rate, we will give a brief review of some of the basic concepts of information theory. The *entropy* of a stochastic variable [Shannon, 1948, 1951], which can be thought of as

a measure of the amount of uncertainty or the mean information from the variable, is defined as

$$H(X) = -\sum_{x \in X} P(x) \log P(x) \tag{4.2}$$

$$h(X) = -\int_S f(x) \log f(x) dx \tag{4.3}$$

in the discrete and continuous case respectively. In the discrete case, X denotes the possible outcomes of the variable, and in the continuous case S denotes the support set of the stochastic variable. The *joint entropy* $H(X, Y)$ of a pair of stochastic variables is defined, here just in the discrete case as the continous case is very similar, as

$$H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} P(x, y) \log P(x, y) \tag{4.4}$$

and the *conditional entropy* $H(Y|X)$ as

$$H(Y|X) = -\sum_{x \in X} \sum_{y \in Y} P(x, y) \log P(y|x) \tag{4.5}$$

Both the joint and conditional entropy described above can be extended to more than two variables in a straightforward manner.

If two variables X and Y are independent, $H(X, Y) = H(X) + H(Y)$. If the variables are not independent, $H(X) + H(Y)$ will be larger than $H(X, Y)$, that is some of the information in $H(X, Y)$ is included in both the marginal entropies. This common information is called the *mutual information*, $I(X; Y)$. It can also be thought of as the reduction in the uncertainty in one variable due to the knowledge of the other, or more formally

$$\begin{aligned} I(X; Y) =& H(Y) - H(Y|X) = H(X) - H(X|Y) = \\ & H(X) + H(Y) - H(X, Y) \end{aligned} \tag{4.6}$$

The mutual information is symmetrical and is always larger than or equal to zero, with equality only if X and Y are independent. The mutual information can also be written more briefly, here in both the discrete and the continuous case, as

$$I(X; Y) = \sum_{x \in X} \sum_{y \in Y} p(x, y) \log \frac{p(x, y)}{p(x)p(y)} \tag{4.7}$$

$$I(X; Y) = \int f(x, y) \log \frac{f(x, y)}{f(x)f(y)} dxdy \tag{4.8}$$

which follows directly from equation 4.6. The expressions in equation 4.7 and 4.8 makes the connection to the *Kullback-Leibler distance* [Kullback, 1959] clearly

visible. The Kullback-Leibler distance $D(f \parallel g)$ measures the distance between two probability mass functions and is defined as

$$D(f \parallel g) = \int f \log \frac{f}{g} \tag{4.9}$$

The mutual information can therefore be interpreted as the Kullback-Leibler distance between the joint distribution $f(x, y)$ and the product of the marginal distributions $f(x)$, $f(y)$. Informally, it is the distance between representing the two variables together in a joint distribution and representing them as independent.

Mutual information can also be viewed as a measure of the dependence between two variables. If the variables are independent, the mutual information between them will be zero. If they are strongly dependent, the mutual information will be large. Other interpretations of the mutual information could be the reduction of uncertainty of one variable due to the knowledge of the second, or the stored information in one variable about another. Mutual information is a general correlation measure and can be generalised to all kinds of probability distributions. It is also, given an appropriate model of the distributions, able to detect non-linear dependencies between variables.

To be able to calculate the mutual information, we have to know both the variables marginal distributions and their joint distribution. In the case of for example binary variables and linear correlation, there are straightforward methods of estimating these distributions, but in the general case we first need a basic assumption of what the distribution will look like. If we assume a too complex model, where each data point essentially has to be considered on its own, we run the risk of overfitting the model so that all variables always look highly correlated. If we on the other hand assume a simple model such that all data are normal distributed, we cannot detect anything but the linear part of the correlation. We have mainly tried two models of the distributions, normal distributions (Gaussians) and *binning* the data using a 2-dimensional grid.

If each measured variable is quantised into discrete values, or *bins*, it is very simple to calculate the mutual information between one variable and another using equation 4.7. Each marginal is discretised, and the joint distribution is modelled by the grid resulting from the two marginal distribution models. Then histograms are constructed from the data using this discretisation and from these the probabilities are estimated. In our tests, we use the Bayesian parameter estimate described in equation 3.70. Here, c_i represents the number of data points in a bin. The parameters α and p_i^p where set to 1 and the $1/M$ respectively, where M is the total number of bins used.

The number of discrete values in the grid is critical. Choosing a too fine grid with more bins than data points results in a mutual information of the logarithm of the number of data points. If too few intervals are chosen, again only the linear part of the correlation, and hardly that, will show.

The method of discretisation used also has a great impact on the results. The perhaps most straightforward approach is to find the maximum and minimum value

of the attribute in the data and then evenly divide this range into a number of bins. However, this results in a rather unstable measure, in the sense that it is very sensitive to noise and outliers in data. The distributions being modelled are often both non-symmetric and multi-modal, and have a few outliers that lie far from the rest of the data points. This has the effect that most bins will have a probability close to zero, and the effective resolution of the grid is greatly reduced. Also, using the measure over different periods of time in the series, where different outliers give different maximum and minimum values, will give very different results.

A better way to quantise the variables range is to use *histogram equalisation*. When discretizising the marginal distributions, the limits between bins are set so that all bins contain the same number of data points. This gives us a much more stable measure. It is less sensitive to both skewness and outliers and the resulting grid uses the highest resolution in the areas where there are most data points.

Another common simplifying assumption about the distributions is that they are Gaussian. This results in a measure where a linear correlation is assumed, and it is only possible to detect linear correlations in the data. To derive an expression for the mutual information between Gaussian variables, we can start from an expression of the entropy for a Gaussian distribution. The entropy of an n-dimensional Gaussian distribution can be written as

$$h(X_1, X_2, \ldots, X_n) = \frac{1}{2}\log(2\pi e)^n |C| \tag{4.10}$$

where $|C|$ denotes the determinant of the covariance matrix. Using equation 4.6 and 4.10, it is easy to calculate the mutual information for Gaussian distributions. When calculating the mutual information under these assumptions, the only parameters needed are the means and variances of the two variables and the covariance between them, all easily estimated from data.

Naturally, there is a very close relation between the mutual information based on Gaussian distributions and the common linear correlation. The mutual information can be written as

$$I(X;Y) = \frac{-\log(1-\rho^2)}{2} \tag{4.11}$$

Thus, the mutual information is in this case just a scaling of the absolute value of the correlation coefficient to a range between zero and infinity. Note however that we no longer have any notion of direction in the correlation measure.

The measures described above are general correlation measures. If we are working with sequential data, the correlation is usually measured as a function of the time shift between the series. This can then be plotted as a correllogram for visual inspection or used for automatic generation of dependency graphs.

The Mutual Information Rate

Let us now have a look at how we can extend the notion of mutual information to efficiently find dependencies between time series. Time series are sequential data;

data that evolve in some random but prescribed manner. Simply from the statement that we are dealing with time series, we can make some very reasonable assumptions about the dependency structure of the data. This basic consideration can be used to construct a correlation measure specific for sequential data, unlike the general measures described in section 4.2 [Gillblad and Holst, 2001]. The resulting measure is a more sensitive and accurate measure of the dependencies in time series.

To construct such a measure, we will start from, loosely speaking, an expression for the uncertainty of a sequence which corresponds to the entropy of a single variable. If we have a sequence of n random variables, this uncertainty can be defined as how the entropy of the sequence grows with n. This is called the *entropy rate* $H_r(X)$ of the process X, and can be defined in two ways,

$$H_r(X) = \lim_{n\to\infty} \frac{1}{n} H(X_1, X_2, \ldots, X_n) \tag{4.12}$$

$$H_r(X) = \lim_{n\to\infty} H(X_n | X_{n-1}, X_{n-2}, \ldots, X_1) \tag{4.13}$$

The definitions in 4.12 and 4.13 correspond to two different notions of the entropy rate. The first is the per symbol entropy of the n random variables, and the second is the conditional entropy of the last variable given the past. The two notions are equal in the limit if the process is stationary. Although the first is perhaps more common, here we will use the second notion since it will make the derivations more natural. The final results are the same for both expressions.

Based on the entropy rate, we can construct a measure of the *mutual information rate*. It can be defined as

$$I_r(X;Y) = \lim_{n\to\infty} I(X_n | X_{n-1}, \ldots, X_1; Y_n | Y_{n-1}, \ldots, Y_1) \tag{4.14}$$

corresponding to the second notion of entropy rate defined in equation 4.13. This can be seen as a measure of the total dependence, or the total amount of information in common, between sequence X and Y. To relate it to the entropy rate, we can also write the mutual information rate as

$$I_r(X;Y) = H_r(X) + H_r(Y) - H_r(X,Y) \tag{4.15}$$

This can be derived directly from equation 4.14 using expression 4.13. Informally, it can be understood by considering the entropy rate of a sequence as analogous to the entropy of a stochastic variable and then applying equation 4.6.

Now we have a mutual information rate expressed in the entropy rates of the two sequences. This way, the mutual information rate measures the complete dependence between the sequences. In the limit, the shift between the sequences is irrelevant. Working with finite sequences though, the entropy rate of equation 4.12 or 4.13 is impossible to estimate perfectly, since the distributions used in the calculations contain infinitely many variables. When we restrict ourselves to the use a finite number of variables, the shift inevitably becomes important. Also, for some applications, we would actually like to have a localised measure that measures the

direct dependence between one sequence and the other with a specific shift between them.

We can make a reasonable estimate using a Markov assumption, *i. e.* we assume that the process has a limited memory so that the value of a variable is dependent only on the closest earlier values. A first order Markov assumption, or assuming that the value of the process only depends on the previous value, can be written as

$$P(X_n|X_{n-1}, X_{n-2}, \ldots, X_1) = P(X_n|X_{n-1}) \tag{4.16}$$

This is a very reasonable assumption for many processes, at least as an approximation. A second order or higher Markov assumption can of course also be made, but the amount of data required to estimate the joint distributions will increase significantly.

When we make the Markov assumption, we also have to take into account the shift of the sequences. If we denote this shift d, using a first order Markov assumption and assuming stationary sequences, the mutual information rate $I_r(X;Y;d)$, can be simplified to entropies of joint distributions as

$$\begin{aligned} I_r(X;Y;d) &= H(X_n|X_{n-1}) + H(Y_n|Y_{n-1}) - \\ &\quad H(X_n, Y_{n-d}|X_{n-1}, Y_{n-d-1}) \\ &= H(X_n, X_{n-1}) - H(X_n) + H(Y_n, Y_{n-1}) - H(Y_n) - \\ &\quad H(X_n, Y_{n-d}, X_{n-1}, Y_{n-d-1}) + H(X_n, Y_{n-d}) \end{aligned} \tag{4.17}$$

using equation 4.16, $H(X_{n-1}) = H(X_n)$ and the fact that $H(X|Y) = H(X,Y) - H(Y)$ [Cover and Thomas, 1991]. The largest joint distribution that needs to be estimated contains four variables, which can be rather difficult depending on the model of the distribution used. Making a second order Markov assumption leads to a largest distribution of six variables, a third order assumption eight variables and so on, which might make them very difficult to reliably estimate from reasonable amounts of data.

To increase our understanding of this approximation of the mutual information rate, we can rewrite the expression in equation 4.17 to a sum of separate informations between distributions that are not conditional as

$$\begin{aligned} I_r(X;Y;d) &= I(X_n, Y_{n-d}; X_{n-1}, Y_{n-d-1}) + I(X_n; Y_{n-d}) - \\ &\quad I(X_n; X_{n-1}) - I(Y_n; Y_{n-1}) \end{aligned} \tag{4.18}$$

Thus the mutual information rate can be interpreted as the information between (X_n, Y_{n-d}) and (X_{n-1}, Y_{n-d-1}) plus the information in X_n about Y_{n-d}, then subtracting both the information in X_n about X_{n-1} and in Y_n about Y_{n-1}. The second term of the expression, $I(X_n; Y_{n-d})$, is the normal mutual information measure. When both sequences X and Y are completely memoryless, the information rate reduces to the mutual information. The last two subtractive terms have a noise reducing property when plotting the correllogram, subtracting each series own information rate, and the first term adds an estimate of the joint information rate of the sequences.

Once again, like in section 4.2, the distributions can be modelled by discretising the values using a multi-dimensional grid. First each marginal is discretised, and then two and four dimensional grids are constructed for the joint distributions based on these marginal grids. With this method, using M bins on the marginal distributions leads to M^4 bins when estimating the joint distribution of four variables. Using a second order Markov assumption, this estimation requires M^6 bins. Therefore, a first order Markov assumption is necessary, except perhaps when there is an extremely large amount of available data. Still, the value of M needs to be kept low to keep the number of bins in the joint distributions as low as possible. This might in turn result in a measure with too low resolution on the marginal distributions to be able to detect the dependencies in the data at the same time as the joint distributions suffer from random effects due to too few available data points.

Instead of using bins, Gaussian distributions can be used, resulting in a linear measure tailored for time series. All entropies are calculated using equation 4.10, and the distributions themselves are easily estimated from data. This is a much more robust measure; it requires much less data to estimate these Gaussians reliably than the use of a grid. The drawback is of course that still essentially only linear dependencies can be discovered.

All in all, the mutual information rate is a sound and relatively easily calculated measure of dependence in a series. It usually produces significantly better results than the general measures of section 4.2, since it takes the sequential structure of the data into account. The main drawback is that it needs more data to be estimated reliably. The Markov assumption also introduces another consideration. Dependencies that are visible at several shifts between the sequences will be handled correctly if this spread of the dependence is dependent on the Markov properties of the series. If the sequences on the other hand interact at two or more speeds this will not be considered and only the strongest correlation will show. However, in practise this may often be sufficient.

Test Results

The described correlation measures have been tested on both synthetic test data and real data from mainly two different process industries, a paper mill and a chemical production plant.

Results on Synthetic Data

Synthetic data were generated for testing purposes and to illustrate the effects of the different models. The process generation model is depicted in figure 4.1, where $x(n)$ and $y(n)$ are random time series generated independently from each other. From $x(n)$, two new time series are generated, one that is delayed 17 time steps and multiplied by a factor of 2, and one that is delayed 4 time steps and multiplied with a factor of 5. The sum of these series as well as $y(n)$ multiplied by the factor

5 is the output of the model, called $z(n)$. The original series $x(n)$ can then be compared to the output series $z(n)$, trying to detect the correlations. Using this model, $z(n)$ is clearly correlated with $x(n)$ at both time delay 4 and 17, the second correlation being somewhat weaker than the first. $y(n)$ represents additive noise to the output signal.

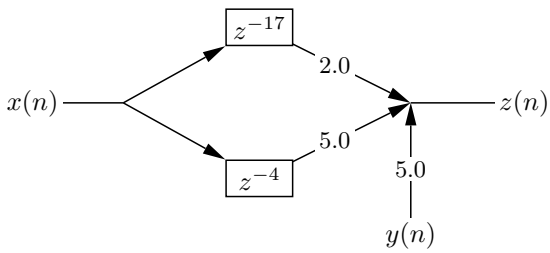

Figure 4.1: The model generating the synthetic data.

The top left diagram in figure 4.2 depicts the correllograms generated by the linear mutual information, *i. e.* using the mutual information with Gaussian distributions. The x-axis represents the delay between the series and the y-axis the degree of correlation. A thin line in the centre of the diagram shows delay 0. The diagram clearly shows a peak, although not very distinct, at delay 4. This corresponds well to the process generation model. However, the other delay in the model of 17 does not show up as a peak in the diagram. Using the linear version of the mutual information rate, the second peak is clearly detectable. This is shown in the top right of figure 4.2. In comparison, the mutual information rate produces a much sharper and more exact diagram than the mutual information, showing distinct peaks at both delay 4 and delay 17, the peak at seventeen being lower than the peak at 4. Looking at the process generation, this is what we would expect the correlation measure to produce.

In the lower two diagrams of figure 4.2 the correllograms from the binned version of the mutual information and the mutual information rate are shown. In both cases, 20 bins were used on the marginals. The binned mutual information looks much the same as the linear mutual information, not detecting the second, weaker correlation at delay 17. The binned mutual information rate detects both peaks, although not very clearly. There is a large amount of noise and artifacts present, resulting from the binning of data. It is not very surprising, however, that the linear version of the measure performs better here since there are only linear dependencies present in the data.

The generated series $x(n)$ and $y(n)$ are both first order Markov processes. Since the mutual information rate measure used here is based on a first order Markov assumption, a good result using the mutual information rate on these series is not that surprising. The improvement using the mutual information rate instead of the

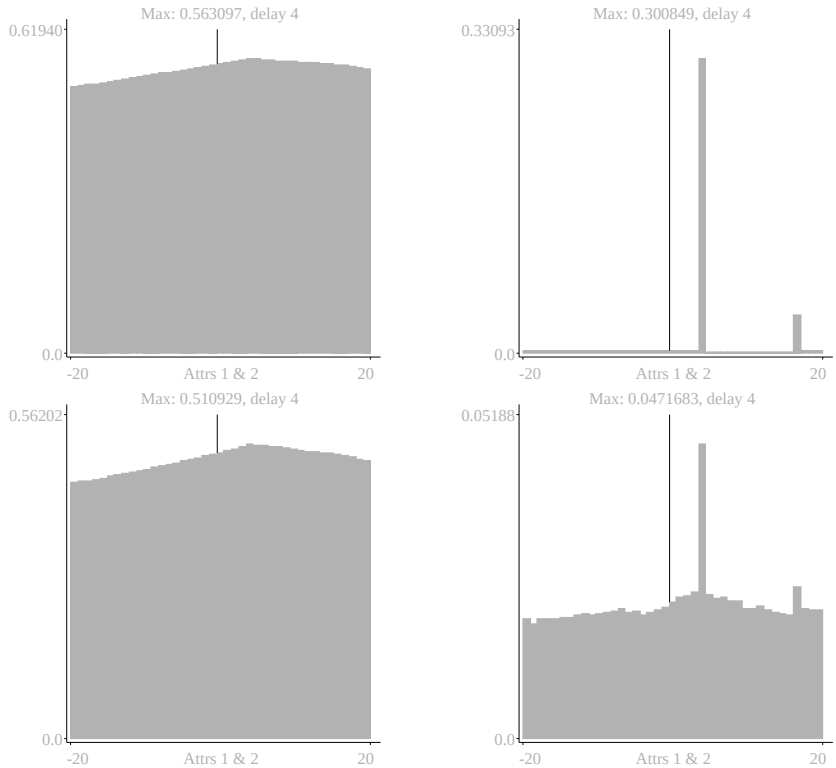

Figure 4.2: Results on synthetic data.

mutual information would probably be lower if the processes were of higher order.

Results on Industrial Process Data

The measures presented here have all been used on several real data sets, mainly from a paper mill and a chemical plant. Here we will present a couple of examples, both taken from the chemical plant application. Only the linear versions of the measures are shown since they proved to be more useful than the binned versions. The names of the attributes were given as short abbreviations to not be directly recognisable, where an X means a measured variable, C a controlled variable and Y a significant output variable. The delay is measured in minutes.

The top left diagram of figure 4.3 shows the linear correlation between the measured variable $X3$ and the controlled variable $C63$. It is an example of a well behaved, linear correlation with a short and reasonable time delay. The correllogram shows just one clear peak at delay -5, indicating that this probably is a

real correlation between the attributes and not produced by artifacts in the data. The mutual information rate correllogram for the same attributes in the top right diagram of figure 4.3 shows the same behaviour. It is a bit more peaky and shows much lower correlation, but the peak is at almost the same place, delay -6, as in the mutual information correllogram.

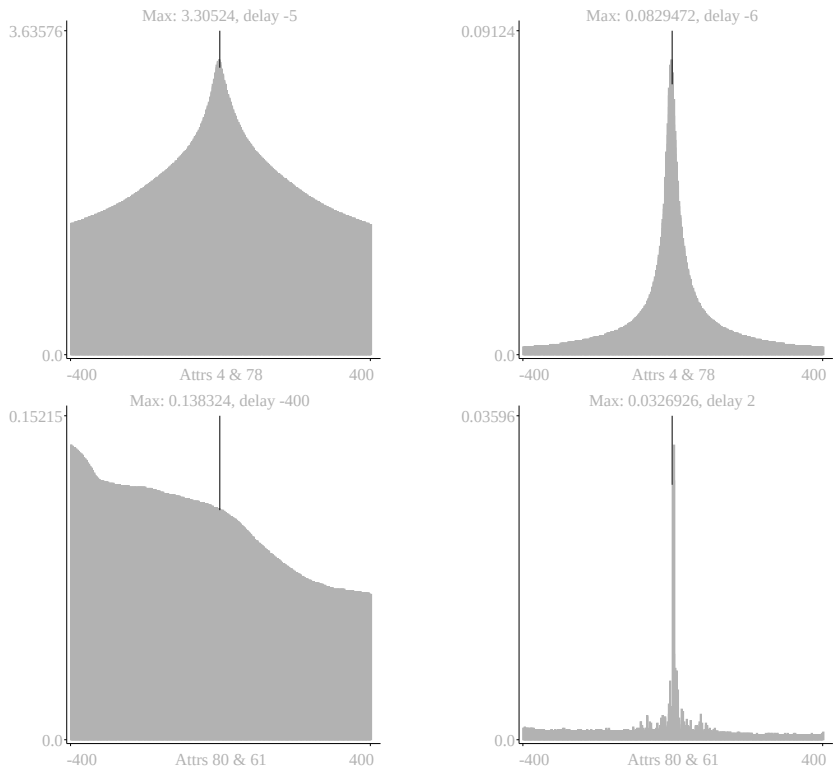

Figure 4.3: Results on chemical plant data.

In the lower left diagram of figure 4.3 the linear mutual information between attribute $X48$ and Y is shown. The correllogram is very smooth, although somewhat low, but the measure is obviously fooled by some general trend in the data since it is constantly increasing with decreasing values of the delay. The maximum is at -400, simply because that is the chosen plot range, and is a too long delay to be considered reasonable in this case. The mutual information rate on the other hand, shown in the lower right diagram of figure 4.3, shows a clear peak at delay 2. That is a plausible value of the delay between the sequences, although the value of the correlation is rather low. The information rate diagram is not at all as smooth as the mutual information, showing several small spikes which are very likely effects

of noise and oddities in the data.

Generally, the relationship between the mutual information and the mutual information rate is the same as in these two examples. The information rate diagram shows less correlation, is more peaky and also more sensitive to noise than the mutual information.

Conclusions and Practical Considerations

Both the mutual information and the mutual information rate suffer from a tendency to be fooled by random effects in sequential data, the information rate somewhat less so than the mutual information. One useful quality though is that the measures are not always fooled by the same behaviour in data, which makes reliable dependency derivation easier.

As discussed earlier, one difference between the histogram model and the linear model (using Gaussian distributions) is that the histogram model can potentially find non-linear relations which the linear model cannot find. On the other hand, the histogram model is more complex and therefore more sensitive to noise and random fluctuations. The linear model on its side is sensitive to extreme values in a way that the histogram is not. If one of the series has a highest peak somewhere and the other series also has an extreme value at some point, selecting a delay that aligns those extreme values will give a strong peak in the correllogram, almost regardless of how the series behaves relative to each other at other times. This is a problem both for the mutual information and the mutual information rate. It introduces a risk that the wrong delay between the series is selected due to purely random extreme values, and of course that the correlation might seem stronger than it actually is. Because of this both kind of diagrams could be used and a feature should hopefully give some evidence in both of them if it is significant.

A similar trade-off exist between the mutual information diagrams and those showing the mutual information rate. Using mutual information or the normal correlation coefficient between time series tends to give a too high value of the correlation. This happens because that if the time series moves slowly enough, the relation between the series at one point in time is likely to be maintained for several time steps. This means that pure random coincidences between the series gets multiplied with a factor depending on how slow the series are, making that correlation seem more significant than it is. The mutual information rate on the other hand, which only considers the new information in every step, correctly compensates for that effect but instead requires a more complicated model to estimate, which makes it more sensitive to noise.

In practise it turns out that the diagrams using histograms have quite little interesting variation, whereas the linear diagrams show more features, some of which are clearly anomalous. As expected, the diagrams with information rates show much lower levels of correlations and more emphasised peaks, *i. e.* not so smeared out over several different delays, but also more noise in the form of random spikes in the correllogram.

All in all it seems that the linear information rate is the one that gives the most reliable indications of correlations between the time series. However, due to the different trade-offs, a general rule is that a feature, a peak, should appear using at least two of the methods to be considered significant.

4.3 A Note on Learning Graphical Structure from Data

Learning the graphical structure from data is one of the fundamental tasks for data-driven graphical modeling. However, as the number of possible network structures is exponential in the number of attributes, finding the correct structure is NP-hard. Several approaches have been proposed, usually based on some kind of heuristic search through the space of possible graphs. Many of these methods are quite complex and difficult to implement, but if we are only interested in a suitable description of the joint distribution and not an explicitly correct description of causal or correlational relationships, rather simple solutions can work quite well in practise.

As we have already briefly touched upon in chapter 3, such a simple method is the greedy generation of a probabilistic dependency tree [Chow and Liu, 1968]. If we start by finding all pairwise correlations as described earlier in this chapter, we can find a suitable tree structure by sorting the dependencies according to strength, and then adding one dependency at a time, starting with the strongest one. If the addition of a dependency to the graph generates a cycle, we skip this edge and proceed with the next dependency.

The result is a graph that often describes the joint distribution rather well. If we combine this with a mixture model, and generate one tree for each component in the mixture, we often end up with a rather good model, capturing most important features of the joint distribution. The mixture may be unsupervised, or, as is commonly the case in practical applications, supervised, where the component indicator class typically is a specified mode of production, product type *etc.* . This is the standard modeling assumption for many scenarios where we have applied the HGMs of chapter 3.

If we want to learn causal networks, useful for describing the actual causal relationships in data, the situation is a litte more difficult. Pearl [Pearl, 2000] provides algorithms that potentially can find such a structure, but their comutational complexity makes them very difficult to use in practise. An alternative could be to use an ordering-based search through all possible combinations of causal dependencies [Teyssier and Koller, 2005; Friedman and Koller, 2003], where we assume an initial ancestral ordering of the attributes which we then modify according to the fitness of the best possible causal network that fits this ordering. Finding this network given an ordering is considerably less computationally demanding than without, and the search therefore becomes of reasonable complexity.

Although finding the causal dependencies of real-world industrial data would be highly useful in order to understand the processes generating the data, the size

and complexity of the data sets are currently prohibitive for these methods, and remains an important area of future research.

4.4 A Note on Finding the Number of Components in a Finite Mixture Model

Assessing the number of components, or *order*, of a finite mixture model is an important problem that is yet to be completely resolved. If the components are to be interpreted as meaningful clusters in data, assessing the number of components, or clusters, is obviously important. When using a mixture model purely for density estimation, without regard to possible interpretations of the components, assessing the order of the model is still important for regularisation purposes to assure proper generalisation perfomance of the model. The problem is very relevant to automatic generation of the structure of the Hierarchical Graph Mixtures discussed in chapter 3, but as it is outside the scope of this text we will here only give a brief introduction to the problem.

One straightforward approach is to estimate the number of modes of a distribution [Hartigan and Mohanty, 1992; Fisher *et al.*, 1994]. Visualisation methods, such as the modified percentile plot [Fowlkes, 1979], can also be used to help the user identify a suitable number of components. However, the components of the mixture have to be well separated in the data to be detectable with these methods. This is somewhat problematic, as a mixture distribution can be unimodal if the components are not far enough apart.

A somewhat more direct approach would be to compare the likelihood of a number of proposed models given the data. For example, we can perform a hypothesis test, using the likelihood ratio between two models with different numbers of components as a test statistic [Duraurajan and Kale, 1979; Polymenis and Titterington, 1999].

Another useful approach is to assess the order of a mixture by comparing the likelihood of models penalised with the number of parameters of the model. In practise, this amounts to finding the model for which the penalised log-likelihood l_p is minimised, where

$$l_p = \log(P(D|M)) + f_p(M) \tag{4.19}$$

$P(D|M)$ represents the likelihood of the data D given the model M, and $f_p(M)$ the penalty for model complexity. Commonly, two forms are used for $f_p(M)$. The first is *Akaike's information criteron* (AIC) [Akaike, 1974], where $f_p(M)$ is equal to the number of free parameters of the model. However, the use of this measure has a tendency to overestimate the correct number of components. Therefore, the *Bayesian information criterion* [Schwartz, 1978] is perhaps more commonly used. It is similar to the AIC, but multiplies the penalty term $f_p(M)$ used in the AIC by the logarithm of the number of examples we calculate the log-likelihood from. This increases the penalty term, and models of lower complexity tend to be selected.

Although these methods can be useful in some modelling situations, how to hierarchically determine the presence of a mixture and its correct number of components in a HGM setting remains an area for future research.

Chapter 5

Incremental Diagnosis

5.1 Introduction

In many diagnosis situations it is desirable to perform a classification in an iterative and interactive manner. All relevant information may not be available initially but some of it must be acquired manually or at a cost. The matter is often complicated by very limited amounts of knowledge and examples when a new system to be diagnosed is initially brought into use. Other complicating factors include that many errors arise because of misuse or wrongly setup equipment, and incorrect answers and inputs to the diagnosis system. Here, we will describe a novel incremental classification system based on a statistical model that is trained from empirical data, and show how the limited available background information can still be used initially for a functioning diagnosis system [Gillblad and Holst, 2006; Gillblad *et al.*, 2006].

5.2 Practical Diagnosis Problems

Real world diagnosis is often complicated by the fact that all relevant information is not directly available. In many diagnosis situations it is impossible or inconvenient to find all feature values to a classifier before being able to make a classification, and we would therefore like the classifier to act as an interactive decision support system that will guide the user to a useful diagnosis.

A typical example of this could be the diagnosis of faults in a vehicle that needs servicing. The mechanics usually need to check parts and subsystems until the cause of a malfunction is discovered. This procedure is often very time consuming, involving inspection of parts and systems that can be difficult to get at and to evaluate. It may also require much experience and training to know what to look for. The scenario is similar in *e. g.* many cases of medical diagnosis, where some information, perhaps the results of certain lab tests, must be acquired manually at a cost that can be measured in both monetary terms and in terms of a patient's

well-being.

To deal with these issues, we have constructed an incremental diagnosis system that is trained from empirical data. During a diagnosis session it evaluates the available information, calculates what additional information that would be most helpful for the diagnose, and asks the user to acquire that information. The process is repeated until a reliable diagnose is obtained. This system both significantly speeds up the diagnosis and represents knowledge of best practice in a structured manner. Since knowledge about how to perform efficient diagnostics of this kind of systems often is acquired in time by people working with it, the diagnose system is potentially very helpful for preserving diagnostics capabilities in spite of personell turnover and to increase the diagnostic capabilities of novice users [Martinez-Bejar *et al.*, 1999].

We will here present a system for incremental diagnosis where the issues above are addressed. It was originally created for diagnosis of faults in a variety of technical equipment used by the armed forces.

Practical Incremental Diagnosis

Using computers to support the diagnostic process is one of the classical applications of artificial intelligence. The methods developed so far are usually expert-systems related and rule-based [Heckerman *et al.*, 1992a; Veneris *et al.*, 2002], neural network related [Holst and Lansner, 1993; Stensmo, 1995; Wichert, 2005], or based on probabilistic methods [Heckerman *et al.*, 1992b; Kappen *et al.*, 2001; PROMEDAS, 2002]. The rule-based systems are in essence implementations of diagnostic protocols, specified in a large part manually through expert knowledge and to a lesser degree by learning from examples.

The approach works well in many diagnostic situations, especially in areas where expert knowledge is easily extracted and where the system does not need to adapt to new classification examples. However, rule based systems often suffer from a very rigid decision structure, where questions have to be asked in a certain order reflecting the internal representation. This might be very problematic in practise, where the order of which data can be retrieved is often arbitrary. Rule-based systems also suffer from the complexity of the rules that need to be implemented, resulting in a high number of conditions that need to be described, and from problems in dealing with uncertain sources of evidence. These issues can be solved, at least to a certain degree, by basing the diagnosis system on a probabilistic model.

Creating a useful probabilistic diagnosis system may not be overly complicated, but there are a few considerations and requirements worth keeping in mind. First, while being robust to erroneous and uncertain inputs, it is important that the number of questions necessary to reach a classification is minimised. In practise, this demand must usually be formulated somewhat differently in that we actually want to minimise the total *cost* of acquiring the information necessary to reach a classification, not just the number of questions.

No matter what methods are used in the diagnosis system, we have to decide

whether to base the system on available expert knowledge, on examples from earlier diagnosis situations using a more data-driven approach, or on both. If we want to be able to update the system as new examples are classified, a data-driven approach is preferable. However, as we mentioned earlier, quite often there are no examples of earlier diagnoses available when a new system is taken into use. This means that we have no choice but to try to incorporate some expert knowledge into the system so that it is at least somewhat useful at its introduction.

Although it is quite possible to encode expert knowledge into the structure of a probabilistic model, such as a *Bayesian Belief network* [Pearl, 1988; Jensen, 1996], we will use a slightly different approach by representing this expert knowledge as a special form of examples. We will refer to a regular example of a diagnosis, *i. e.* the known features and the correct diagnose from a practical diagnosis situation, as a *case*. The special form of examples for representing expert knowledge, *prototypes*, can be seen as generalisations from typical cases [Schmidt and Gierl, 1997]. Each prototype represents a *typical* input vector for a certain diagnose. A number of prototypes can be used to efficiently encode prior knowledge about a system. However, as cases and prototypes do have rather different interpretations, the diagnostic system must be able to account for this difference.

The approach also allows us to let the diagnostic system act as a persistent repository of knowledge that can continuously incorporate information from new classification cases, which may be difficult in rule-based systems. As the system gathers more information, some of the acquired cases may also be generalised by an expert into prototypical data, which could be highly beneficial for both classification performance and system understanding.

Another very practical requirement on the diagnostic system is that the probabilistic model in many cases must be able to handle both continuous and discrete inputs. This complicates matters somewhat, as we will see in later in the description of the models, especially if there are constraints on the time the system can use to calculate which unknown attribute to ask a question on. We also have to consider the fact that some attributes are grouped, in the sense that they are all acquired at the same time, by the same measurement. This complicates question generation somewhat, but can be taken care of within the model.

5.3 Probabilistic Methods for Incremental Diagnosis

As an alternative to rigid decision tree models for incremental diagnosis, we can use a probabilistic model [Stensmo *et al.*, 1991; Wiegerinck *et al.*, 1999; Kappen *et al.*, 2001]. Instead of modelling the decision tree directly, we model the relations between input data and the diagnosis by describing them in a probability model. Let us start again from the Naïve Bayesian classifier described in chapter 3. The probability of a class Z, where each class represents a certain diagnose, given a set

of inputs $\boldsymbol{X} = X_1, X_2, \ldots, X_n$ can be written as

$$p(Z|\boldsymbol{X}) = \frac{p(Z)p(\boldsymbol{X}|Z)}{p(\boldsymbol{X})} \tag{5.1}$$

where $p(\boldsymbol{X})$ fills the role of a normalization constant. Depending on the form of the conditional distributions, estimation of these distributions and calculation of the above expression can be very difficult in practise. A very common simplifying assumption is to assume conditional independence between all input attributes $\boldsymbol{X}$ given the class attribute Z. The complete distribution can then be described as a product of the probabilities of the individual attributes,

$$p(Z|\boldsymbol{X}) \propto p(Z)p(\boldsymbol{X}|Z) = p(Z)\prod_{i=1}^{n} p(X_i|Z) \tag{5.2}$$

The distribution for each attribute given a specific class, $P(X_i|Z)$ is significantly easier to estimate, due to the relatively low number of free parameters. This model is usually referred to as a *Naïve Bayesian classifier* [Good, 1950]. It often gives surprisingly good classification results although the independence assumption is usually not fulfilled, and by its simplicity it is very suitable as a starting point for statistical incremental diagnosis systems.

Creating an incremental diagnosis system based on a statistical model such as the Naïve Bayesian Classifier is relatively straightforward. Essentially, we would like to perform three operations on the model. The first is to find the class distribution if we know the values of a number of input attributes. This being an inherent property of a classification model, we will assume we can perform this. Second, we would like to calculate how much each unknown input attribute is likely to contribute to the classification given a number of known attributes. Finally, we would also like to be able to estimate how much each known input attribute contributed to a classification to provide feedback to the user. First, let us have a look at how this can be formulated without considering the details of the statistical model.

We will approach the incremental diagnosis problem as the task of reducing the uncertainty of the diagnosis. To determine the probable impact of gaining information about a currently unknown attribute, we can calculate the *expected reduction in entropy* in the class distribution if we learn the value of the attribute. This is a rather common approach, used *e. g.* in [Kappen, 2002] and [Rish *et al.*, 2005] and is the traditional approach used within our research group. If Z is the class distribution, $\boldsymbol{x} = \{X_1 = x_1, \ldots, X_n = x_n\}$ the already known input attributes, and $Y \in \boldsymbol{X}$ the unknown attribute we want to calculate the impact of, we can write this *entropy gain* $G_H(Y)$ as

$$G_H(Y) = H(Z|\boldsymbol{x}) - E_Y[H(Z|\boldsymbol{x}, Y)] \tag{5.3}$$

where $H(X)$ denotes the entropy [Shannon, 1948] of a stochastic variable X and E_Y the expectation with regards to Y. This expression is guaranteed to be equal

to or larger than zero since conditioning on the average reduces entropy. If the unknown attribute Y is discrete, the expression becomes

$$G_H(Y) = H(Z|\boldsymbol{x}) - \sum_{y_j \in Y} p(Y = y_j|\boldsymbol{x}) H(Z|\boldsymbol{x}, Y = y_j) \tag{5.4}$$

There is no requirement that Y must be discrete. However, calculation of the expectation in equation 5.3 may require numerical calculation of a rather complicated integral. The expression can in some situations be estimated with limited computational effort. If not, as long we can draw random numbers from $p(Y|\boldsymbol{x})$, we can always resort to calculating the expression through using a Monte-Carlo approach, where we draw N samples y_i from $p(Y|\boldsymbol{x})$ and approximate the entropy gain by

$$G_H(Y) = H(Z|\boldsymbol{x}) - \frac{\sum_{j=1}^{N} p(Y = y_j|\boldsymbol{x}) H(Z|\boldsymbol{x}, Y = y_j)}{\sum_{j=0}^{N} p(Y = y_j|\boldsymbol{x})} \tag{5.5}$$

This expression usually converges quite fast, but may still require prohibitively extensive computational resources. Also note that for none of these expressions there is in fact any restriction on the class attribute Z to be discrete as long as we are able to effectively calculate the entropy of the distribution.

Accounting for the costs associated with performing the queries is very straightforward as long as they are all measured at the same scale. If the cost of acquiring an unknown attribute is $C(Y)$, we can calculate the cost weighted entropy gain G_{H_C} as

$$G_{H_C} = G_H(Y)/C(Y) \tag{5.6}$$

This can then be used to rank which attributes to query instead of the entropy gain. The expression is similar, but not equal to what is often referred to as the *value of information*, usually defined as the expected reduction in cost compared with making a diagnose without the information [Howard, 1966].

Also note that in practical applications we may need to consider attributes to be grouped, in the sense that their values are acquired together. As an example, this could be a number of values that are provided by a time-consuming lab test. In this case, the cost of acquiring all these values must be weighed against the expected reduction in entropy of acquiring *all* the attributes,

$$G_H(Y) = H(Z|\boldsymbol{x}) - E_{\boldsymbol{Y}}[H(Z|\boldsymbol{x}, \boldsymbol{Y})] \tag{5.7}$$

where $\boldsymbol{Y}$ represents these grouped unknown attributes. Calculation of this expression may however be very time consuming in the case $\boldsymbol{Y}$ contains many attributes, as we need to evaluate the expectation over all combinations of outcomes of the variables in $\boldsymbol{Y}$.

To provide an estimate for the *explanation value* of a certain known attribute, *i. e.* how much knowing the value of this attribute contributes to the classification,

we can calculate the difference in entropy of the posterior for class Z when the attribute is known and when it is not. If Y is the attribute we would like to calculate the explanation value for and $\boldsymbol{X}$ all other known attributes, we define this explanation value $E_H(Y)$ as

$$E_H(Y) = H(Z|\boldsymbol{x}) - H(Z|\boldsymbol{x}, Y) \tag{5.8}$$

This expression does not however reflect the contribution to the classification of a certain attribute if there are dependencies between input attributes.

The user may also want to know how much each known attribute contributed to the certainty of the class having a specific outcome. This can be approximated as

$$E_p(Y) = |p(Z = z|\boldsymbol{x}) - p(Z = z|\boldsymbol{x}, Y)| \tag{5.9}$$

that is the absolute value of the change in probability of class z knowing attribute Y or not.

It is worth noting that although the above expressions are conceptually simple, their computation can get intractable for some statistical models [Kappen, 2002].

5.4 Incremental Diagnosis with Limited Historical Data

Let us now construct a statistical model for incremental diagnosis that can effectively make use of limited historical data. Although using a statistical model, we are here going to take an approach that is not that different from an instance based learner, where each new pattern is compared to all examples in the available historical data to find the most similar ones [Cover, 1968]. Assuming a clear distinction between prototypical and case data, the former describing a typical, distilled instance of a class, and the latter an example of a diagnostic case, we let each prototype form the basis of a component in a *mixture* [McLachlan and Peel, 2000], a weighted sum, of Naive Bayes classifiers. That is, if there are m prototypes in the data, we create a mixture of m simpler classifiers, whose classifications are then combined (see figure 5.1). This also allows us to effectively describe classes that manifest themselves in rather different manners in data, as long as there are available prototypes that describe these different situations.

This structure can easily be described as a hierarchical graph model (see chapter 3), represented as a mixture of mixtures of graphical models. More formally, if $\boldsymbol{X} = X_1, \ldots, X_n$ denote the input attributes, the posterior class distribution of Z can be written as

$$p(Z|\boldsymbol{X}) \propto \sum_{z \in Z} p(Z = z) \sum_{k \in P_z} \left(\pi_{z,k} \prod_{i=1}^{n} p_{z,k}(X_i|Z = z) \right) \tag{5.10}$$

where P_z is the set of prototypes that are labelled with class z, and $\pi_{z,k}$ denotes the mixing proportion corresponding to prototype k for class z. To arrive at the actual distribution $p(Z|\boldsymbol{X})$, we only need to normalise over Z in equation 5.10.

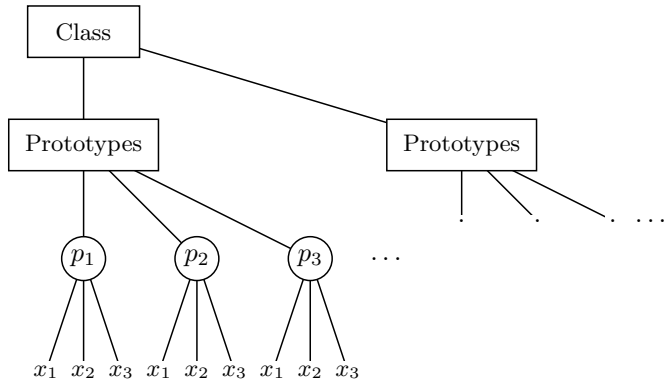

Figure 5.1: An overview of the model structure. Each class is associated with a number of prototypes, $p_1, p_2, \ldots$. The prototypes all use the same input attributes $x_1, x_2, \ldots$, each prototype corresponding to a single prototypical entry in the historical data.

Equation 5.10 relates to four distributions. First, $P(Z|\boldsymbol{X})$, the posterior class distribution conditioned on $\boldsymbol{X}$ *and* the prototype and case data. Secondly, $P(Z = z)$, the class distribution conditioned on prototype and case data. Furthermore, $\pi_{z,k}$ is an assumed distribution over prototypes k belonging to class z, and finally $p_{z,k}(X_i|Z = z)$ is the distribution of X_i for prototype k of class z and again conditional on prototype and case data. A simplistic attitude to handling equation 5.10 is to *estimate* the quantities of the right hand side of 5.10 from prototype and case data and to plug the estimates into equation 5.10. A more principled approach is to make a Bayesian inference of the class distribution, which leads to a posterior distribution for the class distribution. Since the class distribution will ultimately be used for expected utility decision making, all probabilities should be estimated using means over the posterior. For estimates of 5.10, this means that the right hand side quantities are plugged in to give the mean estimate of $P(Z|\boldsymbol{X})$. For the discrete distributions (all distributions on the right hand side of 5.10 except $p_{z,k}(X_i|Z = z)$ for continuous attributes X_i) the mean estimator is the Laplace estimator. Laplace's estimator is used to get the mean estimate of a discrete probability distribution from occurrence counts and a uniform prior over the set $L_d = \{\boldsymbol{x} : \sum_j x_j = 1, x_i \geq 0, i = 1, \ldots, d\}$ of possible discrete probabilities. The estimator for x_i with counts n_i is $x_i = \frac{n_i+1}{n+d}$, *i. e.* the estimate is the relative frequency after adding one to each occurrance count.

For the case of a continuous attribute, we will assume that the conditional

distribution $p_{z,k}(X_i|Z = z)$ is Gaussian,

$$p_{z,k}(X_i|Z = z) = \frac{1}{\sqrt{(2\pi)\Sigma_i}} \exp\left(-\frac{(x_i - \mu_i)^2}{2\Sigma_i}\right) \tag{5.11}$$

where μ is the mean and Σ the variance of the distribution. We will here take the standard estimates of mean and covariance and plug into the density function formula for the normal distribution. Technically, the mean estimator in this case is not a normal but a t-distribution [Gelman *et al.*, 2003], since we average over a long tailed distribution for σ^2. However, for reasonably large samples, the difference is small. Also, we will assume that we in practise usually will have an estimate of the variance or the range of a variable based on domain knowledge. Note that setting an estimated distribution to a normal distribution also means that an outlier can dominate the naïve Bayes classification (because of the exponential decay of the likelihood), so it is important to detect outliers and handle them appropriately.

Note that the use of Gaussian distributions can be avoided by discretising the continuous attributes, which could be beneficial for computational complexity as evaluating expression 5.3 is easier for discrete attributes. If an attribute is known to operate within certain distinct regions, *e. g.* a temperature that can be considered low, normal, or high, discretising it to these classes might be useful.

Let us now have a closer look at the parameter estimates. We start with the mixing proportions $\pi_{z,k}$. These essentially represent the relative importance of each prototype within a class, where $\sum_{k \in P_z} \pi_{z,k} = 1$. They are set manually, and should roughly correspond to the proportion of actual diagnosis cases that the prototype usually represents within the class. Lack of knowledge of this corresponds to a zero sample and with Laplace's estimator a uniform distribution.

To arrive at an effective estimation procedure for the prior class distribution $p(Z)$ and the conditional attribute distributions $p_{z,k}(X_i|Z = z)$, we are going to use a Bayesian approach as decribed above. We will however not present complete derivations of the expressions used here, but would like to refer the reader to section 3.9. To be able to properly incorporate the prototypical data with the actual cases, we are going to use a hierarchy of priors that will be estimated in turn before we arrive at the final distribution. First, we will assume a non-informative uniform prior, that is used to estimate a distribution from the prototype data. This distribution in turn will, in a sense, be used as a prior for the estimation of the actual distribution. Let us start with describing how this is performed for the discrete distributions $p(Z)$ and $p_{z,k}(X_i|Z = z)$.

In the case of the discrete class distribution Z, the parameter p_z representing the probability of each outcome z can be estimated through

$$p_z = \frac{c_z^c + \eta\left(\frac{c_z^p + \theta}{C^p + \theta|Z|}\right)}{C^c + \eta} \tag{5.12}$$

where c_z^p and c_z^c are the number of outcomes z in the prototype and case data respectively, and C^p and C^c the total number of examples in each of these data sets.

$|Z|$ denotes the total number of outcomes in Z. The parameter η represents how much trust we put in the prior distribution estimated from the prototypes, while θ can be interpreted as a kind of smoothing parameter for the prior distribution.

To be able to properly incorporate the prototypical data with the actual cases, each conditional $p_{k,z}(X_i|Z=z)$ for a certain class and prototype is estimated from corresponding cases using a prior distribution, in turn estimated from the specific prototype. This estimation from a specific prototype uses a prior estimated from all prototypical data, which in turn uses a non-informative uniform prior. Let us walk through these estimations step by step, starting with the estimation of a prior distribution based on all prototypes.

In the case $p_{k,z}(X_i|Z=z)$ is discrete, the parameter p_x^p representing the probability of each outcome X is estimated through

$$p_x^p = \frac{c_x^p + 1}{C^p + |X|} \tag{5.13}$$

where c_x^p are the number of outcomes x in the data, C^p the total number of prototypes, and $|X|$ the number of outcomes in X. In the continuous case, we estimate the parameters of a Gaussian as

$$\mu^p = \sum_{\gamma \in D_p} x^{(\gamma)}/C^p \tag{5.14}$$

$$\Sigma^p = \sum_{\gamma \in D_p} (x^{(\gamma)} - \mu^p)^2/(C^p - 1) \tag{5.15}$$

where D_p represents the set of prototypes and $x^{(\gamma)}$ one prototype value.

Now, before we incorporate the case data, we will estimate the distribution $p_{z,k}^0(X_i|Z=z)$, which represents the final parameter estimation in case there is no case data available, and otherwise forms the basis of the parameter estimation from case data. In the discrete case the parameter p_x^0 is estimated as

$$p_x^0 = \frac{v_x^k + p_x^p/|D_p|}{1 + 1/|D_p|} \tag{5.16}$$

where v_x^k is an indicator variable that is one if the outcome is equal to x and zero otherwise, and $|D_p|$ the number of prototypes in the data. In the continuous case, the parameters are estimated as

$$\mu^0 = \frac{x_k + \mu^p/|D_p|}{1 + 1/|D_p|} \tag{5.17}$$

$$\Sigma^0 = \Sigma^p \tag{5.18}$$

where x_k represents the value of X in prototype k. The mean of the distributions varies with the prototype, while the variance is the same for all of them.

To make efficient use of the case data, we want to use it for estimation in such a way that each prototype distribution is updated in proportion to how likely it is that each specific case was generated from it. In detail, the importance of each case for a certain prototype k is weighted by the probability that the case was generated from it by calculating the likelihood that each case was generated from prototype k and normalising over the patterns within the class,

$$p(k|\boldsymbol{x}) = \frac{\pi_k \prod_{i=1}^{n} p_{z,k}(x_i|Z=z)}{\sum_j \pi_j \prod_{i=1}^{n} p_{z,k}(x_j|Z=z)} \tag{5.19}$$

where $\boldsymbol{x}$ denotes a case pattern. The procedure can be viewed as performing one step of the *Expectation Maximisation* algorithm for the mixture. The final expressions for the parameters for a certain prototype k in the discrete case then become

$$p_x = \frac{\sum_{\gamma \in D_c} p(k|\boldsymbol{x}^{(\gamma)}) v_x^{(\gamma)} + \psi p_x^0}{\sum_{\gamma \in D_c} p(k|\boldsymbol{x}^{(\gamma)}) + \psi} \tag{5.20}$$

where $v_x^{(\gamma)}$ is an indicator variable that is one if $X = x$ and zero otherwise. D_c denotes the set of cases and $\boldsymbol{x}^{(\gamma)}$ case γ in this set. In the continuous case, the parameters are estimated through

$$c_k = \sum_{\gamma \in D_c} p(k|\boldsymbol{x}^{(\gamma)}) \tag{5.21}$$

$$\mu^c = \frac{\sum_{\gamma \in D_c} p(k|\boldsymbol{x}^{(\gamma)}) x^{(\gamma)}}{c_k} \tag{5.22}$$

$$\Sigma^c = \frac{\sum_{\gamma \in D_c} p(k|\boldsymbol{x}^{(\gamma)}) (x^{(\gamma)} - \mu^c)^2}{c_k} \tag{5.23}$$

$$\mu = \frac{c_k \mu^c + \psi \mu^p}{c_k + \psi} \tag{5.24}$$

$$\Sigma = \frac{c_k \Sigma^c + \psi \Sigma^p + \frac{c_k \psi}{c_k + \psi} (\mu^c - \mu^p)^2}{c_k + \alpha} \tag{5.25}$$

where $x^{(\gamma)}$ is case value γ, and μ and Σ the final parameter estimates. In both the discrete and continuous case, the parameter ψ represents how much trust we put in the prototypes compared to the cases and can be expected to be set to different values for different applications.

In summary, the estimations above follow a Bayesian scheme, but some approximations were made that can lead to under-estimating the uncertainty.

Calculating the entropy gain as given by expression 5.3 and 5.4 is straightforward, as $p(Y = y_j|\boldsymbol{X})$ can be directly calculated from

$$p(Y|\boldsymbol{X}) \propto \sum_{z \in Z} p(Z = z|\boldsymbol{X}) \sum_{k \in P_z} \pi_k p_k(Z) p_k(Y|Z) \tag{5.26}$$

where $p(Z|\boldsymbol{X})$ is calculated from equation 5.10 using the known input attributes. If there are continuous attributes represented by Gaussians present, the situation is a little different, as we have to integrate over the attribute in question instead of calculating the sum in equation 5.4. This can be solved by using a Monte-Carlo approach, where a number of samples are generated from $p(Y = y_j|\boldsymbol{X})$ and used to calculate the expectation. Another solution that produces very accurate results is to sample a number of points from each prototype model, *e. g.* at equal intervals up to a number of standard deviations from the mean.

Experiments

To evaluate the performance of the model, we have used one synthetical and several real data sets containing both discrete and continuous attributes. Models were estimated from data, and tested using noisy versions of correct input patterns, both to evaluate the diagnostic performance and to find appropriate values for the significance parameters between prototypes and cases.

When a diagnosis is performed, unknown attributes are incrementally set based on the largest entropy gain. Thus, unknown attributes are set in the order in which their contribution to the final *hypothesis* is maximised. The order in which different attributes are set depends mainly on the significance ψ between prototypes and cases. In addition, the significance η of the class distribution may also influence the importance of specific attributes. Often only a few significant attributes need to be known in order to obtain a final hypothesis, while remaining attributes are redundant in the current context.

While attributes are not necessarily set strictly based on the entropy gain in a real-world diagnosis scenario, since other factors may influence the choice of attribute to set for the user, for testing purposes we will assume that they are. Also, in a real world diagnosis scenario it is not necessarily important to determine at what exact point in the answering sequence we should claim that we have a valid hypothesis about the class. The user can usually determine this reliably just by looking at a presentation of the class distribution, which also provides information on uncertainty and alternative hypothesis. However, being able to signal to the user that we have a valid hypothesis is naturally useful, and for testing absolutely necessary as we need to be able to automatically decide when we have a relevant classification.

A natural way of determining when we have reached a hypothesis is to see if one class has significantly higher probability than the other classes. Unfortunately, it is by no means easy to give a general expression for what constitutes a "significantly higher" probability. The measure is also flawed in that it in practise often is impossible to find one class with significantly higher probability, *e. g.* if two classes are expressed in almost exactly the same way in the data. Instead, we can rely on the calculated entropy gain for the unknown attributes. To reduce the number of attributes that do not significantly contribute to the actual hypothesis, we have introduced a thresholding parameter ξ. When the entropy gain for all unknown

attributes is smaller than ξ, the hypothesis is considered final. This is an unbiased measure of the validity of the hypothesis that does not suffer from the problems discussed above.

In order to test the general performance of our model, we have performed several series of experiments using data sets that contain discrete or continuous attributes. In all of the experiments, we have measured the number of questions needed to achieve a final hypothesis and the number of correctly diagnosed samples. In order to reduce computational demands, the results of the diagnostic performance are based on the average of no more than 10 runs to obtain statistical significance in all of the experiments. Further, in some cases two (or more) classes in the datasets contain nearly similar sets of known attributes, leading to ambiguous diagnoses. In order to avoid such diagnoses, we allowed for the diagnostic model to use a first and second trial, in which the first and second diagnoses of highest confidence were tested against the target diagnosis.

Experiments on Discrete Datasets

We have here tested the diagnostic performance of our model when varying ξ, ψ and η. For this purpose, three original datasets with discrete attributes were used. The first dataset contains 32 classes of animals described by a total of 82 attributes. The second dataset contains 31 classes of common mechanical faults that appears in military terrain vehicles, described by 39 attributes. The third dataset contains 18 classes of common mechanical faults that appears in military tanks, described by 83 attributes. While the first data set is artificial, the other data sets represent real data on which the model will be used in practise.

These original datasets are completely clean, in the sense that they do not contain any unknown attributes, or, as the data from a testing viewpoint also must be interpreted as the true solutions, any incorrect attributes. In order to evaluate our complete diagnostic model, synthetic prototypes and cases were extracted from the original datasets. Prototypes and cases are here defined to contain both known and unknown values. In addition, cases can also contain incorrect attribute values.

For each class, two prototypical samples were extracted directly from the original datasets. Based on the complementary prior for each attribute, the value was set as unknown or to the correct known value. The prototypes were then used in the diagnostic model to create five synthetic cases for each class. Based on the largest entropy gain, specific attributes were set using known values from the original dataset. As mentioned earlier, cases can contain incorrect attribute values. Therefore, noise was introduced through setting 20% of randomly selected attributes in each case to be set to a random value based on the prior for the specific attribute.

Initially, two series of baseline experiments for each dataset were performed while varying ξ. In both experiments, each one of the datasets were used as prototypes. In the first series of experiments, the same dataset was used to directly set the attribute values without noise. In the second series of experiments, a subset of

the attributes (20%) were randomly selected to contain noise as described, while remaining attribute values were set directly from the original dataset.

ξ	Correct (%) Trial #1	Correct (%) Trial #2	Known attributes
	Without noise		
4.7×10^{0}	3.13	3.13	0
4.7×10^{-1}	100.00	0.00	5.03
4.7×10^{-2}	100.00	0.00	5.69
4.7×10^{-3}	100.00	0.00	6.34
4.7×10^{-4}	100.00	0.00	7.19
4.7×10^{-5}	100.00	0.00	7.91
4.7×10^{-6}	100.00	0.00	8.66
	With 20% noise		
4.7×10^{0}	3.13	3.13	0
4.7×10^{-1}	56.56	8.75	5.08
4.7×10^{-2}	95.00	1.88	8.50
4.7×10^{-3}	96.25	1.56	9.96
4.7×10^{-4}	97.50	0.94	10.69
4.7×10^{-5}	98.44	1.25	13.03
4.7×10^{-6}	99.06	0.94	13.81

Table 5.1: The animal dataset. The table shows the diagnostic performance achieved in one of two trials and the number of attributes needed to obtain a final hypothesis when varying ξ.

When performing diagnoses in which all necessary attribute values are correct, we observe from the results in table 5.1–5.3 that the number of attributes needed to achieve a correct diagnosis is close to $\log_2 n$, where n is the total number of attributes in the dataset. When 20% noise is used, we observe that ξ can be used to improve the diagnostic performance on the first trial. Thus, the need for explicit inconsistency checks for incorrect attribute values is reduced, since our diagnostic model is able to find the correct diagnosis, using only a few more known attributes. Further, if the diagnostic model is allowed to use a second trial, we observe that it is possible to achieve a diagnostic performance of more than 95% in all of the baseline experiments.

Similar to the baseline experiments, two additional series of experiments for each one of the datasets were performed using extracted prototypes and cases. In these experiments, diagnoses were performed when varying the parameters ψ and η while keeping $\xi = 4.7 \times 10^{-5}$ fixed. In figure 5.2–5.4, the diagnostic performance on each dataset and the number of known attributes needed for a final hypothesis is shown. We observe that the significance of prototypes in general needs to be large in order to obtain a diagnostic performance closer to the baseline experiments.

ξ	Correct (%) Trial #1	Correct (%) Trial #2	Known attributes
	Without noise		
4.7×10^{0}	5.56	5.56	0.00
4.7×10^{-1}	100.00	0.0	4.28
4.7×10^{-2}	100.00	0.0	5.17
4.7×10^{-3}	100.00	0.0	6.11
4.7×10^{-4}	100.00	0.0	7.39
4.7×10^{-5}	100.00	0.0	8.17
4.7×10^{-6}	100.00	0.0	9.39
	With 20% noise		
4.7×10^{0}	5.56	5.56	0.00
4.7×10^{-1}	65.56	7.22	4.28
4.7×10^{-2}	93.33	2.78	7.46
4.7×10^{-3}	97.78	1.11	8.89
4.7×10^{-4}	98.33	1.11	10.46
4.7×10^{-5}	99.44	0.56	11.39
4.7×10^{-6}	98.33	1.11	12.90

Table 5.2: The tank dataset. The table shows the diagnostic performance achieved in one of two trials and the number of attributes needed to obtain a final hypothesis when varying ξ.

Further, we observe that the number of known attributes needed to obtain a final hypothesis decreases when indicate that the number of known attributes needed to obtain a final hypothesis decreases when ψ is large. Since all classes are uniformly distributed, the overall diagnostic performance does not change significantly when η is varied.

Experiments on Continuous data

We have also performed experiments using a dataset that contains both continuous and discrete attributes. Samples in the dataset were extracted from the evaporation stage of a paper mill. The dataset contains 11 classes with 6 samples each, specified by 106 continuous attributes and 4 discrete attributes. Since prototypical samples were unavailable, we used a synthetic set of prototypical samples with all attribute values set to unknown, while using the whole dataset as cases in the model. A fixed subset of the dataset was used to set attribute values when diagnostics was performed.

Three series of experiments were performed in which we varied ψ and η. Since the classes are uniformly distributed we did not perform any experiments varying ξ. In the first series of experiments, the attribute values were set directly without

ξ	Correct (%) Trial #1	Correct (%) Trial #2	Known attributes
	Without noise		
4.7×10^{0}	3.23	3.23	0.0
4.7×10^{-1}	87.10	12.90	4.81
4.7×10^{-2}	87.10	12.90	5.55
4.7×10^{-3}	87.10	12.90	6.10
4.7×10^{-4}	87.10	12.90	7.58
4.7×10^{-5}	87.10	12.90	8.39
4.7×10^{-6}	87.10	12.90	9.81
	With 20% noise		
4.7×10^{0}	3.23	3.23	0.0
4.7×10^{-1}	62.58	15.16	5.67
4.7×10^{-2}	75.48	14.19	7.72
4.7×10^{-3}	80.32	13.23	8.42
4.7×10^{-4}	78.07	15.48	10.64
4.7×10^{-5}	79.03	17.74	11.95
4.7×10^{-6}	80.97	14.84	13.77

Table 5.3: Terrain vehicle dataset. The table shows the diagnostic performance achieved in one of two trials and the number of attributes needed to obtain a final hypothesis when varying ξ.

induced noise. In the two remaining series of experiments, we used two different approaches to induce 20% noise in order to investigate how noise-sensitive the model is when using continuous data. Thus, in the second series of experiments, the probability of noise was based on a Gaussian prior distribution estimated from each attribute, calculated from the sample mean and standard deviation measured within each class. In the third series of experiments, the Gaussian prior for each attribute was estimated from the sample mean and standard deviation, measured on the whole dataset.

We observe from the first series of experiments that the diagnostic performance mainly is dependent on the value of ψ (figure 5.5a). Naturally a small value on ψ leads to a higher classification rate since the model is based only on real cases. We observe that the total average classification rate can be improved combining the result of two trials (figure 5.5a, b). In figure 5.5b the classification rates varies compared to the results in figure 5.5b. It is likely that the classification rates in the second trial is more susceptible to the parameter settings and possibly to the Monte-Carlo sampling step that is performed on continuous data. Further, our results indicate that the number of steps needed for a final hypothesis depends more on ψ than on η, specifically when ψ is very large or very small. For an average of 4.76 known attributes (taken over all classes and 10 runs) we obtain a

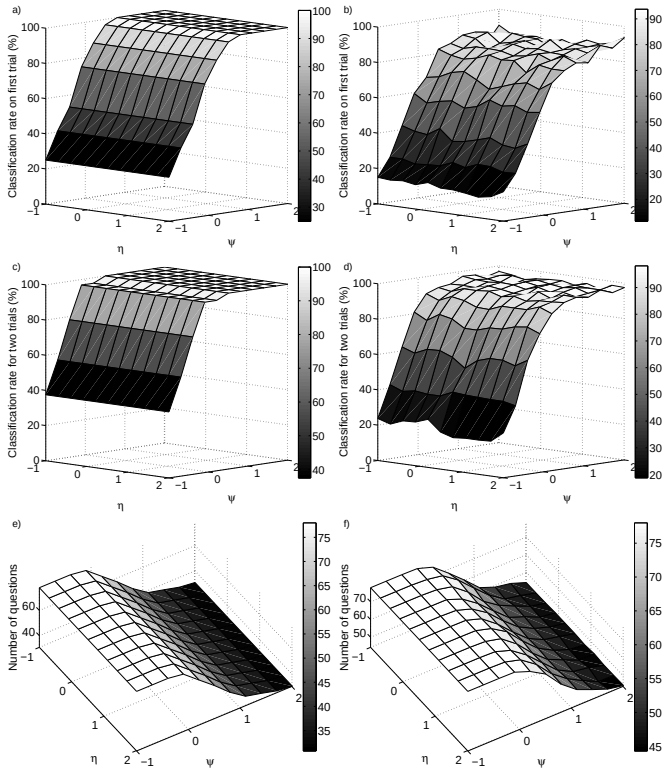

Figure 5.2: Diagnostic performance on the animal dataset when varying η and ψ. The figure shows a) the classification rate on the first trial, b) the classification rate on the first trial with 20% noise, c) the combined classification rate for two trials, d) the combined classification rate for two trials with 20% noise, e) the number of known attributes needed for a final hypothesis and f) the same as in e) but with 20% noise.

classification rate of 100% using $\eta = 0.01$ and $\psi = 0.001$, compared to 13.3 known attributes using $\psi = 10^{-5}$ (figure 5.5a,c). However, when ψ is set to a fixed medium value, we observe that the model produces nearly the same classification rate using fewer steps when increasing η (figure 5.5c). In this case increasing the value on ψ reduces the separability between classes. In effect, the significance of varying η is increased, such that it affects only the number of known attributes needed for a final hypothesis but not necessarily the classification rate. However, this only applies as long as ψ is set to a medium value. When ψ is sufficiently large the final hypotheses starts to repeatedly indicate only a limited (and possibly fixed) subset of the classes which reduces the classification rate and the significance of η. In this

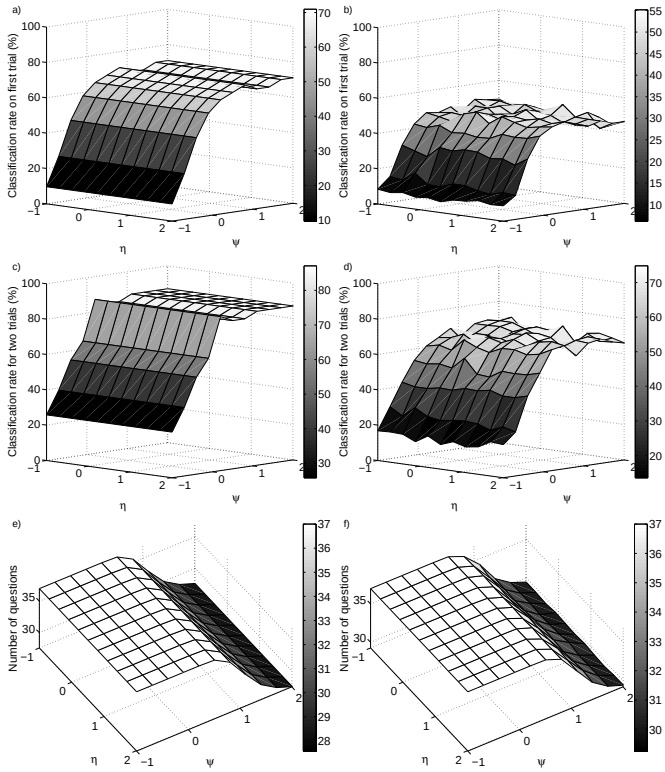

Figure 5.3: Diagnostic performance on the terrain vehicle dataset when varying η and ψ. The figure shows a) the classification rate on the first trial, b) the classification rate on the first trial with 20% noise, c) the combined classification rate for two trials, d) the combined classification rate for two trials with 20% noise, e) the number of known attributes needed for a final hypothesis and f) the same as in e) but with 20% noise.

case, for instance, the final hypotheses indicate only one certain class when ψ is sufficiently large, regardless the value of η. Conversely, when ψ is set to a very small value, class separability is increased and the significance of η is reduced.

In the second series of experiments, we observe that the diagnostic performance slightly decreases compared to the first series of experiments (figure 5.6a, b). Our results indicate that by inducing noise based on the prior Gaussian distribution for each attribute within each class, we can obtain satisfactory performance if ψ is set low. In the third series of experiments, we observe that the diagnostic performance is less robust to noise (figure 5.7a, b), compared to the results in the second series of experiments. For example, the number of correct diagnoses is 70.91% compared to

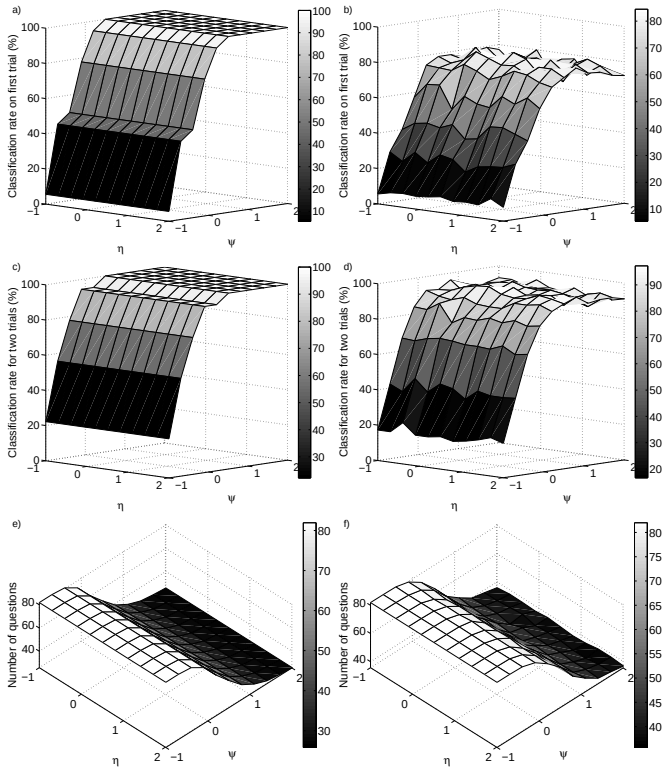

Figure 5.4: Diagnostic performance on the tank dataset when varying η and ψ. The figure shows a) the classification rate on the first trial, b) the classification rate on the first trial with 20% noise, c) the combined classification rate for two trials, d) the combined classification rate for two trials with 20% noise, e) the number of known attributes needed for a final hypothesis and f) the same as in e) but with 20% noise.

100% in the second series of experiments when $\psi = 10^{-5}$ and $\eta = 10^{-5}$. Further, we observe that the number of known attributes needed for a final hypothesis varies depending on how noise is induced (figure 5.5c, 5.6c and 5.7c).

Obtained classification rates indicate that the model is more sensitive when noise, based on the Gaussian prior distribution for each attribute estimated over the whole dataset, is induced. It should be noted that the attribute values have a large variance between different classes which therefore causes a lower diagnostic performance when noise is induced this way. We therefore conclude that the model is more robust for inexact attribute values as long as the values are set within the prior distribution of the class.

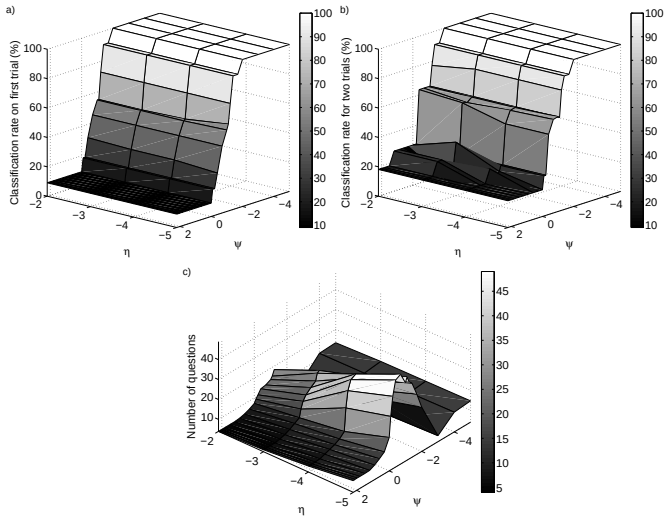

Figure 5.5: Diagnostic performance without noise on the evaporation dataset when varying η and ψ. The figure shows a) the classification rate on the first trial, b) the combined classification rate for two trials and c) the number of known attributes needed for a final hypothesis.

5.5 Anomalies, Inconsistencies, and Settings

In practise, incremental diagnosis poses a few more problems than those we have discussed so far. The perhaps primary one relates to the fact that users do make mistakes or acquire the wrong information. We must expect erroneous values as inputs to the classifier. However, in an interactive system we have a chance to counter these errors as it is possible to ask the user to specify a doubtful attribute again. This reduces the risk that the diagnosis will be wrong or uncertain, and increases the fault-tolerance of the system.

Basically, what we need is a mechanism for detecting *inconsistencies* in the attribute values, which in this context means combinations of inputs which are very unlikely (but not necessarily impossible). In essence, we would like to check if any of the known input values are very unlikely given everything else we know about the situation. This can be directly formulated as calculating the likelihood that the conditional distribution of each attribute y given all other known attributes $\boldsymbol{x}$ generated the specific outcome,

$$\lambda_m^y = p(Y = y|\boldsymbol{x}, M) \tag{5.27}$$

where $\boldsymbol{x} = X_1 = x_1, \ldots, X_n = x_n$ are the known attributes and M the model

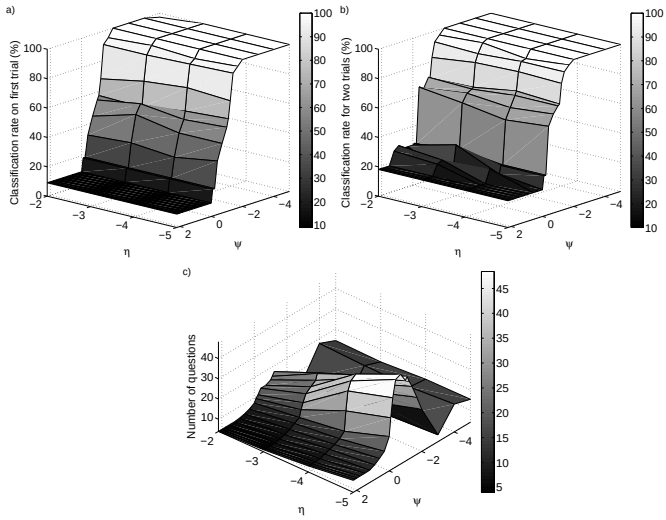

Figure 5.6: Diagnostic performance on the evaporation dataset for varying η and ψ, with 20% noise based on the prior distribution for each attribute within each class. The figure shows a) the classification rate on the first trial, b) the combined classification rate for two trials and c) the number of known attributes needed for a final hypothesis.

parameters. In our case, the expression above can be written as

$$\lambda_m^y = \sum_{z \in Z} p(z|\boldsymbol{x}) \sum_{k \in P_z} \left(\pi_{z,k} \prod_{i=1}^{n} p_{z,k}(Y = y|Z = z) \right) \tag{5.28}$$

where $p(z|\boldsymbol{x})$ is calculated from equation 5.10. To find a suitable limit κ_m on λ_m^y for when to alert the user, we just need to define below what level of probability a value should be considered a possible inconsistency. The exact value certainly depends on the type of application, but we have consistently been using $\kappa_m = 0.05$ throughout our tests with good results.

A related problem occurs if we would like to be able not only to adapt the model to new cases, but, if the current input vector is inconsistent with the current model, suggest to the user that a new prototype should be created based on these inputs. Generally, we would like to decide whether or not the current input vector of known attributes should be regarded as normal or not given the current model. This can be done by calculating the likelihood λ_p that this input vector was generated by the model, or

$$\lambda_p = p(\boldsymbol{x}|M) \tag{5.29}$$

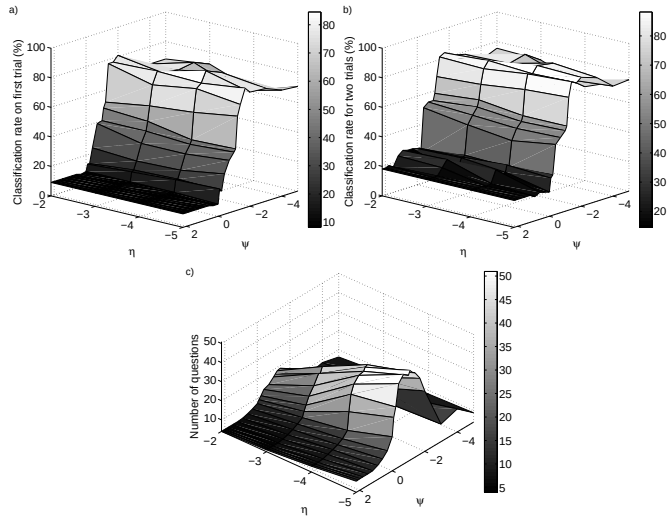

Figure 5.7: Diagnostic performance on the evaporation dataset for varying η and ψ, with 20% noise based on the prior distribution for each attribute over the whole dataset. The figure shows a) the classification rate on the first trial, b) the combined classification rate for two trials and c) the number of known attributes needed for a final hypothesis.

where $\boldsymbol{x} = X_1 = x_1, \ldots, X_n = x_n$ are the known attributes and M the model parameters, and can in our case be found directly from equation 5.10.

As we can see, calculating the likelihood of a pattern is straightforward, but below what level of likelihood should we start considering the pattern to be sufficiently abnormal? Again, let us start with the whole input vector. Assuming all prototypes should be interpreted as normal, a reasonable limit on the likelihood κ_p can be estimated as a fraction τ_p of the minimum of the likelihoods that each prototype was generated by the model,

$$\kappa_p = \tau_p \min_i p(\boldsymbol{x}_i|M) \tag{5.30}$$

where $\boldsymbol{x}_i$ are the attribute values of prototype i that are known in the current input pattern. A suitable value of τ_p then depends on how early we would like to trigger an inconsistency warning.

A problem that relates to the inconsistency checking described above is that of managing settings and prerequisites for a system. A system to be diagnosed often requires a number of prerequisites to be able to diagnose a class at all. A very simple example of this could be the requirement of having the ignition on in a car to be able to diagnose a fault in its electrical system. Most of the time,

these requirements are much more complicated than this, and to enumerate all combinations of settings that constitute a possible user error is impossible simply because the vast majority of settings are incorrect. Only a few combinations are in fact valid settings. If this is the case, it is not practical to formulate this as a diagnosis problem, as we cannot easily find representative examples of all types of errors.

Instead, let us consider the possibility of specifying all combinations that represent *correct* settings. If this is indeed possible, as it often is in practical applications, we can estimate a statistical model based on data that representing different kinds of correct prerequisites. We can then detect *anomalies* in the input vector by estimating the likelihood of a new pattern being generated by the model in the exact same manner as for detecting inconsistencies in expression 5.29.

Although used differently, the demands on the model used for this type of anomaly detection are very similar to those of the classification model we have used for diagnosis. We would like to combine prior system knowledge, which can be expressed as prototypical data, with actual usage cases in order to adapt the model to current circumstances. We will therefore here use exactly the same kind of model, the difference being that the classes will not represent a certain kind of *condition* to be diagnosed, but rather a certain kind of *scenario* for which the settings are valid. As before, one class or scenario can have many typical expressions, and thus many prototypes.

This allows us to calculate reasonable limits for when the likelihood of a certain pattern should be considered abnormal just like we did when we needed to detect inconsistencies through equation 5.30. If a pattern of settings is considered abnormal, we can naturally also use expression 5.27 to determine which settings attributes are most likely to be wrong.

Just like the diagnosis situation, we may not actually know all relevant information initially. We would therefore potentially like to perform the anomaly detection incrementally. However, calculating and presenting the expected reduction in entropy in the class distribution for each unknown attribute provides little information. We do not want to determine what class (condition) the pattern indicates, but rather to determine whether the pattern seems to belong to any of the classes at all. Therefore, we would like to rank our unknown attributes according to the *expected reduction in likelihood* to the whole pattern if we learn the value of one attribute Y,

$$G_L(Y) = p(\boldsymbol{x}|M) - E_Y[p(\boldsymbol{x}, Y|M)] \tag{5.31}$$

where $\boldsymbol{X}$ are the already known attributes, and E_Y the expectation according to Y. If Y is discrete, this can be written as

$$G_L(Y) = p(\boldsymbol{x}|M) - \sum_{y_j \in Y} p(Y = y_j|\boldsymbol{x})p(\boldsymbol{x}, Y = y_j) \tag{5.32}$$

As before, there is no requirement that Y must be discrete, but the calculation of the expectation in equation 5.31 may be complicated if it is not. The expected

reductions in likelihoods are presented to the user, who inputs new information accordingly.

This allows us to efficiently perform incremental anomaly detection, but there are certain practical limits to what we can detect. Without other attributes than those representing the settings or prerequisites, it is impossible for us to qualitatively separate two correct settings from each other. The specified settings may be acceptable, but unsuitable for the specific kind of diagnose we would like to perform. Introducing input attributes that in some way represent what kind of diagnosis the user wants to perform could provide a solution, as would the possibility of the user actually specifying the class attribute of the model, representing the current diagnosis scenario. Note also that if this class is unknown, propagating its conditional distribution to the actual diagnosis model could improve diagnosis performance since it provides an idea of what the current scenario is. This of course depends on the availability of specified scenario attributes in the training data for the diagnosis model.

Experiments

We have investigated the diagnostic performance when using inconsistency checks on all the discrete datasets and on the continuous dataset. For this purpose, we performed two series of experiments in which the degree of noise was gradually increased.

In the first series of experiments, we measured the diagnostic performance without using inconsistency checks to obtain baseline performance. The baseline results were the compared to the diagnostic performance in the second series of experiments in which inconsistency checking was used. In both series of experiments the noise level was set to 20%, 35% and 50%. All of the experiments on both types of datasets were repeated 10 times in order to obtain statistical significance on the diagnostic performance.

Discrete datasets

In the discrete case, we performed experiments using only prototypes in the model. Noise was induced by setting a random selection of attributes to the incorrect value based on the prior taken over the whole dataset, as described. The likelihood limit for all attributes in the discrete case was set to $\kappa_m = e^{-3}$. The prototype significance was set fixed to $\psi = 1.0$ while the thresholding parameter varied as $\xi = \{4.7 \times 10^{-6}, \ldots, 4.7 \times 10^{-1}\}$.

We observe that the diagnostic performance improves by the use of inconsistency checking (figure 5.8–5.10). For example, the percentage of correctly diagnosed samples on the terrain-vehicle dataset increases by 9.03 percentage points for 50% noise compared to the corresponding baseline result when $\xi = 4.7e{-}6$ (table 5.5). In table 5.4–5.6, we observe that the number of anomaly checks in general matches the number of incorrect attribute values. In addition, our results indicate the number

of known attributes needed to obtain a final hypothesis (including the 'extra' inconsistency questions) can be reduced, compared to when not using inconsistency checking (figure 5.8–5.10).

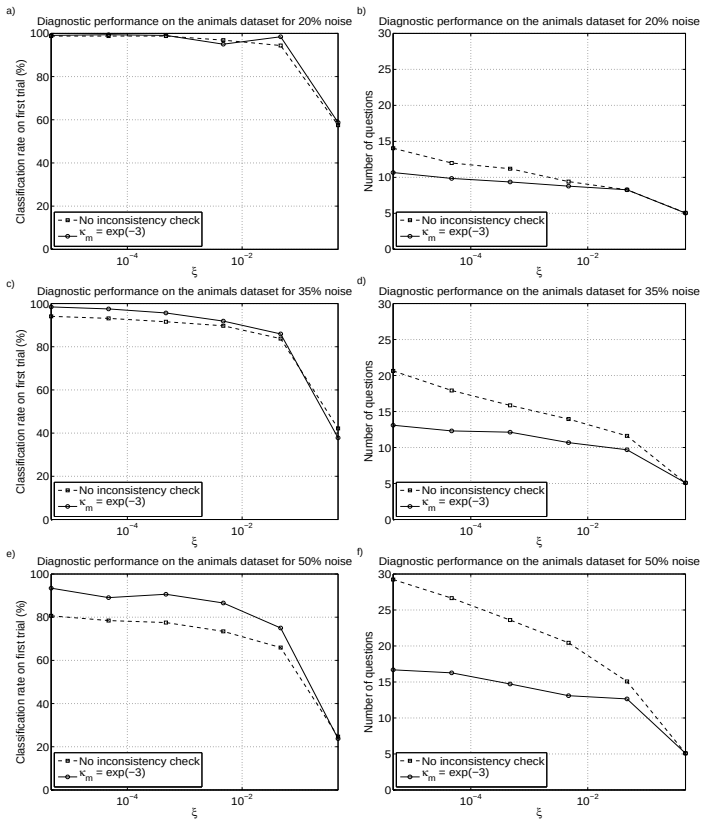

Figure 5.8: Diagnostic performance on the animal dataset using inconsistency checking for varying ψ and ξ: a,c,e) the total diagnostic performance from one trial: b,d,f) the number of known attributes needed for a final hypothesis.

The Continuous Dataset

In the continuous case, the samples in the dataset were used as cases in the model, whereas a fixed subset of samples was used to set attribute values, as earlier described. Noise was induced based on the Gaussian prior estimated from the sample mean and standard deviation of each attribute taken over the whole dataset.

Setting an inconsistency limit κ_m for an attribute is more complicated in the continuous case compared to the discrete case. For a discrete distribution, we can

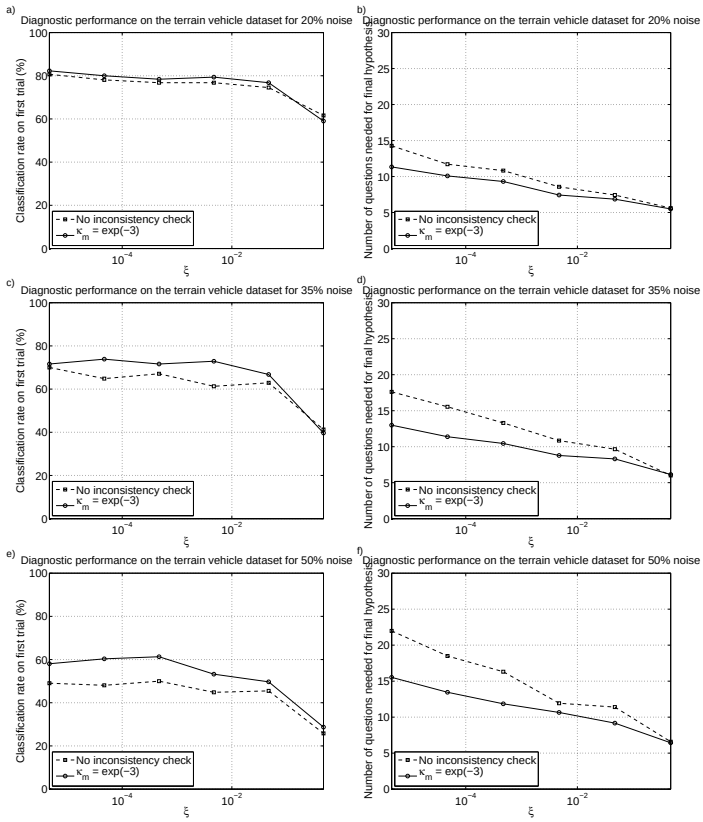

Figure 5.9: Diagnostic performance on the terrain vehicle dataset using inconsistency checking for varying ψ and ξ: a,c,e) the total diagnostic performance from one trial: b,d,f) the number of known attributes needed for a final hypothesis.

easily find the actual probability of a certain outcome. In the continuous case, we can find the probability density at a certain value. However, this is not a proper probability, as it in a sense depends on the scale of the attribute, possibly assuming values larger than one. What we can do, however, is to find the value of the probability density for which a certain fraction of the total density is below, and use this as our threshold. If we can find this value, we only need to decide which fraction to consider anomolous, similar to the discrete case.

Here, we will set the limit κ_m on the probability density individually for each continuous attribute to the probability density for which a certain fraction of the total probability mass is lower in the Gaussian prior estimated from the complete

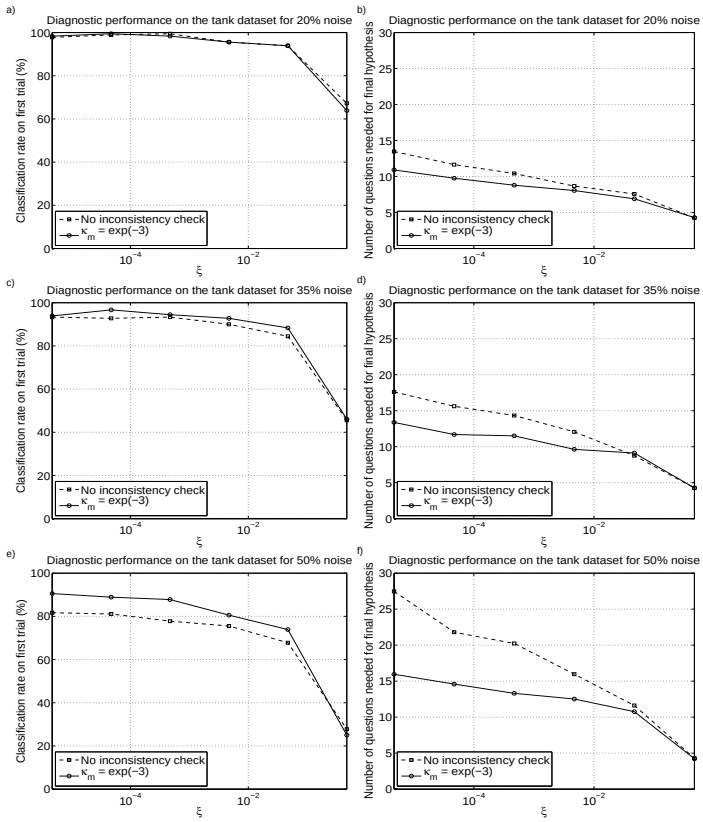

Figure 5.10: Diagnostic performance on the tank dataset using inconsistency checking for varying ψ and ξ: a,c,e) the total diagnostic performance from one trial: b,d,f) the number of known attributes needed for a final hypothesis.

set of samples. This can be calculated from

$$\kappa_m = \frac{e^{-\mathrm{erf}^{-1}(c-1)^2}}{\sqrt{2\pi\sigma_m^2}} \tag{5.33}$$

where σ_m^2 is the estimated variance of attribute m, c the desired fraction, and erf^{-1} is the inverse error function. This will hopefully provide us with a rough approximatition of an appropriate value for the limit, but not necessarily a very good one.

Ultimately, we would like to set the limit based on the conditional distribution of the attribute given the already known values, as the scale of this distribution might change drastically when further attributes become known. However, as this

Noise (%)	Correct (%) Trial #1	Known attributes	Incorrect attributes	Anomaly checks
	Without anomaly checking			
20	98.75	14.04	1.23	0
35	94.06	20.64	2.59	0
50	80.62	29.23	4.71	0
	With anomaly checking			
20	99.06	10.67	0.91	1.31
35	98.44	13.09	1.78	2.49
50	93.44	16.69	3.16	4.13

Table 5.4: Diagnostic performance on the animal dataset for $\psi = 1.0$ and $\xi = 4.7 \times 10^{-6}$ when using inconsistency checks.

Noise (%)	Correct (%) Trial #1	Known attributes	Incorrect attributes	Anomaly checks
	Without anomaly checking			
20	80.65	14.27	1.09	0
35	70.00	17.61	2.13	0
50	49.03	21.98	3.45	0
	With anomaly checking			
20	82.26	11.34	0.88	1.20
35	71.61	12.99	1.77	2.20
50	58.06	15.53	2.91	3.01

Table 5.5: Diagnostic performance on the terrain vehicle dataset for $\psi = 1.0$ and $\xi = 4.7 \times 10^{-6}$ when using inconsistency checks.

distribution is expressed as a mixture of Gaussians, calculating the proper value of the limit analytically becomes impossible. Therefore, we will settle for the approximation described above in our tests.

In the tests, the inconsistency limit κ_m was based on a fraction $c = e{-}3 \approx 0.05$ of the probability density function for each continuous attribute. For the discrete attributes the inconsistency limit was strictly set to e^{-3}. Thus, the inconsistency limit varied between different attributes. Here, the prototype significance was set to $\psi = 10^{-5}$ and the thresholding parameter to $\xi = 10^{-5}$.

We observe that the diagnostic performance on continuous data can be improved by the use of inconsistency checking. For example, we see that the diagnostic performance improves 13.64 percentage points when 20% noise is induced compared

Noise (%)	Correct (%) Trial #1	Known attributes	Incorrect attributes	Anomaly checks
	Without anomaly checking			
20	97.78	13.48	1.19	0
35	93.33	17.61	2.44	0
50	81.67	27.47	4.97	0
	With anomaly checking			
20	98.33	10.92	0.97	1.16
35	93.89	13.37	2.07	2.45
50	90.56	15.96	2.97	3.32

Table 5.6: Diagnostic performance on the tank dataset for $\psi = 1.0$ and $\xi = 4.7 \times 10^{-6}$ when using inconsistency checks.

Noise (%)	Correct (%) Trial #1	Known attributes	Incorrect attributes	Anomaly checks
	Without anomaly checking			
20	65.45	10.39	2.15	0
35	49.09	8.41	2.95	0
50	36.36	7.36	3.69	0
	With anomaly checking			
20	79.09	10.13	2.06	1.03
35	63.64	8.96	3.03	1.10
50	40.00	6.60	3.26	1.15

Table 5.7: Diagnostic performance on the evaporation dataset for $\psi = 10^{-5}$ and $\xi = 10^{-5}$ when using inconsistency checks.

to the baseline experiments (table 5.7). However, the number of questions needed to obtain a final hypothesis does not seem to be significantly affected in general. We also observe that the number of anomaly checks does not fully match the number of incorrect attributes as well as in the discrete case. This is partly due to the fact that detecting inconsistencies in the continuous case is a more delicate matter than in the discrete case, and partly that the continuous data set contain more ambiguities than the tested discrete data sets.

Since we have observed that the average percentage of correctly diagnosed samples may vary about 5% between different runs of the experiments, further repetitions of the experiments may be needed in order to obtain statistically reliable results. Observations during the experiments also indicate that the inconsistency

limits may have to be re-calculated using another value on κ_m when the degree of noise is changed. In spite of this, our results indicate that our approach of inconsistency checking could be used to obtain higher diagnostic performance on continuous data.

5.6 Corrective Measures

After an incremental diagnosis session has been taken as far as practically possible, we often would like to propose a suitable corrective measure to the user, a way to manage the diagnosed problem. This can be rather simple, as one diagnose often corresponds directly to one corrective measure: For one diagnosed problem, there is only one solution. As long as there is a description of the corrective measure associated with each possible diagnosis in the database, suggesting corrective measures is trivial.

If this is not the case, as *e. g.* in many situations within medical diagnosis, we can still provide the user with valuable information by presenting similar earlier case descriptions which can suggest a suitable treatment. What constitutes a similar case though is not necessarily obvious, but we can at least differentiate between a number of general alternatives.

Ultimately, we would like to find cases that are likely to have *similar corrective measures*. Unless the diagnosis corresponds directly to a corrective measure as discussed earlier, we will assume that we do not have sufficient information to determine this in the database. A good approximation of this, which is indeed often the correct one if a diagnosis are directly connected to a corrective measure, is to determine how similar two cases are by determining how *similar their class distributions* are given the known inputs. This can be done by calculating the *Kullback Leibler distance* [Kullback and Leibler, 1951] between the class distributions,

$$D(p_1||p_2^i) = \sum_{z \in Z} p_1(z) \log \frac{p_1(z)}{p_2^i(z)} \tag{5.34}$$

where $p_1(z)$ denotes the class distribution of the current diagnosis and $p_2(z)$ the class distribution of the i:th case, using the same known input variables as for the current diagnosis. Although the measure is highly useful for ranking cases by similarity to the current diagnosis, it is relatively safe to assume that distances closer to or above $\log |Z|$, where $|Z|$ denotes the number of outcomes in Z, show that the two distributions have very little similarity and can safely be assumed not to be related.

It is not unlikely that each of the prototypes of the database can be associated with a distinct corrective measure, even if the class itself cannot. In this situation, and indeed many others were knowing what prototype corresponds the best to the current diagnose situation can be beneficial to the user, we can for the model used easily calculate the probability distribution over the prototypes given the known

inputs as

$$p(k|\boldsymbol{x}) = \sum_{z \in Z} p(Z = z)\pi_{z,k} \prod_{i=1}^{n} p_{z,k}(x_i|Z = z) \tag{5.35}$$

This allows us to present the user the prototypes the current inputs are most likely to be drawn from, along with the probability. It can only point to relevant prototypes while ignoring other data in the form of cases, but the computational complexity is significantly lower than calculating the distance in class distribution as above for each available case. If suitable, though, we can always calculate the distance to each case through an expression similar to equation 5.34, but using the prototype distribution instead of the class distribution.

Lastly, there is always an opportunity to calculate the similarity between cases by determining the *distance in input space* between two patterns. Here, however, our statistical model is of limited assistance when defining a suitable distance measure, and we refer the reader to the case based reasoning literature instead.

5.7 Designing Incremental Diagnosis Systems

Let us now review and summarise the methods discussed earlier by describing a more complete and practically useful incremental diagnosis system. In a wider setting, we would like to not only offer a diagnosis, but also suggest corrective action based on this diagnosis and collect information from new diagnostics cases. Figure 5.11 shows a simplified view of the complete process.

First, *diagnosis* is performed incrementally until we have arrived at an acceptable hypothesis. This can be determined by the system as in the experiments of section 5.4, or by the user. The diagnostic procedure basically involves two parallel tasks: one to perform the actual diagnosis and one to determine whether the prerequisites for performing this diagnosis are fulfilled or not.

In the diagnostics case, the user first enters new information based on the presented entropy gains. The known input attributes are then checked for inconsistencies as described in section 5.5, and these inconsistencies signalled to the user. If relevant for the specific application, the prerequisites or settings are checked so that they are likely to be correct. If not, information about what settings that are most probably incorrectly set should be shown to the user. If the values of the settings must be acquired at a cost, the user could be re-directed to perform this anomaly detection incrementally, as discussed in section 5.5.

After verifying the settings, a new classification is performed where the entropy gain is calculated for the unknown attributes and the conditional class distribution presented to the user. If we by now have arrived at an acceptable hypothesis, we can move on to assist the user in finding suitable *corrective measures*.

As we have described earlier, suggesting corrective measures is often rather straightforward as there is only one or a few suitable actions to take for a certain diagnose. However, if this is not the case, we can assist the user by finding the relevant prototypical and case data as in section 5.6.

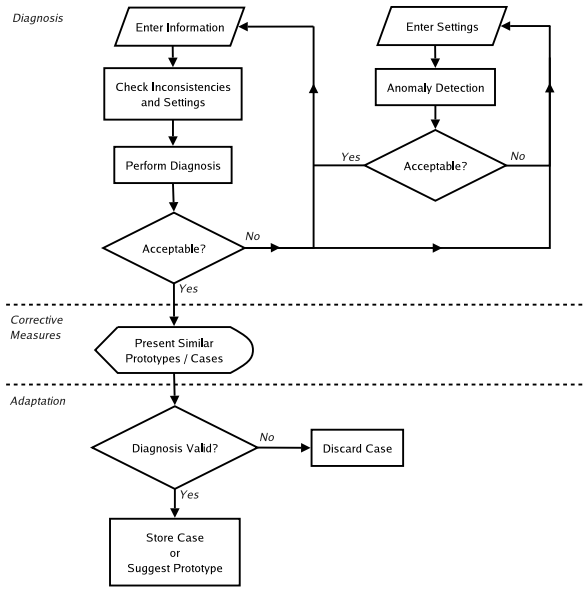

Figure 5.11: A simplified view of the complete diagnostics process.

Finally, we usually want the system to *adapt* to new cases. Before entering the case into the database and adapting the model to it, the case should be validated by the user so that the diagnosis specified is actually the correct one in order to avoid reinforcement of erroneous classifications by the model. If the actual diagnosis is new and not represented in earlier data, the case should instead be refined into a new prototype to give the system an opportunity to perform a correct classification in the future. A new prototype should potentially also be suggested if the case seem to represent a new representation of the class that is very dissimilar from earlier examples. This can be done by using the methods described in section 5.6.

The user interface to the diagnosis system should reflect the possibility of answering questions in any order or at any point changing the answer of an earlier question. This is a significant leap in flexibility compared to many expert system approaches, and to design an interface that only allows the user to answer the currently most important question as in a traditional system defeats some of the advantages of the approach. Therefore, it is more beneficial to *e. g.* present a list of unknown attributes, ordered by their respective entropy gain, that can be set in arbitrary order along with earlier set attributes.

Figure 5.12 shows an early prototype interface intended for use in the field by armed forces to diagnose technical equipment. It allows for selecting different types of technical equipment, and through that change the current database for the statistical model. The diagnosis can be focused on certain sections of the equipment,

selected in a tree structure at the top of the application. The table to the left shows all possible questions and their current entropy is gain shown as a bar in the table. The top right table shows all possible diagnosis, sorted in descending order by their current probability which is also shown as a bar in the table. The frames below show instructions on how to repair the selected condition and what resources are necessary for the operation, and on the bottom of the window there is a button which adds the current case to the database.

This interface is likely to change, but can still serve as a useful example of what an interface to the diagnosis system may look like.

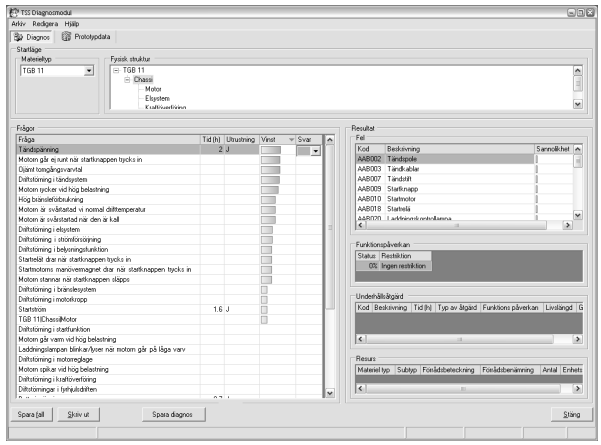

Figure 5.12: A prototype interface for diagnosing technical materiel in the armed forces.

5.8 Discussion

We have shown a flexible, robust and efficient incremental diagnosis system trained from empirical data. Since the system can make use of both expert knowledge and examples of actual diagnosis cases, it provides a powerful alternative to rule based systems. Also, because the presented statistical model does not work with explicit rules, it does not have the same problem as rule based systems with inconsistencies in the data. Exceptions and ambiguities are handled in a natural way.

A highly desirable feature of the system for practical use is the fact that questions can be answered in arbitrary order. It is possible to avoid answering a certain question, answering another one instead, or even change a previous reply at any point. Initially providing some information also makes the diagnosis start from there, without having to traverse questions in a predefined order. This is highly

usable if there *e. g.* are some automatic readings available or some system checks have already been performed.

Also, in spite of not using a rule based approach, we can still get simple explanations from the system, in terms of the primary causes for the systems' conclusion. Since it is difficult to build user confidence without providing explanations of the diagnosis systems conclusions, this property is critical in practical use. The fact that the system at every point provides a distribution over the classes also helps build user confidence and makes the diagnosis task easier.

Inconsistencies in the inputs can be reliably detected, giving the user an opportunity to correct faulty inputs to the system. The same mechanisms can also be used to determine whether the prerequisites to performing the classification are fulfilled or not. This also makes it possible to identify user errors, something that is indeed very common in practical situations.

Being somewhat similar to an instance based learner, the model does suffer from some of the same type of problems. Most importantly, as the number of prototypes increase in the system, so does the complexity of performing classification and by consequence the complexity of calculating the information gain for unknown attributes. This effect is not necessarily present in other models such as just using one naive Bayesian classifier, for which the classification complexity would remain constant with the number of prototypes. However, adding new cases to the database, which in time most likely would constitute the bulk of the historical data, imposes no increase in computational complexity.

The fact that the system can both easily learn from each new diagnosed case and determine whether the case may be a suitable candidate for refinement into a prototype, are also very important in many situations. The properties of the diagnosed system is usually not completely known when it is taken into use, making the possibility of collecting and incorporating new information continuously absolutely crucial. This way, the diagnosis system also provides a means for structuring and storing knowledge that might otherwise be lost with the departure of an experienced operator.

Future work includes testing the system on more practical cases to determining the performance of the model. These practical experiences would hopefully also allow us to deduct heuristics for adjusting the balance of significance between prototypical data and actual case data so that the system works well in all applications.

Acknowledgements

The author would like to thank the Swedish procurement agency, FMV, for providing insight and understanding on how to perform and support diagnosis of technical systems. We would also like to thank Palmarium AB for their work on the user interfaces to the diagnosis system for the Swedish armed forces. This work derives from and is inspired by the work on incremental diagnosis performed within the SANS/CBN group at KTH by, among others, Anders Lansner and Anders Holst.

Chapter 6

Creating Applications for Practical Data Analysis

6.1 Introduction

In this chapter, we will discuss how to create machine learning and data analysis applications in practise. We introduce a new methodological approach to data preparation that does not suffer from the limitations of earlier proposals when it comes to processing industrial data. We show how the methodology can be reduced to a small set of primitive operations, which allow us to introduce the concept of a "scripting language" that can manage the iterative nature of the process. We also show how both the data preparation and management operations as well as a complete modelling environment based on the Hierarchical Graph Mixtures can be implemented into a fast and portable library for the creation and deployment of applications. Coupled to this, we demonstrate a high-level interactive environment where these applications can be easily created.

6.2 The Data Analysis Process

Although differing in goals, challenges and approaches, most data analysis projects follow a common general process consisting of problem specification, data preparation, modelling, and perhaps deployment into existing systems. It is tempting to consider the modelling phase, where specific models of data are chosen and specified, as the most important and time consuming part of a project. However, the success of the project in practise depends to a perhaps higher degree on the understanding and detailed specification of the problem along with the preparation and representation of data.

To help data analysis projects succeed and to minimise the effort with which the results are achieved, we therefore need to address the whole data analysis process. There is not only a need of better data analysis or machine learning algorithms, but

also of better methodology, common practise, and tools. Naturally, it is difficult to produce consistent methodologies for procedures that may be very different from one case to the next, but it still serves a number of purposes. First, the findings can be used to determine common practise and guidelines for data analysis tasks. Hopefully these guidelines can serve as a structured approach to at least a majority of common analysis tasks, but will at the very least be useful as a collection of advice regarding methods and common pitfalls. Second, and more important, is the fact that studying the data analysis work-flow and creating methodologies for it is absolutely vital if we want to create tools that support the analysis process.

For the sake of the discussion, let us divide the whole data analysis process into five main phases: *Problem Understanding*, *Data Preparation*, *Modelling*, *Validation*, and *Deployment* (see figure 6.1). These phases are somewhat overlapping and the whole process usually contains some degree of iteration, but let us start with an overview of each phase in turn.

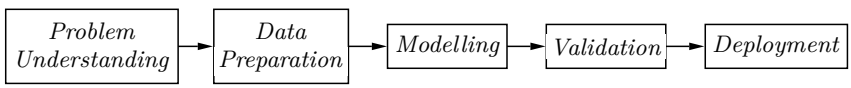

Figure 6.1: A simplified overview of the data analysis process.

Problem Understanding The first part of any data analysis task involves getting acquainted with the problem and the application area. The analyst quite often has no previous experience with the area, something that can make it very difficult to fully comprehend the problem and what is actually needed in terms of data and algorithms to solve it. The problem specification can also be rather vague, perhaps formulated as a need to understand why a certain phenomenon, which in itself can be described in very loose terms, manifests itself.

The work of the analyst during this phase is therefore dominated by understanding the specifics of the application area and formulating the problem in a more tangible form. This formulation usually recalls a machine learning or data analysis task such as prediction, clustering, or dependency derivation. It is of course difficult to predict all steps that need to be taken during analysis and modelling, but a good understanding of the ultimate goal of the exercise, even though it may turn out to be impossible to achieve, is very important as this influences all consecutive steps of the process.

When the application area and problem are at least to some degree understood, suitable data need to be located and extracted. In practise there is often not that

much choice at this stage: One or a few databases are available, and additional data are very difficult to extract. If data supporting the intended application or problem formulation cannot be found in the already available databases it is simply not possible to come up with a solution. It is not uncommon that a data analysis project ends at this early stage because of lack of data, although it is much more common that this happens later on in the project when this becomes more apparent.

Supporting this first stage of data analysis with tools is difficult due to the fundamental requirement, at least in the foreseeable future, on a very high degree of human participation. It may be possible to create efficient protocols and methodologies that can speed up the process considerably, but this is very much out of the scope of this text. We will therefore focus very little of our effort to speed up the data analysis process with respect to this initial phase.

Data Preparation Once the possible data sources have been identified and relevant data extracted, the data needs to be prepared and structured to fit the needs of the subsequent analysis and modelling. The available data sets rarely have a structure suitable for these tasks, as it usually was not collected with these applications in mind. On top of that, the data from real-world databases are often notoriously dirty. It usually contains high degrees of noise, coding artifacts, missing data and redundant or irrelevant attributes, all of which needs to be dealt with in order to progress with the actual analysis and modelling of the data.

This phase of the analysis project is often very time consuming and involves a high degree of exploratory work. The analyst is usually not too familiar with how the data are represented and structured, and a lot of time can be spent initially on visualising and understanding the collected data. After this, the data needs to be cleaned from errors, different data bases need to be merged, and then transformed into suitable data sets for analysis and validation. The process usually requires interaction from experts that can answer questions about how the data were generated and represented in the data bases.

Although this process is exploratory by nature, it can be structured in such a way that it becomes more efficient and possible to support by appropriate tools. In section 6.3 we will discuss how this can be done in practise and present a structured approach to the data preparation phase.

Modelling The modelling phase mainly revolves around finding suitable parameterisations for the models we want to estimate from data. This can be very difficult or impossible for some applications, but is often surprisingly straightforward in many practical cases once the original data have been transformed in a suitable way. However, whether this transformation is regarded to be a part of the modelling or the earlier preparation phase is of course just a matter of semantics.

The procedure of finding a suitable parameterisation is highly dependant on the type or family of models used. For example, specifying the number of hidden layers and number of units in a neural network trained with back-propagation is

very different from specifying the graphical structure of a Bayesian belief network. Still, given a family of parameterisations, a lot can usually be done to simplify the procedure of specifying parameters in the model *e. g.* through appropriate visualisation.

Validation The validation phase is concerned with evaluating the performance of the created models, such as the estimated generalisation error of a predictor or a classifier, or the validity of a detected dependency or cluster. It is by nature highly connected to the modelling phase, since a multitude of models or hypotheses usually are evaluated and modified iteratively during a project. It is here given the status of a separate entity in the work-flow largely because of its importance: without proper and rigorous validation, we cannot be sure that the application is likely to perform well in practise or that the conclusions we have drawn based on available data have any real significance.

Although there are several useful methods for estimating generalisation performance and hypothesis validity originating in *e. g.* statistics and machine learning, proper validation is still often very difficult in practise. Data often contain dependencies, *e. g.* temporal, that mean that the instances in the data cannot be considered to be independent. Care has to be taken to assure that what we actually do estimate is the proper generalisation performance, and not the performance on something that closely resemble the training data.

Deployment When a model has been validated and can be expected to perform well in practise, it can finally be developed into the intended application. This can mean the creation of a separate software application or integration into a larger framework, such as a company's production environment. Naturally, not all results of a data analysis project can or are intended to be used this way. They may mainly provide increased understanding about the processes generating the data and the data itself, *e. g.* by confirming or refuting hypothesis about the data, rather than providing *e. g.* a prediction or classification that can be used as new data arrive or the situation changes.

Deployment can easily become the most time consuming part of a data analysis project. Not only must the chosen model and perhaps estimation procedures be correctly implemented, but the same applies to the data preparation and transformation procedures that the model relies on. The procedure can be simplified greatly by providing the opportunity to export programs on some form from the data analysis and modelling tools used to arrive at the final model. However, this relies on the possibility to specify standard interfaces for interacting with such an exported component.

6.3 Data Preparation and Understanding

Data available for analysis is often stored in a wide variety of formats and in several different repositories, and efficient methodologies and tools for data preparation and merging are critical. Experience shows that data analysis projects often spend the major part of their effort on these tasks, leaving little room for model development and generating applications. In this section we will try to identify and classify the needs and individual steps in data preparation. The focus is on industrial data, but the methods should be applicable to most areas without major changes.

We will here suggest a methodology for data preparation suited for industrial data, based on experiences from analysing data in *e. g.* the pulp and paper industry, chemical production and steel manufacturing. In particular, the methodology takes into account the iterative nature of the data preparation process and attempts to structure and trace the alternating steps of data preparation and analysis.

Based on the methodology, a small set of generic operations is identified such that all transformations in data preparation are special cases of these operations. Finally, a proof of concept data preparation system implementing the proposed operations and a scripting facility to support the iterations in the methodology is presented along with a discussion of necessary and desirable properties of such a tool [Gillblad *et al.*, 2005b, 2005a].

Introduction

Data analysis and data mining are traditionally used to extract answers and useful patterns from large amounts of data. In classical statistics, much of the effort goes into deciding what data are necessary, and into designing ways to collect that data. Data mining and knowledge discovery are different in that the analysis has to work with whatever data is available. These data will have been collected for various other purposes and are unlikely to fit the new needs perfectly. Data bases need to be merged, using different formats and ways of identifying items, where special and missing values will most certainly have been encoded differently.

The problem can become even harder when working with data gathered from industrial production. Such data are generally a mixture of logs from sensors, operations, events and transactions. Several completely different data sources, modes and encodings are not uncommon, making preparation and preprocessing of data essential before an often quite complex merging operation can be performed. Sensor data will reflect that sensors drift or deteriorate and are eventually replaced or serviced, often at irregular intervals. Anomalies showing in one part of the process may often be due to problems in earlier process steps for which data are sometimes not even available. Still, data analysis is often critical, *e. g.* when trying to increase the efficiency of the production process or finding explanations for the origin of phenomena that are not completely understood.

Data analysis uses an abundance of methods and techniques, but common to all is the need for relevant and well structured data. In practise, many data analysis

projects will spend most of their time collecting, sampling, correcting and compiling data. This is even more pronounced when working with data from industrial processes. The quality of the subsequent analysis is also highly dependent on the preparation of data and how the merging of the data sources is performed.

Background and Related Work

Data preparation usually refers to the complete process of constructing well structured data sets for modelling. This includes identification and *acquisition* of relevant data; *cleaning* data from *noise*, *errors* and *outliers*; *re-sampling*, *transposition* and other transformations; *dimensionality* or *volume reduction*; and *merging* several data sets into one. All these operations are normally performed in an exploratory and iterative manner. The complete process would benefit greatly from a more structured approach, taking particularities of the application area into account.

Several attempts have been made to describe the entire knowledge discovery and data analysis process [Fayyad *et al.*, 1996a; Clifton and Thuraisingham, 2001]. Of these, the CRISP-DM (CRoss-Industry Standard Process for Data Mining) [Reinartz *et al.*, 1998] initiative is perhaps the best known among recent proposals. Other examples include SEMMA, a methodology proposed by SAS Institute Inc. for use with their commercial analysis software [SAS Institute Inc., 1998], and the guidance for performing data mining given in textbooks such as [Adriaans and Zantinge, 1996; Berry and Linoff, 1997]. Most of these process models have roughly the same scope, covering the entire process from problem understanding to model deployment, through steps such as data preparation, modelling and evaluation. The methodology proposed here instead focuses on and details the early parts of the process, referred to in CRISP-DM as *data understanding* and *data preparation*. The guidelines given in [Reinartz *et al.*, 1998] for this phase are sensible and well chosen but are described on a level too general to be directly applicable in practise, at least for the specific task of preparing industrial data.

There have also been some more detailed descriptions of data preparation (*e. g.* [Pyle, 1999; Reinartz *et al.*, 1998; Lawrence, 1991]) that provide a good understanding of the problem and deliver plenty of tips and descriptions of common practise. However, there is still a lack of a more practical guide to what steps should be performed and when, or how to create an application that supports these steps. These descriptions are generally more focused on corporate databases and business related data than on industrial applications [1].

More recent work also advocate the need for efficient data preparation. Zhang and Yang [Zhang *et al.*, 2003] sum up future directions of data preparation as the construction of interactive and integrated data mining environments, establishing data preparation theories and creating efficient algorithms and systems for single and multiple data sources.

[1] Pyle [Pyle, 1999] *e. g.* clearly states that the book deals with corporate databases, while CRISP-DM [Reinartz *et al.*, 1998] hints at its business orientation by referring to the first stage of the data mining process as "Business Understanding".

The methodology that is discussed here takes this further with a focus on industrial data, basic methodology and the generic operations supporting the methodology. As far as we know, this is the first attempt to do so. We also emphasise the need for means to document, revoke, and reconstruct a sequence of steps in the data preparation process, and stress the importance of repair and interpretation, effective type analysis, and merging.

A Methodology for Data Preparation

The different tasks in the preparation of data generally include a number of steps, some of which may have to be iterated several times. It is very common that the need to refine the result of an earlier step becomes apparent only after the completion of a later step, and data preparation is often revisited quite far into the analysis and modelling process. The same applies to early representation choices that turn out to be inappropriate, and late introduction of innovative recoding that simplify the modelling and analysis.

To handle these problems, the methodology includes a number of distinct operations and explicit iterations. The methodology is intended to provide a structured context to support the data preparation process, and an overview is shown in figure 6.2. This section describes each step and indicates under what circumstances it will be necessary to back-track to an earlier step.

Problem Classification

Problem classification includes identification and classification of the type of data analysis task. As discussed before, typical tasks include *error* or *anomaly detection*; *fault diagnosis, state classification* or *prediction*; and *proposing corrective actions for errors*. The type of data analysis task entails different choices in encodings and models which will be relevant to later steps. It is usually highly beneficial to have as good an understanding as possible of the problem at this early stage, as this allows for better initial decisions on type determination and representation, reducing the risk of having to revisit and reformulate earlier steps of the process.

Selection of Data Sources

The collection of data typically involves extracting information from several data bases with different representations and sample rates. Common sources are logs for individual or related groups of sensors, target and measured values for certain parameters related to the type of product produced, and various quality measures. These sources are typically very heterogeneous and need to be individually preprocessed before being merged into a single data source on which the analysis task can be performed. Naturally, the availability and choice of data determine the type of analysis that will later be possible.

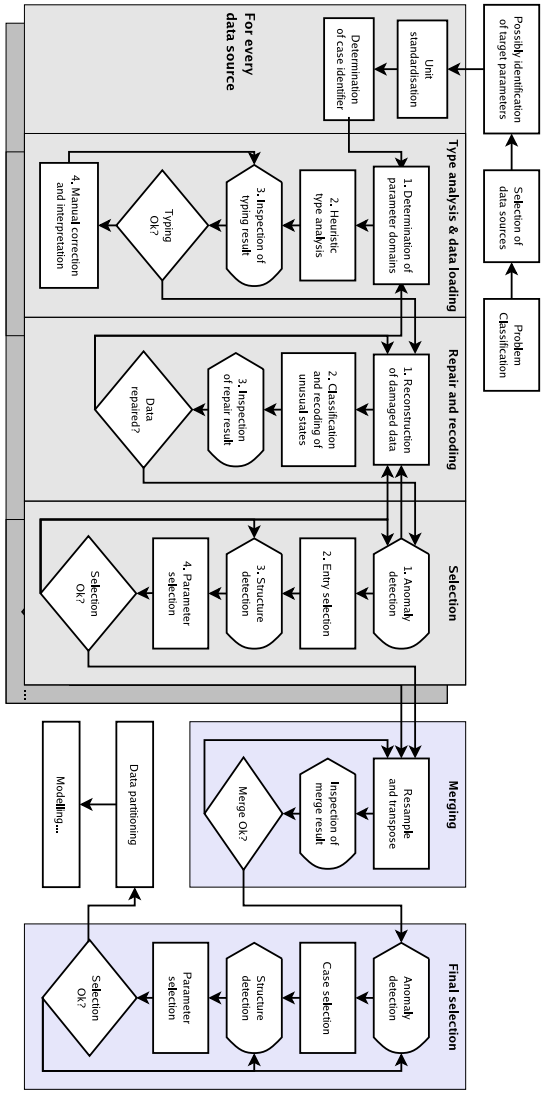

Figure 6.2: Methodology overview.

Identification of Target Parameters

For many, although not all data analysis tasks, it is necessary to find a subset of the parameters which uniquely identify the situations or conditions that the analysis should recognise or predict using the remaining parameters. These are the *target parameters* for which we seek an explanation, prediction, classification or cause. If it is possible to identify them at this stage we will benefit from better knowledge of what parameters are important and of suitable transformations.

Unit Standardisation in Each Data Source

The standardisation of units in all data sources is necessary to have comparable ranges of values for related parameters in the final merged data set. Usually this is done by a straightforward mapping of values in one unit into another, such as mapping imperial units to metric if both are present in the data set, but often the need to do so is discovered only comparatively late in the process. Because later steps such as anomaly detection and repair may depend on the choices made here, it is still placed early in the data preparation process. Although important for some models, transformations such as normalisation and whitening are however usually not necessary at this stage.

Determination of Case Identifier

When several data sources have to be merged, it is often necessary to choose and sometimes (re)construct a parameter in each data source that uniquely identifies each case or item in the *final* analysis. This may have to be done in different ways for each data source and may not generate unique entries in every source.

Type analysis of each data source

Analysis of the type, range and format of data represented in each parameter is an often surprisingly difficult problem that will have a significant impact on the result of the later preparation, modelling and analysis steps. For example, the representation and limitations of models of data are often very different depending on whether parameters are continuous or discrete, and results are highly dependent on correct estimates of parameter ranges or domains. Correct type analysis is therefore a very important step in the data preparation process.

Ideally, an expert with insight into how the data bases were constructed should be consulted when selecting the most suitable type and range of each parameter. In practise this is seldom feasible, since the experts rarely have the necessary time to spare and the number of parameters selected initially can be very large. For this reason it is desirable to simplify and speed up the type analysis step by making it at least semi-automatic.

Type analysis may also be complicated by the occurrence of damaged or missing data, or the use of ad-hoc encodings. A common example of this is the encoding of

missing data by using a particular value outside (or even worse, inside) the normal range of the parameter, *e. g.* string or characters in data that normally consist of integers, or specific values such as -9999 or 0. This entails that the result of the type analysis step is not only a unique type and range for each parameter, but also input to the repair and recoding step described below. Type analysis involves:

1. Identification of the representation and actual range or domain of each parameter in every data source.

2. Application of some heuristic method to decide if a particular parameter seems to represent samples from a discrete or continuous domain. The heuristic analysis should preferably not only result in a type classification for each parameter, but also give the grounds for this classification. This allows the user to catch type errors at an early stage.

3. Inspection and evaluation of the results.

4. Manual correction of the results and descriptions from the heuristic method using common sense and/or expert knowledge, rudimentary interpretation and possibly classification and encoding of text parameters as *e. g.* discrete or continuous data and missing values. This step should also include annotation of data that, by the use of expert knowledge, describes both valid parameter ranges and domains and unit/dimension where applicable. This information is highly valuable in later modelling and anomaly detection steps.

The result of the manual correction must again be evaluated, perhaps again using heuristics for automatic type identification, and the procedure iterated until the result is satisfactory.

Repair and Recoding of Each Data Source

The visualisation and inspection of the result of the type analysis generally reveal deficiencies and anomalies in the data that need to be rectified before data preparation and analysis can proceed. This situation is especially pronounced in industrial applications where data often consist of samples of sensor readings which quite frequently drift or report anomalous values, combined with *e. g.* transmission errors and erroneous manual entries of laboratory measurements and error reports.

Parameters may also be correlated in such a way that the fact that an entry for a particular parameter is missing or is out of range means that values for a subset of the other parameters are useless or should be interpreted in a different way. The detection of such cases generally requires knowledge of the processes involved and/or of idiosyncrasies in the data collection routines. It is also common that the analysis has to proceed quite far before any such anomalies are detected. The repair and recoding step may thus have to be revisited several times, possibly unexpectedly late in the modelling and analysis process.

Methods for automatic detection of possible errors and data cleaning are also highly useful here. Typically, some typographical errors such as misspellings or misplaced decimal points can be comparatively easily detected (see *e. g.* [Lee *et al.*, 1999]). If proper operating ranges for parameters have been defined during type analysis, values that fall outside of these can also be found easily. In fact, some errors can be detected just based on the type of the parameter: if a temperature reading shows a value below zero Kelvin, something must be wrong. In process industry data, data reconciliation by *e. g.* mass and energy balancing may also be useful to detect errors.

In most cases, the repair and interpretation step involves:

1. Reconstruction of obviously damaged parameter data in several data sources. A simple example could be identifying a missing product number from a specified batch, when all product numbers in a batch are identical.

2. Identification, classification and possibly recoding of process states that give rise to missing and deviant data in several data sources. The representation of missing or irrelevant data should also be taken care of in this step. Not only recoding of parameters, but also introduction of derived parameters should preferably be performed at this point, especially if these parameters are necessary for a later merge operation.

3. Visualisation and validation of repaired data.

All of these steps may have to be iterated and revisited several times.

Selection of Parameters and Entries in Each Data Source

This step is really a refinement of the choices made in the selection of data sources. The reason to introduce a specific step to revise the choices is that we are, after the type analysis and repair steps, in a better position to assess the quality and usefulness of the data. We may also want to revise or refine the choices made in a first iteration of this step based on results from several steps of preliminary analysis of *e. g.* correlations between parameters in the data, or exclude subsets of entries identified as anomalies. Obviously, the criteria for selecting entries differ from those used for selecting parameters:

1. Detection of anomalies and artifacts in the data. Automatic or semi-automatic methods for the early detection of anomalies would be highly desirable, but that represents a non-trivial data analysis problem in itself and is out of the scope of this text. See *e. g.* [Barnett and Lewis, 1994; Aggarwal and Yu, 2001; Eskin, 2000] for an introduction and further references.

2. Selection of entries. This generally involves exclusion of entries for which information on the target parameters is missing; filtering of damaged parameters and other artifacts that have not been possible to correct in the repair and recoding step.

3. Redundancy detection and collection of expert knowledge relevant to parameter selection. This includes correlation analysis, cluster detection and several other techniques (see *e. g.* [Duda and Hart, 1973; Liu and Motoda, 1998; Jain and Zongker, 1997]).

4. Selection of parameters. This is most often based on the result of the type analysis and preliminary structure detection but may also rely on expert knowledge. Different levels of such knowledge should be considered, from naive interpretation of parameter names in the original data sources to more or less complete knowledge of the physical, chemical, economical and social factors involved in the studied process and their representation in the data.

During selection, we might also consider sampling data to reduce the size of the data set. Sampling theory is well described (*e. g.* [Cochran, 1977]) and critical in many data analysis tasks. For industrial data however this is rarely an issue, due to the practical difficulties in acquiring enough data even for a representative view of all aspects of the process.

Merging of Several Data Sources

Merging of data from several data sources is in many cases the single most complex step in the preparation of data. In addition, since data is often given in the form of time series with different sampling rates, there is an obvious need to be able to reliably re-sample the data using *e. g.* interpolation, aggregation, integration and related techniques.

Many problems with data in individual data sources become obvious only when merging several data sources into one. Since the merge itself generally is quite a time consuming and error prone operation, it is desirable that the way that the merge is computed is very precisely specified and, most importantly, that it is reproducible in the (very common) case that some earlier data preparation steps have to be revisited.

Common merge problems include cases where one or more data sources contain time series of measurements, possibly with different sampling rates, corresponding to a single entry in another data source. In such cases it is usually necessary to fold (transpose) the time series into a set of additional parameters for the single entry, possibly re-sampling, integrating and transforming the data in the time series into a more compact representation, or one believed to be more relevant to the analysis task at hand.

It should be noted that such transformations can easily reduce but also bring out correlations that otherwise could be very difficult to detect by automatic methods. Choices of this kind are really part of the modelling phase, and the correct decisions should be based on thorough knowledge of the problem, on the analysis task at hand and on a fair bit of experimentation with alternative representations and models.

Differences and errors in textual representations of identifiers can also cause severe problems while merging. Although such problems are often solvable by

defining two corresponding sets of regular expressions for matching, more elaborate methods might have to be applied [Hernandez and Stolfo, 1997].

Final Selection of Entries and Parameters

At this point, after merging several data sources, it is usually a good idea to revisit the data selection stage, but now for the complete data set. Quite often, a merge operation introduces redundancy in both parameters and entries, and structure that was not visible before the merge (since the data sets were not complete) is now detectable.

The final selection of entries and parameters has the same structure as the one performed for each data source, but some correlations (or lack thereof) may be difficult to detect before the data sources have been merged. Once the usefulness or uselessness of certain subsets of data has been determined, the selection can either be done for each data source or directly in the merged data. The methodology outlined in figure 6.2 singles out the second alternative, but going back to the selection step for each data source instead might make sense if the merge operation is simplified by a refined selection on the individual data sources.

Data Partitioning

At this stage it is suitable to, if appropriate, partition data into *training* (estimation), *validation*, and *testing* data sets for the modelling phase. This partitioning is not necessarily easily made in industrial process data, due to the fact that the production process usually moves quite slowly through a huge state space. The different data sets may have to be separated in time by weeks, or even months, to be useful for estimating a model's actual predictive capabilities. However, estimating the necessary partitioning reliably is difficult, and this step might require a high degree of trial and error.

Modelling

As already pointed out, it is not easy to draw a clear line of where data preparation ends and modelling begins. In fact, we have already touched upon procedures that might be referred to as modelling in earlier steps, perhaps most notably in case and parameter selection, as determining dependencies and finding clusters rely on assumptions and parameterisations of data that are in effect models themselves. For the current discussion it is sufficient to note that at this stage it is often suitable to perform another kind of preparatory modelling: introducing derived parameters, or *features*.

By features we mean new parameters that can, often quite easily, be calculated from other parameters in data. Features may have been introduced earlier to be able to merge data sets with no simple parameter overlap. The reason to introduce them in this step is slightly different in that it is here primarily a mean to insert expert knowledge into the data. For example, the temperature over gas volume

quotient may very well contain predictive information, and is easily calculated if we know both the volume and temperature. The reason to introduce this feature into the data set is that it represents useful knowledge: although we already know the parameters from which it is calculated, a particular analysis method might not easily find this relation by itself.

In industrial data, common features include everything from simple functions such as durations between time points to complex and area specific correlations, as well as recoding of data in orthogonal bases using *e. g.* principal component analysis (PCA) [Jolliffe, 1986] or calculating spectral properties using Fourier or wavelet transforms [Daubechies, 1990]. What kind of derived parameters that are useful for a certain application is often very difficult to know beforehand. Cataloguing common useful features for a variety of problems and industrial areas would be an interesting research problem in itself. Feature construction and selection, both automatic and manual, are difficult problems, but an introduction to some useful methods can be found in *e. g.* [Liu and Motoda, 1998; Jain and Zongker, 1997].

Generic Operations and Transforms

The methodology outlined in the previous section specifies *what* needs to be done to the data in order to prepare it for modelling and analysis, and proposes an order in which to perform the preparation steps. What remains to describe is *how* to perform the necessary preparations. For this purpose we have identified a small set of generic operations, with a focus on data transformation, of which all data preparation tasks outlined above are special cases. These operations can be composed into a *reproducible sequence of transforms*, thereby making it possible to move back and forth between the preparation steps in a controlled manner.

The set of operations is for practical purposes complete, and although it is deliberately small, we have allowed for some overlap in the functionality of the operations. This implies that a given data preparation task may be accomplished in several ways.

The aim is that, using the generic operations, it should be comparatively quick and easy to get data into a state where anomalies and deficiencies can be subjected to a preliminary analysis. It should then be possible to iteratively devise means to correct and compensate for these anomalies and deficiencies by reiterating and modifying the operations already used, and introducing new operations at the most suitable point in the process. An overview of the generic operations is provided in table 6.1.

Note that we have listed no supporting operations for problem classification, selection of data sources and identification of target parameters apart from visualisation tools since these steps by their nature require manual inspection and decisions.

Preparation step	Operation	Description
Visualisation and inspection	*inspect*	Inspect data
	plot	Plot / visualise data
Type analysis and data loading	*guess*	Guess parameter types
	sample	Load sample of data source
	read	Load complete data
Repair and recoding	*replace*	Replace entries
	derive	Compute derived parameter
	match	Match parameters
Selection	*select*	Select entries and parameters
Merging	*merge*	Re-sample and merge data sets

Table 6.1: Overview of generic operations and transforms

Visualisation, Inspection and Other Supporting Operations

Although not described as a separate step in the methodology, visualisation (used here to describe graphical presentation of data) and inspection (used here for viewing data in a textual format) routines are implicitly used by many of the other steps for exploration and verification of results. In fact, throughout the whole data preparation process, visualisation and inspection of both type information and the data itself is essential (see *e. g.* [Fayyad *et al.*, 2001]), not only for discovering structure and dependencies in data, but also to get a more basic understanding of what the attributes represent and on what form. Although visualisation has not been the main focus of our work, a number of operations and application properties that are useful for data preparation have been identified. For very large data sets it can be a good idea to base initial visualisation and inspection on a *sample* of the complete data.

An effective tool should include at least mechanisms to *inspect* the type derived for each parameter in the data, and be able to describe and visualise the distribution of both continuous and discrete parameters. Apart from calculating common descriptive statistics measuring *e. g.* the central tendency and dispersion, this includes inspection of the number and distribution of cases for discrete parameters as well as *e. g.* histograms or dot plots [Tukey and Tukey, 1990] for continuous parameters.

Visualisations, or *plots*, are very useful for determining the nature of the relations in the data and finding potential clusters. Examples of useful visualisations are scatter plots in several dimensions [Swayne *et al.*, 1998], parallel coordinates plots, and simple series plots, possibly combined with correllograms for time dependent data.

Common for all types of visualisations during data preparation is that they should support identification of anomalous data entries and verifying type information. For example, a discrete variable mistakenly represented as continuous can

usually easily be identified in a scatter plot, as can potential outliers. Other important properties of the graphics system are interactivity and consistency. Graphics should directly reflect changes to the data or its representation, making it easier for the user to explore the data and explain phenomena in it. Changes made in one visualisation should be noticeable in others that represent the same data.

Visualisation and inspection can be viewed as operations that provide information on useful instantiations of data transformation primitives such as replace, derive and select. This is also true for anomaly detection and structure analysis; these tasks could also be viewed as generic operations on the same level as visualisation and inspection. Detecting anomalies and structure in data are discussed in somewhat more detail in chapter 4.

Type Analysis and Data Loading

As discussed in the methodology, type analysis is important for understanding, modelling and error correction of the data. Also, just reading data into a database and storing it efficiently generally require some rudimentary form of interpretation. Since data analysis frequently deals with very large amounts of data it may not be feasible to store *e. g.* numbers and enumerated types as strings. For these reasons we need a mechanism to specify and preferably also at least semi-automatically derive, or *guess*, parameter types from the data. We also need mechanisms for modifying this type information by explicit specification when necessary. Specification and derivation of field types need to be done for several data sources and there is a need for a mechanism to uniquely refer to the results of such a read operation. This gives us that a generic *read* operation should support

- Sufficiently general input formats, *e. g.* typical export formats of common database systems and spread sheets.
- The ability to guess the type of each parameter using a suitable heuristic, and then modifying this guess according to a manually generated specification.
- A mechanism to extract only a *sample* of a given data source. This is for practical reasons, as experimenting on complete data sets throughout the whole data preparation process might be too computationally demanding.

These mechanisms can be conveniently provided by parameterising a single read operation for each data source.

Repair, Recoding and Derivation

There is frequently a need to transform data into another unit or representation. We may also wish to modify a determined type into another one: a discrete parameter into a continuous, a string into an enumerated type or a date etc. This is sometimes most conveniently done by replacing the original value by a value computed from

the old by a function, possibly also using values of other parameters as input. Such a mechanism can also be used to discretise a continuous parameter, replacing certain ranges of parameter values with a uniform representation of the fact that the value is missing, or to repair, reconstruct or re-sample data. This operation, *replace*, should

- Use a single transformation function that returns values in accordance with a single resultant type.
- Let the transformation function inspect any number of parameter values in addition to the one being replaced.

Some but not all data analysis methods are capable of automatically detecting correlations between clusters of parameters. However, in many cases it is as already argued useful to introduce derived parameters (features) from ones already present in the data. We wish to allow the analyst the choice of which path to take and therefore provide a mechanism to manually add derived parameters. The primitive operation used to do this, *derive*, should

- Return for each entry in the input data a derived value as a function of a fixed selection of parameters in the old data.
- Return data conforming to a predetermined type, but allow for handling of anomalies in input types.

A special case of replacement and derivation in the above sense is where the input fields contain data that need to be discretised or classified. One very convenient way to do this is to specify each class as a regular expression that can be matched to the original data. For this reason we propose a specialised form of the derive operation, *match*, which takes a finite list of regular expressions and returns a discrete value corresponding to an enumerated type with as many cases as the number of regular expressions given. Each regular expression corresponds to one discrete class of the original data.

Selection of Entries and Parameters in a Single Data Source

Frequently the data sources contain redundant or irrelevant parameters, and entries that for some reason or other are not usable. Since this may not be obvious initially it is very useful to be able to select subsets of the data in a traceable and retractable way. Selections of parameters can generally be done using only indexes or parameter names while selection of entries may depend on the actual values in subsets of fields in each entry. The primitive operation *select* allows us to do simultaneous selections of parameters and entries by specifying a list of names or indexes of the parameters to be selected and a boolean filter function used to select entries from the input data.

Merging Data

One of the most complex operations in data preparation is the merging of several data sources. It would be almost impossible to provide high level primitives capturing all possible merge and re-sample needs. For this reason we have chosen to specify a very general mechanism for merging data sources, that for non trivial cases will require programming of functional parameters. It appears, however, to be possible to generalise and with time build up a library of working solutions as generic *merge functions or subroutines*. For the majority of *merge* tasks, we have made the assumption that

- A merge is always done between exactly two data sources. If more than two sources need to be merged this will have to be done by a sequence of merges[2].
- The entries in the merged data source will always consist of data derivable from exactly one entry in the first input data source and a number of matching entries in the second[3].
- The match should be (efficiently) computable by comparing a specified number of parameter pairs, each pair consisting of a parameter from each input data source.
- For each entry in the first input data source a specified number of derived entries will be computed from the matching entries in the second.

These assumptions allow us to partition the computation of the merge into two steps:

1. Computation of a number of matching entries in the second input data source for each entry in the first input data source. This can be done in a completely general way, using a functional parameter to specify the match criterion.
2. Computation of each derived entry by a supplied merge function from the one entry in the first input data source and the corresponding match of entries in the second. Several types of common merge functions have been identified, such as interpolation, gradients and derivatives.

In short, a primitive merge operation should take as input exactly two data sources, a matching specification and a merge function, and produce a single data source as output.

[2]In cases where we wish to re-sample parameters in a single source, we can either use the derive operation or produce a new data source that can be merged with the original one in the manner described here.

[3]This assumption may seem overly restrictive but for all cases we have so far encountered, introducing a derived parameter in one or (rarely) both data sources have allowed us to conveniently use a merge operation with this type of restriction.

Implementation of a Tool for Data Preparation

The general, primitive operations presented in the previous section could be implemented in many different ways. Several of the operations are similar to those in data base languages such as SQL [Date and Darwen, 1987], and if it is the case that the data sources are available as tables in a relational data base the implementation of some of the primitives would be trivial. SQL is, however, in its standard form unsuitable for *e. g.* advanced forms of merging.

Another common way to execute the primitive operations is to use text processing tools such as *sed*, *awk* and *perl*. Using such tools generally involves a significant amount of programming, and the programs developed may not always be as reusable as could be desired. A system with better integrated transformation primitives, using a higher level of abstraction would be preferable.

To support the methodology in section 6.3 an interactive tool for data preparation has been implemented. This tool is based on a previously developed general analysis environment, *gmdl* ([Gillblad *et al.*, 2004]), which will be discussed in some more detail in section 6.5. The tool is intended as a framework for data preparation, suitable for both interactive and batch oriented use, facilitating all common preparation tasks. Procedures for cleaning, transformation, selection and integration are provided, implementing the general operations discussed in section 6.3.

The implementation is directly related to, but takes a different direction than [Sattler and Schallehn, 2001], which uses an extension of SQL to provide an environment that supports more general data preparation tasks such as data filtering, reduction and cleaning. [Sattler and Schallehn, 2001] makes use of an imperative language, while we propose a functional approach that better suits the use of generic operations, an approach that hopefully will be more general and allow for faster prototyping.

Requirements Specification

A tool for data preparation, analysis and modelling should be designed to fulfil a number of criteria. While keeping the interface simple, it should be possible to handle a large variety of problems by keeping it extensible and adaptable. A general data preparation system must also be fully programmable by the user. Almost all data preparation tasks differ slightly from each other, and while *some* subtleties might be possible to handle in a system that cannot be fully programmed or extended, it is bound to one day run into a task that cannot be handled within such a system.

As discussed throughout this section, data preparation is a highly iterative process and it must be possible to go back and change previous decisions at any step of the process. Hence, a very important requirement is that a support tool should contain mechanisms for documenting each step of the data preparation process in such a way that the process can be easily re-traced and re-run with changed operations.

In addition to the mechanisms dedicated to the data preparation, the implemen-

tation should contain facilities for *data abstraction*, *type management* and consistent representation and transparent handling of *missing values*. It should provide a mechanism to refer to the parameters that is independent of the order in which the parameters were given in the data source and allow free naming and index insertion of derived fields. Preferably, values of derived fields should also be computed dynamically and on demand, so that modification of input data automatically results in re-computation of the derived fields whenever their values are requested. These facilities are provided automatically by the *gmdl* environment on which the data preparation routines are based.

Design Choices

In the underlying analysis environment, data are abstracted to a set of *entries* (rows), each consisting of a set of *parameters* (columns), the intersection of an entry and a parameter being referred to as a *field*. Each parameter has an associated *type*, used to interpret and encode all the fields of that parameter. The set of types is currently limited to: *continuous* (real), *discrete* (integer), *symbol* (enum), *date*, *time* and *string* (uncoded) although additional annotations of each parameter are provided to encode additional type information such as unit, dimension etc.

Data are represented in essentially two main ways in the library. The user does not have to differentiate between them, but the concept is still relevant for this discussion. It can either be stored explicitly, to provide efficient storage of and access to static data, or as virtual data providing a facility to derive new parameters from existing data using a filter construct. This provision is used extensively in the implementation of the data preparation primitives to allow a type of lazy computation of derived parameters and transforms. This also gives us very fast execution of a sequence of operations and inspection of subsets of the data before committing the data to disk.

Most of the data preparation operations described below take an arbitrary data object as input and return a virtual data object as result. This makes the operations clean in the sense that the original data are always unchanged. For most operations it also makes the transformation very fast. However, accessing the data in the resulting virtual data object may be slow. For this reason an explicit *check point* mechanism has also been implemented that will transform the virtual data object to a storage data object and optionally write a representation of it to disk. This means that during the incremental construction of a script to specify a sequence of transformations of the original data, the result of stable parts of the script may be committed to disk. The parts of the script that generate the check point files are then re-executed only when really necessary. The scripting mechanism includes a script compiler and a top level interpreter, which handles the check point mechanism and provides abstractions of the primitive operations as *commands* where the input and output data objects are implicit.

In the following subsections the elements of the implementation most important to data preparation are briefly described and exemplified. The examples are taken

from a real case of error detection in steel manufacturing (see [Holst and Kreuger, 2004]). This pilot case was first prepared using *e. g.* emacs, sed and awk, and then reconstructed using the methods and tools described in this section. Apart from the time spent on understanding the application area, the time invested in the first iteration of data preparation for this test case was still comparable to the total time taken to both implement the support tool and preparing the data using the new tool.

Scripting and Maintenance Functions

The functions `define-script`, `script-read`, `script-check` and `script-write` are used to specify a sequence of data transformations as a script, for I/O and for convenience during the iterative development of the final version of the script. Example 6.3 illustrates one typical use of these functions.

```
(define-script (example-1 SaveName)
  (script-read '(((int (0 . 999)) (0 . 3))
                 (string 4 5 6)))
  <command 1>
  <command 2>
  (script-check)
  <command 3>
  ...
  (script-write SaveName))

;;; (example-1 "<file spec>" "<destination path>") ; Call script
```

Figure 6.3: Example of script I/O functionality.

Example 6.3 defines a script which takes as an argument a file name used to store the result of executing the script. When called it takes in addition and as its first argument an input file specification which is implicit within in the script but passed on to the `script-read` command. `script-read` should always occur first in the script and will generate the initial data object operated on by the remaining commands in the script.

The `script-read` command will analyse the file indicated by `<file spec>` and generate a tentative typing of all the parameters in the file, modify the typing as specified by its first argument and store the typing information to disk. The type specification in this case indicates that the parameters here given as `0` through `3` should be interpreted as integers in the range `0..999` and the parameters `4`, `5` and `6` as strings of characters. This result is stored to disk as a type file associated with the data file indicated by `<file spec>`. If the script is rerun, the typing will not be performed again unless forced by adding an additional argument to `script-read`. Instead the type-file is used to read in the data.

If the first argument supplied to the script is a sample specification, the sample will be generated and stored to disk. Using either the sample or the original data file the `script-read` command will then generate and pass a storage data object to `<command 1>`, which will in turn pass its own result as a data object to `<command 2>` and so on.

The result of executing `<command 2>` will be saved as a check point file before being passed to `<command 3>` unless the check-point already exists, in which case neither the `script-read` nor the following commands up to the `script-check` will be executed at all. Instead the check point file will be used and the result of reading it will be passed to `<command 3>`.

The script returns the result of its last command, in this case a write command with the obvious semantics.

Data Transformations

Example 6.4 below illustrates a range of simple transformations.

```
(define-script (script-name SaveName)
  ...
  (data-replace (replace-filter >= 0)
    '(cont (0 . 2000)) '(8))
  (data-derive (access -)
    '(int (0 . 172800) "ct0-ch") '(71 29))
  (data-match 85
    '(disc ("No" "hasp.*680" "hasp.*750") "HT"))
  (data-select (lambda (entry) #t)
    '(2 3 4 (8 . 12) 14 (16 . 19) 21 23))
  ...)
```

Figure 6.4: A range of simple transformations.

`data-replace` returns a virtual data object in which the values of all fields for a given set of parameters have been mapped to new values of a corresponding new field type. A transform function given as an argument should take three arguments: the original data object, an entry and a parameter. In example 6.4, the command replaces all values below zero in the parameter `8` with the dedicated symbol for missing values (`#?`) and restricts the type of the parameters to be in the range between `0` and `2000`.

The `data-derive` command returns a virtual data object to which a new parameter of a given type has been added. The values of the fields in the new parameter are derived by a supplied transformation function that is applied to the original data object, an entry and a list of the parameters. In the example, a new parameter named `ct0-ch` is added, with a continuous type with a range between `0` and `172800` computed as the difference of the values of parameters `71` and `29`.

`data-match` adds a new parameter, `HT,` classifying the string values in parameter `85` as one of three values depending on the result of a match with the regular expressions `/hasp.*680/` and `/hasp.*750/`, defaulting to `No` if no match is found.

The `data-select` command, finally, selects a subset of parameters. `data-select` always returns a virtual data object where only the parameters given and entries that passes a given test function remains. The test function provided in example 6.4 selects the given parameters and leaves all entries of the input data object in the output data object.

As discussed in section 6.3, merge is one of the most complex operations in data preparation. Therefore, the `data-merge` command is defined as a general function that in all but trivial cases will require programming of functional parameters.

`data-merge` merges data from two data-sources, `db0` and `db1`. This is done by adding one or more parameters derived from `db1` to a new virtual data object initially identical to `db0`. First, a match is generated. The match consists of the entries in `db1` that match each entry in `db0` for each of the parameter pairs given.

The remaining arguments specify how to derive additional parameters. For each parameter specified in `db1`, `a` fixed number `n` of new parameters will be derived from the match. A supplied *value function* computes the value of each new field of a type computed by a corresponding type function. The value function should take four arguments: the data object; a list of entries, which represents the indexes in `db1` matched to the current entry in `db0`; a parameter in `db1;` and an integer between `0` and `n-1`. This is typically used to sample or enumerate fields in consecutive entries in `db1` to create several new fields in the resulting data object. As an example, to transpose the vector of values of a field in several matched entries into a number of new fields, the following value function could be used:

```
(lambda (db1 entries param i) (db1 (list-ref entries i) param)
```

The next argument is a *type function* is useful for deriving the new field format as a function of the format of the parameter from which its value is derived. To use a unique version of the original parameter name one could *e. g.* use the following type function:

```
(lambda (param i type dom name)
  (list type dom (format #f "~a-~a" name i)))
```

A small number of useful and convenient merge value functions and utilities are provided by the scripting module. Example 6.5 illustrates the use of `data-merge`.

The data object produced by the prefix is merged with a data object produced by the script `scrpt1` called with the source file `<source 1>` matching entries for parameters `0` and `1` in both sources. In this case the match is assumed to contain exactly one entry from `<source 1>`, the value of which is appended to the primary data object obtained from `<source 0>` with the same parameter name as in `<source 1>` but with a continuous type with a range between `0` and `100`. Exactly one new parameter for each parameter between `2` and `29` in `<source 1>` is generated.

```
(define-script (scrpt1 Src1 Src2)
  ...
  ;; Merge source 1
  (data-merge (scrpt1 ',Src1)  ; Merge subscript 1
    '((0 . 0)(1 . 1))          ; Match parameters
    (lambda (d rws c i)        ; Value function
      (d (list-ref rws i) c))
    (lambda (c i tp dmn nm)    ; Type function
      `(cont (0 . 100) ,nm))
    1 '((2 . 29)))             ; No. of samples & source fields
  ;; Merge source 2
  (data-merge (scpt2 ',Src2)   ; Merge subscript 2
    '((0 . 0)(1 . 1))          ; Match parameters
    (lambda (d rws c i)        ; Value function
      (if (null? rws) "No" "Yes"))
    (lambda (c i tp dmn nm)    ; Type function
      '(disc ("No" "Yes") "Sliver"))
    1 '(0))                    ; No. of samples & source fields
    ...)

;;; (scrpt "<source 0>" "<source 1>" "<source 2>"); Call script
```

Figure 6.5: Merging two data sources.

The result of this operation is then merged with a data object generated by the `scrpt2` called with `<source 2>`, using the same matching parameters, but now adding a boolean parameter depending on if the match is empty or not.

`resample` is a generic sampling utility function for `data-merge` that iterates over the supplied entries and applies a given binary operator to a given parameter `s` of each consecutive entry and a supplied value `l` until it becomes true. It then applies a supplied interpolation function to the resulting entries and parameters.

Typically this is used to search the values of a particular parameter `s` for a value that exceeds the value `l`. The sample function typically interpolates between values of the fields in a given set of parameters for the current and the previous entries using *e. g.* linear interpolation, gradient, derivatives or splines. Two common sample functions are currently implemented as library routines: `interpolate` that returns an interpolated value of one parameter at which another parameter would take a given value and `gradient` that returns a gradient of one parameter at which another parameter would take a given value. Example 6.6 shows the use of `resample`, `interpolate`, and `gradient`.

This example illustrates a more complex merge operation with two distinct types of sampling of the data in the secondary data object. In both cases a fixed number of samples (`12`) are computed from a variable number of matched entries in

```
(define-script (scrpt Src1)
  ...
  (data-merge (frnc ',Src1)  ; Merge subscript
    '((0 . 0)(1 . 1))        ; Match parameters
    (lambda (d rws c i)      ; Value function 1
      (resample d rws 5 >= (* (++ i) 100)
      interpolate c 5))
    (lambda (c i tp dmn nm)  ; Type function 1
      `(,tp ,dmn ,(format #f "~a at ~a°C" nm (* (++ i) 100))))
    12 '(3 4 (6 . 14))       ; No. of samples & source fields
  (lambda (d rws c i); Value function 2
    (resample d rws 5 >= (* (++ i) 100) gradient c 3))
  (lambda (c i tp dmn nm)    ; Type function 2
    `(cont ,(if (= c 4) '(0 . 300) '(0 . 30))
    ,(format #f "~a incr/min at ~a°C" nm (* (++ i) 100))))

    12 '(4 5 7 8 9 12 13 14)) ; No. of samples & source fields
  ...)
```

Figure 6.6: A more complex merge of data sources.

the secondary object. Both use the library procedure `resample` but with different sample functions (`interpolate` and `gradient`). Most of the data for which the gradient data are computed is also sampled by interpolation.

Summary and Concluding Remarks

In this section, we have discussed a methodology for data preparation of complex, noisy data from multiple data sources and shown how the methodology can be supported by a small set of high level and generic operations. By constructing a tool implementing the generic operations we have suggested how the operations can be put into practical use for the preparation of industrial data, prior to modelling and analysis.

Although data preparation and merging is exploratory by nature, the process can benefit greatly from the structured approach suggested in the methodology. Being able to restart the sequence of operations at any point is a major advantage whenever the need to modify, add or delete individual transformation operations is detected later in the preparation or modelling process. This is, as we have already stressed, a very common situation. The implemented tool combines the set of primitives with a scripting facility and checkpoint mechanism as described in section 6.3, making each step in the process of data preparation *self documenting*, *revocable*, *reproducible* and generally much more efficient in terms of time invested.

Although the primitive operations are derived from the methodology, the methodology itself could be used without the proposed primitives, *e. g.* in the context of an SQL data base or when using tools such as *sed*, *awk* or *perl*. However, we argue that a dedicated implementation of this particular set of primitives is an efficient way to reduce the traditionally considerable time spent on data preparation, especially for the analysis of industrial data. The time spent on the preparation phase was reduced by orders of magnitude for our pilot case, and we firmly believe that this will be typical also for future applications. Also, the importance of efficient, semi-automatic type analysis should not be underestimated, as this is usually a difficult problem.

As for future developments, the methodology and implementation will be applied to other case studies, and based on such results, possibly refined. Different types of automatic or semi-automatic techniques for detection of anomalies and correlations would also clearly facilitate the data preparation process, and could additionally be of use in the modelling steps following the data preparation. For example, extending the use of the type analysis and annotation of parameters with range, domain and unit information would be highly beneficial.

To be truly useful for the general analyst, the implementation of the merge operation requires a library of common merge and interpolation subroutines. Some examples of such library routines were given in section 6.3. Further application of the implementation is expected to result in a larger set of useful such routines being identified. Similarly, a set of useful features for different types of industrial applications should also be identified, along with guidelines of when they are usable.

6.4 Modelling and Validation

Although by no means less important than data preparation, it is very difficult to provide a modelling and validation methodology applicable to a multitude of machine learning models and applications. Suitable approaches to modelling can differ greatly between different types of machine learning algorithms, as the model structure and parameters usually are very different. A usable methodology for applying the hierarchical graph mixtures described earlier would of course be highly desirable and possible to study, but this is unfortunately outside the scope of this thesis and must be considered to be future work.

Validation suffers from a similar problem in that it is often difficult to find a usable validation scheme that provides a good estimate of generalisation performance in practise, where available data are often limited or sampled from a limited subset of the total state space. A good understanding of the application area is often a necessity when evaluating models. For data that at least approximately fulfil a number of validation requirements that are often taken for granted, such as samples being independent and a non-biased sample set, there are a number of useful approaches already discussed in chapter 1, but we would like to refer the reader to the standard machine learning and statistical literature for a thorough

discussion of the subject (see *e. g.* [Mitchell, 1997; Witten and Frank, 1999]).

6.5 Tools for Data Preparation and Modelling

Introduction

To support the data analysis and modelling process, we have constructed a set of tools and libraries that are intended to address a number of rather different needs of a complete data analysis environment (see figure 6.7 for an overview). For many applications, especially within real-time systems, speed and efficiency are very important. The models and data transformations that form the foundation of the result of the whole analysis process should also be easy to incorporate into new or existing applications.

For these reasons, the foundation of the environment is an efficient low-level C++ library, without any external dependencies in the form of additional libraries. This makes it easier to create specialised applications based on the library and to embed it into other software frameworks. This library, which we will refer to as *mdl* (or *m*odel *d*escription *l*ibrary), mainly provides functionality for managing and transforming data, creating and estimating various models, validation, and exporting the results.

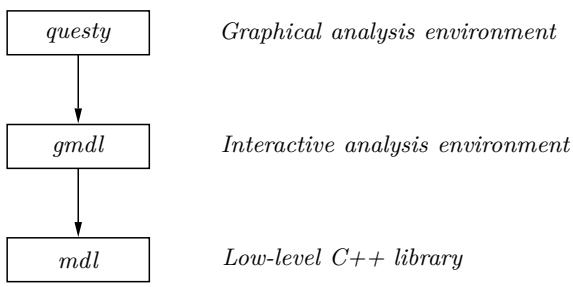

Figure 6.7: An overview the implemented analysis tools.

The *gmdl* (or *g*eneral *m*ode*l*ling *l*anguage) environment builds on the *mdl* library to provide an interactive, programmable, and easy to use data analysis framework. It is in essence an extension to the SCHEME programming language, and intends to provide for simple and efficient data transformation and rapid model prototyping and testing. All of the functionality of the *mdl* library is provided but in a much more accessible form.

Based on this interactive analysis environment is *questy*, an experimental graphical user interface that tries to support the data analysis process in an intuitive way. It is by no means a complete analysis workbench, but serves as a suggestion to how such an application could be constructed.

In this section we will give a brief overview of each of these libraries and applications.

A Low-Level Library for Data Analysis and Modelling

The focus of the low level *mdl* library is to provide a framework for resource intensive tasks within machine learning and data analysis. The library is intended to be portable and, to as large degree as possible, self contained, *i. e.* without dependencies to other libraries.

A basic requirement for the library is a consistent representation of data for all models and transformations. In *mdl* this *data object* is represented as a table, with a number of attributes or fields for each outcome. Most commonly, each instance in the table would be associated with an integer value specifying the "row number", but it is perfectly possible to use multi-dimensional indexing of the data object, *e. g.* for representing the levels of red, green, and blue in a two-dimensional image. In this case, the data could be accessed through a vector of integers of dimension two, specifying the position of the pixel in the image. Different cuts and views of any multi-dimensional data object can be accessed without using additional memory.

The data object provides a small set of access and modification methods to keep the interface simple. Representation wise, the data object is an ordered collection of *attribute* objects, representing the columns of the table. The attribute objects can be of several different types. The base class is abstract, and provides a common interface for specialisations that *e. g.* provide efficient storage of data or lazy evaluation of transformation functions. The latter is performed through a *filter* object, which contains references to one or more other attributes. These attributes are then used as input to a transformation function as specified in the classes that inherit from the filter class. The relations between the base classes is shown in 6.8. Also note that it is possible for attributes to contain references to other data objects, allowing for more flexible and efficient storage.

As we in many situations need to know on what format the data is stored, *e. g.* if an attribute is discrete or continuous, each data object is associated with a *data format*. Aside from storing some meta data common for the whole data object, the data format is represented as an array of *field formats*. The field formats can be accessed trough their position in the vector, or, if the order of fields is not important, through the name of the field. All attribute objects are associated with a field format.

The field format objects can be of several different types, most notably *continuous*, *discrete*, *integer*, *string*, and *data object* formats. These formats store information about *e. g.* field names, ranges in the case of continuous and integer data, and number of and names of the outcomes of a discrete format. The formats

also specify if the values stored are scalars or arrays, and, in the case of the latter, the expected dimensionality of these arrays.

Both data formats and field formats use copy on write semantics. The objects only store a pointer to the actual object representation, meaning that format copying or comparison is a mostly a matter of pointer assignment or comparison. Only when an object that is referenced multiple times is modified, memory is allocated, copied and modified.

There are several methods available for guessing the formats of data files and modifying these. Formats also provide means for converting between a textual external representation and the internal representation, which brings us to how we store and communicate data between procedures in the library.

Since we cannot assume that we will know all formats of data, model specifications etc. at compile time if we want to build a dynamic or interactive system with the library, we also need to be able to perform dynamic typing of data. This is done in the library through the *dtype* (*d*ynamic *type*). This is a discriminated type that can contain values of different types, indifferent to interpretation while preserving type information. A dtype can be assigned any copy constructable objects, storing objects smaller than a specified size directly within the dtype object itself, while larger objects are stored as a pointer to an object. The type of value contained can be tested with

```
dtype_is<typename>(const dtype& value)
```

and the value converted to the correct type using

```
dtype_cast<typename>(const dtype& value)}.
```

The library provides appropriate type definitions for all primitive types, such as continuous, integer and discrete data. The library also provide for consistent management of *unknown* values. Any primitive type or dtype can be tested for containing an unknown value using the procedure *is_unknown(...)*, returning true if the value is unknown and false otherwise. All procedures within the library are implemented to accept possible unknown values.

All models of data in *mdl* inherit from a basic *model* object (see figure 6.9). The base class keeps track of what attributes in a data set it is supposed to operate on, and, if appropriate, an associated data format specifying the domain on which the model is valid. All models inheriting from this base class must provide two methods: *estimate*, for estimating the parameters of the model from a data object, and *predict_values*, that should fill in the unknown values of an input vector as good as possible given the model parameters and the known values. This model object does not provide any separation between input and output attributes. If such a separation is appropriate or necessary, *e. g.* for a feed-forward neural network, the class must inherit from the *io_model* specialisation that provides such a separation. Also note that many models in the library are in fact hierarchical, *i. e.* composites of (possibly) simpler models such as ensemble models and models providing access to boosting and bagging algorithms.

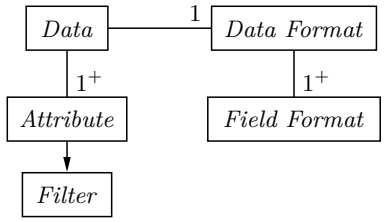

Figure 6.8: A somewhat simplified overview of data management related components.

A significant specialisation of the model class is the *distribution*, used as a base class for all multivariate distributions within the library. It extends the *model* with a number of necessary methods: *predict_distribution*, which calculates the conditional distribution over the unknown values in the input vector given the known values; *marginal*, returning the marginal distribution over the specified fields; and *probability*, returning the probability (or probability density) at the position specified in the input vector. These three methods allows us to perform most common operations on statistical models. The library provides representations for both simple parameterisations such as Gaussian (normal) distributions, general discrete distributions, and Poisson distributions; as well as more complex distributions such as graphical models and mixture models. The models available in the library also provides methods for calculating the entropy and Kullback-Leibler divergence, which *e. g.* allows for the construction of more complex information measures.

As there are standard interfaces for estimating models from data and performing classification and prediction, it is also possible to construct general validation routines. The *mdl* library provides routines for several kinds of cross-validation, where the results are calculated according to a wide variety of measures that can be inspected by the user.

In general, most resources in the library such as data objects, models and distributions, are managed through reference counting. All of these objects inherit from a basic *mdl_reference_counted* class that keep track of the number of references to an object, which is released when this counter arrives at zero. It provides *reference* and *release* methods to increase and decrease the counter respectively. The whole library is implemented using reference counting, as it is a relatively fast and portable approach to resource management. However, since it does increase implementation complexity somewhat and does not allow for circular data structures, a garbage collector can optionally be used, reducing the *reference* and *release* operations used in the library to null operations.

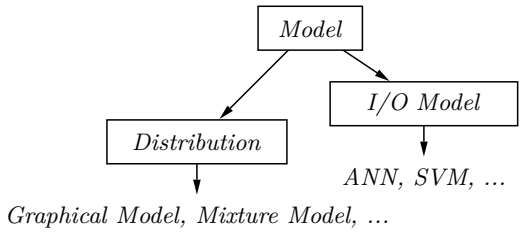

Figure 6.9: An overview of the model definition hierarchy.

Although an integral part of exploratory data analysis and model development, the library does not provide any visualisation functionality. As graphics and graphical user interfaces are highly dependent on the platform, it is very difficult to provide a portable implementation. However, the system does provide extensive facilities *e. g.* for descriptive statistics and hypothesis testing that do not require any graphical presentation.

Examples

Although it is easier to use the *gmdl* environment for most modelling purposes, there may be situations where it is more appropriate to use the low-level library directly. An example of this is the creation of specialised embedded applications. Therefore, let us have a look at two simple examples of using the low-level *mdl* library. The examples do not actually reflect typical use in the sense that they perform tasks that they are highly specialised, not taking any user input and using hard coded transformations and assumptions about data. The library interfaces are somewhat more suitable for slightly more complex applications, but the examples should still be easy enough to follow.

The code in figure 6.10 shows a manual specification of a data format, reading data from file, and using a nearest neighbour model to predict the unknown values of an example pattern. First, we specify the two outcomes *foo* and *bar* of a discrete attribute. We construct an array holding the field formats that we want to use for our data description, consisting of four continuous fields, one discrete field with the outcomes mentioned above, and one integer field. Using this field format array, we construct a data format, which in turn is used when creating a data object that we read from the file "datafile.dat".

We then specify the fields over which we want to construct a model, and create a k-nearest neighbour model with k set to 10. The model is estimated from the

```
char* field5_outcomes[2] = { "foo", "bar" };
field_format fformats[6] =
  { continuous_format(), continuous_format(),
    continuous_format(), continuous_format(),
    discrete_format(2, field5_outcomes),
    integer_format() };
data_format  dformat(fformats, 6);
data_object  data(dformat, "datafile.dat");

int model_fields[5] = { 0, 1, 2, 4, 5 };
k_nearest_neighbour knn_model(10, 5, dformat, model_fields);
knn_model.estimate(&data);

dtype data_vec[6] =
  { continuous(0.32), unknown(),
    continuous(0.83), unknown(),
    unknown(), integer(4) };
knn_model.predict_vals(data_vec);

printf("The predicted values are \%g and \%d\n",
        dtype_cast<continuous>(data_vec[1]),
dtype_cast<discrete>(data_vec[4]));
```

Figure 6.10: A simple example of using the *mdl* library.

data we read earlier, and can now be used for prediction. Learning the nearest neighbour model does not need to amount to much more than storing a reference to the training data, but here it will estimate normalising factors for all fields and set up the distance measure to reflect that we have a mix of continuous, discrete, and integer attributes.

Finally, we specify a data vector containing unknown values that we want to predict. The known values are explicitly cast to their correct types so that we can be sure there are no type errors. When the model then is used to predict the unknown values, it overwrites the unknown values with all the values it can predict, and finally these values are printed on screen.

In our second example, shown in figure 6.11 and 6.12, we will perform somewhat more complicated operations. The code shown is a slight simplification of the actual code used for testing anomaly detection performance for the network intrusion problem discussed in 7.5.

The main part of the code, shown in 6.11, starts by reading an already created data format from the file “netdata.form”, and then using this to create two data objects, containing data for estimation and evaluation, read from “netdata_estimation.dat”

```
data_format dformat("netdata.form");
data_object estimation(dformat, "netdata_estimation.dat"),
            evaluation(dformat, "netdata_evaluation.dat");
transform_data(estimation), transform_data(evaluation);
attribute* res_attrs[2] = {
  evaluation.get_attribute(12),
  new_storage_attribute(continuous_format(),
                        evaluation.entries()) };
data_object result(res_attrs, 2);

int model_attributes[4] = { 5, 6, 7, 8 };
distribution* components[5];
for(int i = 0; i < 5; i++)
  components[i] =
    delayed_release(new gaussian(4, dformat, model_attributes));
mixture_model mixture(5, components);
mixture.estimate_prior(&estimation);
mixture.estimate(&estimation);

dtype values[evaluation.fields()];
for(int i = 0; i < evaluation.entries(); i++) {
  mixture.fill_vals(values, &evaluation, i);
  dtype v = mixture.log_probability(values);
  result.set_entry(1, i, v);
}
```

Figure 6.11: A more complex example of using the *mdl* library.

and "netdata_evaluation.dat" respectively. These data sets are then transformed using the function shown in 6.12.

The transformation function operates on two attributes, 7 and 8, both transformed in the same manner. First, we add a constant one to the attribute by setting the attribute used in the dataset to a filter that performs this operation, using the original attribute as its input. We then proceed with adding a filter that takes the logarithm of all values. Indeed, the constant added before is simply to avoid taking the logarithm of 0 as the values stored in these attributes represent a transmission rate, ranging from zero to very large numbers. The logarithmic scaling is however not quite enough to clearly separate interesting clusters in data, which is why we also add a filter that takes the logarithmic value to the power of three, a parameter that was deemed suitable through manual inspection of data.

When the input data sets have been transformed, we start preparing a data object in which we can store the result of our experiment. This data set will

```
void transform_data(data_object& data) {
  for(int i = 7; i <= 8; i++) {
    data.set_attribute(i,
      delayed_release(new add_filter(data.get_attribute(i), 1.0)));
    data.set_attribute(i,
      delayed_release(new log_filter(data.get_attribute(i))));
    data.set_attribute(i,
      delayed_release(new power_filter(data.get_attribute(i), 3.0)));
  }
}
```

Figure 6.12: A more complex example of using the *mdl* library.

contain two attributes: one that contains the available classification of the type of network traffic and one where we store our estimation of how "normal" each pattern is. We create an array of two attributes, the first is selected directly from our evaluation data set and contains the traffic classification, and the second is a new attribute where we can store values, using a continuous format (and thus also a storage model that is efficient for this type of data) and allocating enough space to store the same number of values as there are entries in the evaluation data object. We then create a data object using these attributes. Note that no data are copied in the process, and that the first attribute in the new data object simply refers to the same attribute as the last attribute of the evaluation data, a useful property since in reality these data sets are quite large. Also note that the rather clumsy notation using the function `delayed_release` is due to resource management: the `delayed_release` function simply reduces the reference count of the newly allocated object, but puts an eventual delete operation on hold until the next time release is called. If we did not perform this operation here, the objects would never get deleted as they would assume one reference to many. This problem simply goes away if a garbage collector is used, simplifying notation and making the programs less error prone.

We then move on to create a model, in this case a mixture of Gaussian distributions. We specify what inputs to use for the model, which in this case is only a small subset, four attributes, of the available attributes, since the rest of the attributes are either redundant or unimportant for the application. An array of distributions containing different Gaussian distributions is specified, and a mixture model using these distributions is created. We then estimate the prior of the mixture model based on the estimation data set before finally estimating the mixture using the default estimation procedure, in this case a standard EM algorithm. Using a prior reflecting the complete estimation data set for all the component distributions is useful for regularisation of the model.

We calculate how "normal" a pattern is by finding the (logarithm of the) likelihood of a pattern being generated by the model for each pattern in the evaluation set. This is performed by creating a value vector of the correct size that for each instance in the data set is filled with the information relevant to the model, and then calculating the log-likelihood of these values. This is then stored directly in the result data set.

These are relatively simple examples that do not convey all aspects of the system, but could easily be used as starting points for writing more complex applications with the library.

Related Work

There are quite a few libraries available for machine learning related tasks, all with different areas of focus. The Weka library [Witten and Frank, 2005] has a structure somewhat similar to the *mdl* library, but is implemented in Java and provides visualisation routines as well as graphical user interfaces to the library. It is highly focused on standard machine learning methods and algorithms, on a level of abstraction that does not necessarily lend itself to the construction of application specific specialised models. Although the library is very extensive and efficient for exploring standard methods, it is perhaps less suitable for creating tailored, embeddable applications.

Another machine learning library that has reached a relatively high level of maturity is Torch [Collobert *et al.*, 2002]. Similar to *mdl*, the library is written in C++, but while *mdl* makes extensive use of more advanced features of the language, Torch deliberately uses only a very limited subset. Partly therefore, data management and representation are more limited in Torch compared to *mdl*. Torch does have a very strong focus on the machine learning algorithms, gradient descent related methods in particular, with a conceptual separation between the model parameterisation and the algorithm used for estimation. Although this may provide additional flexibility in some cases, the need for this separation may be limited in a library intended for creating machine learning applications and unnecessarily complicates most modelling situations.

An Interactive Environment for Data Analysis

A data analysis and modelling system should be designed to fulfil a number of criteria, sometimes in conflict with each other. For example, we want to provide both a simple interface and being able to handle a large variety of problems. We also want the system to be extensible and adaptable, while maintaining a simple model abstraction so that a problem can be described in a consistent manner. By removing weaknesses and restrictions of the system itself, in the way models, data and other functionality are being described, we can form an efficient and practical modelling environment that will support most data analysis tasks. The system we will describe here, *gmdl*, uses general abstractions of models and data and a

relatively small number of primitives and procedures to provide a flexible data analysis and modelling environment.

Another important requirement is that the system should be interactive and support fast prototyping with incremental changes. This implies that a programming language suitable for interpreted use should be chosen for the implementation. This narrows down the scope of possible language families somewhat, but although we would like to use a relatively modern, sound and preferably functional language, we also want to steer clear of languages that strictly use lazy evaluation and instead allow for side effects. Side effects come at a cost, but they make it easier to build a data analysis environment where efficiency in evaluation and memory usage is important. Lazy languages *do* have problems with liberal stack usage in Data Mining applications [Clare and King, 2003].

Based on this rationale, *gmdl* is based on the SCHEME programming language [Kelsey *et al.*, 1998], which is both simple, expressive and flexible enough to support most major programming paradigms. Unlike many data analysis systems in use today, this means that *gmdl* provides a sensible programming environment. *gmdl* is also intended to be interactive to as large an extent as possible. Data analysis is exploratory by nature, and this should be factored into and encouraged by the system. This might sometimes come into conflict with the need for efficiency in large calculations, but this conflict can, as we will see, be avoided by careful system design.

Redundant functionality has been avoided as far as possible in the design of *gmdl*. This will hopefully make the system more intuitive as a whole, but might deter some first time users since basic functionality might actually seem to be missing. With a basic understanding of the *gmdl* system and the usual work flow, this will hopefully not be a problem.

The procedures and types available in *gmdl* for modelling and analysis largely reflect those of the *mdl* library. There is a special model type, with procedures similar to those of the *mdl* library associated with it, although in a format that is more suitable to SCHEME. Specialised model objects represent distributions, ensemble models, principal component analysis models, and so on. Data sets are represented in a data set type, reflecting all the functionality of the underlying library but in a vastly more accessible way, along with procedures for interaction with standard data types such as vectors and lists. Format specification is simple and easily managed, due to the copy on write semantics employed.

Different types of data, such as continuous, integer, and string, are associated with and represented by built in data types such as real and integer numbers. One additional primitive type has been introduced, representing unknown values and written as `#?`, similar to the representations of the boolean values true (`#t`) and false (`#f`). We will here use a couple of examples to try to convey how the environment is used. For a thorough description of the language, please refer to [Gillblad *et al.*, 2004]

Examples

As we have already discussed a few examples related to data transformation and preparation in section 6.3, we will here discuss a couple of examples, one simple and one more complicated, related to modelling.

```
(define format (dformat (fformat-disc 3)
                        (fformat-cont)
                        (fformat-cont)))
(define data (make-data "datafile.data" format))

(define model
  (ada-boost
   (repeat 5 (nearest-neighbour '(0 1 2) format
                                :k 5 :normalize #t
                                :distance-weighted #f))
   '(0) :algorithm 'ada-boost-M1 :break-on-error #t))

(define res (model-cross-validate! model data '(0)))
(test-results-write (car res) "test-results.txt")

(model-estimate! model data)
(model-predict model (make-entry format 1 0.5 2 0.3))
```

Figure 6.13: A basic example of using *gmdl* for model generation, testing and application.

Example 6.13 shows the construction, estimation and validation of a simple machine learning model. It starts by specifying a data format with three fields; one discrete field with three outcomes and two continuous fields. Data are then read from the file `datafile.data` assuming this format. We then move on to specify a model, which in this case is an ensemble model consisting of 5 k-nearest neighbour models. The nearest neighbour models use data fields 0 to 2, assuming the format we specified earlier. The optional keywords in the model specification mean that we want to use the five nearest neighbours for prediction and classification, that the attributes should be normalised before distances are calculated and that we do not want to weigh the influence of an instance with its distance to the current pattern.

The ensemble model used uses the ada-boost algorithm for estimation, where attribute 0 is used to calculate the classification errors. Optional parameters specified with keywords set the specific ada-boost algorithm to be ada-boost M1, and specify that we do not want to stop estimating sub-models when the error is sufficiently small but rather use all five k-nearest neighbour models provided. The performance of the model is then estimated through leave-one-out cross validation

over the data set we specified earlier, calculating a large number of performance measures that are then written to the file `test-results.txt`. The following commands then estimate the model using the complete data set, and then uses the model to predict the class when the continuous attributes are set to 0.5 and 0.3.

```
(define form  (make-dformat "qrdata.format"))
(define pdata (make-data "prototypes.data" form))
(define cdata (make-data "cases.data" form))
(define class-field 0)
(define input-fields (diff (all-fields pdata) (list class-field)))

(define qrmodel
  (supervised-mixture-model
   class-field
   (count class 0 (fformat-outcomes (dformat-ref form class-field))
    (let ((class-data
           (data-select pdata '...
                        (data-field-match pdata class-field class))))
      (mixture-model
       (map
        (lambda (entry)
          (let ((prototype (data-select class-data
                                        '... (list entry))))
            (graph-model
             (map
              (lambda (field)
                (if (fformat-disc? (dformat-ref form field))
                    (disc-distr
                     (list field) prototype
                     :prior (disc-distr (list field) prototype
                                        :prior 'uniform))
                    (gaussian
                     (list field) prototype
                     :prior (gaussian (list field) prototype
                                      :prior 'uniform))))
              input-fields))))
        (all-entries class-data))
       :estimation-type 'graded-supervised)))
   :estimate cdata
   :class-prior (disc-distr (list class-field) pdata
                            :prior 'uniform)))
```

Figure 6.14: A more complex example of using the *gmdl* for creating statistical models.

In our second example, we will have a look at something a bit more complicated. Figure 6.14 shows the complete code for specifying and estimating the rather complex model used in chapter 5 to perform incremental diagnosis. Again, we start out by specifying formats and data sets. The format is read from file, and is then used when reading both the prototypical data and the case data used to estimate the model. We also specify which field we would like to use as the class field in the diagnosis, and specify the inputs to be used to predict this class as all fields in the format except for the class field.

The model is then specified as a supervised mixture model, where the first argument denotes the field used for specifying the mixture component and the second argument is a list of component distributions, calculated by a rather complex procedure that we will describe shortly. The optional keywords specify that the whole model, including component distributions, will be estimated from case data. The class (or component) prior is specified as a discrete distribution estimated from prototype data, which in turn uses a uniform prior.

The component distributions of the supervised mixture model are generated through the `count` expression, which iterates the subsequent expression a number of times equal to the number of outcomes in the class variable, binding the value `class` to the number of the current iteration starting at 0. The returned values for all iterations are returned as a list, representing the component distributions for the supervised mixture. Each of these components is specified as a mixture model, with the same number of components as there are prototypes for the certain class, found through the `data-select` procedure. The estimation procedure for the mixture is set to an algorithm using one step of the EM algorithm that uses the actual component distributions as starting points, *i. e.* no randomisation is performed initially.

Each of these mixture components is in turn specified as a graphical model containing one distribution for each input attribute, *i. e.* essentially a product model. If a certain attribute is discrete, we generate a discrete distribution estimated from a specific prototype, and setting the prior distribution for all further estimations to one again estimated from the specific prototype, using a uniform prior. If the attribute is not discrete, we instead assume a Gaussian distribution estimated in a similar manner to the discrete distribution.

Although the created model is highly specialised, designed to manage specific problems that occur in the incremental diagnosis case we describe in chapter 5, its expression is still relatively simple in the modelling language. The complexity of the expression comes from very specific assumptions on what data we want to use to estimate certain parts of the model and the use of several hierarchies of Bayesian prior distributions, all of which must be explicitly specified.

A Graphical Environment for Data Exploration and Model Generation

Based on the methodology and libraries we have discussed earlier, we have constructed an experimental graphical interface for data analysis and creating machine learning applications. The interface runs within, and can therefore be used in conjunction with *gmdl*. It mainly acts as a graphical shell to *gmdl* to simplify the creation and deployment of machine learning models.

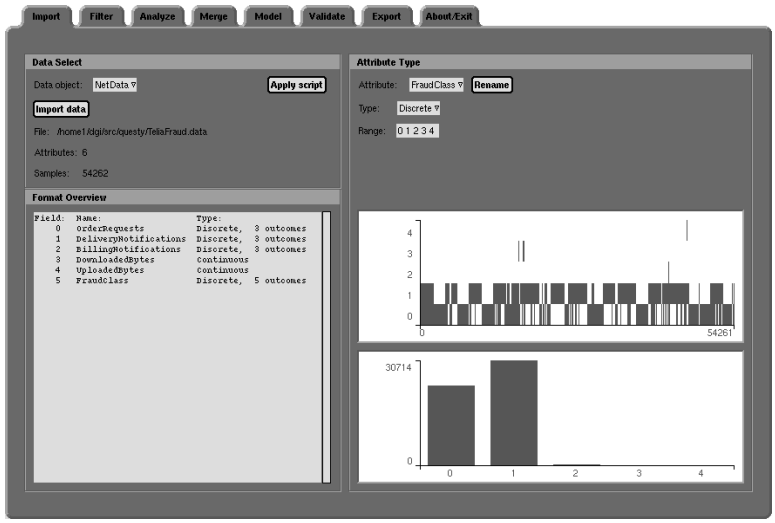

Figure 6.15: Importing data and type analysis.

The interface is divided into a number of sections: *import*, *filter*, *analyse*, *merge*, *model*, *validation*, and *export*. The first four sections, shown in figures 6.15 to 6.18, roughly correspond to the type analysis, repair and recoding, selection and merge phases of data preparation described earlier, but adapted to fit a graphical application. In the import section, the user can import one or several data sets to work with. If the types and format of the data is not known, the application will perform an automatic type analysis and present the result to the user, showing an overview of the types of all fields of a data set to the left and providing detailed information for one field at a time to the right. The data of currently selected field is plotted using the current interpretation of the data to allow the user to assess whether the type is the correct one or not. The types of the fields can be edited freely by the user, allowing the user to test and change the interpretation of the data.

When the types seem to be accurate, the filter section provides an opportunity to visualise and transform the data one field at a time. The data for the selected field

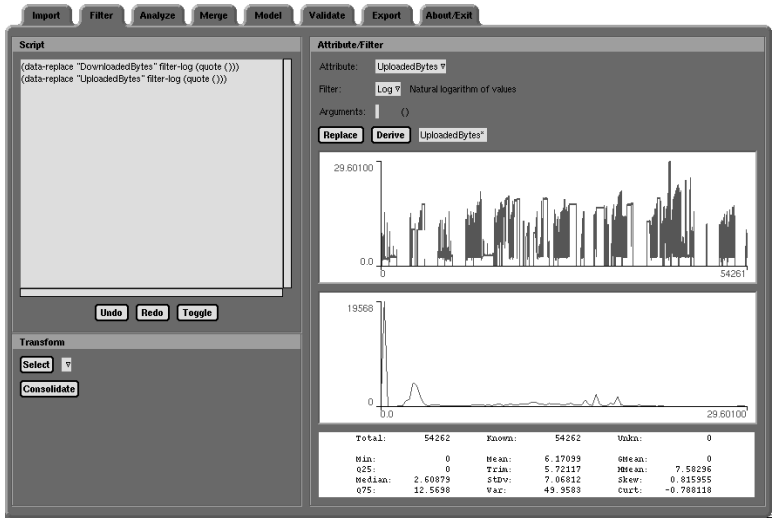

Figure 6.16: Filtering and transformation.

is presented as a series plot suitable for the data type, a histogram, and a collection of descriptive statistics. A collection of common filter functions is available, and the result of applying the transformation can either be derived into a new attribute of the data set, or replace the current one. All transformations that have been performed are listed in editable format to the left, and can be undone, redone, or reordered.

In the analyse section, data is visualised and manipulated not one field at a time, but rather multiple fields to detect correlations, clusters, possibly redundant attributes etc. The visualisations are divided into two categories: pairwise plots, displayed in the upper right of the section, and multiple field plots, displayed in the lower right. A number of different plots are available, and any plot can be shown in a separate window if necessary as well as written to file in a number of common formats. Data can be transformed and attributes and instances can be selected in the lower right section. As in the filter section, all transforms are listed to the upper left.

The merge section is highly simplified in this version, and basically just provides the user with the ability to select two data sets in the left part of the window, and to the right selecting a pre-defined function for merging the two data sets. The set of available merge functions is limited to a number of straightforward transformations and re-sampling functions, but will hopefully be extended in the future.

The modelling section, as shown in figure 6.19, contains facilities for creating statistical models using hierarchical graph mixtures. The model is specified in the left part of the window, while a plot to the right shows a graphical representation of

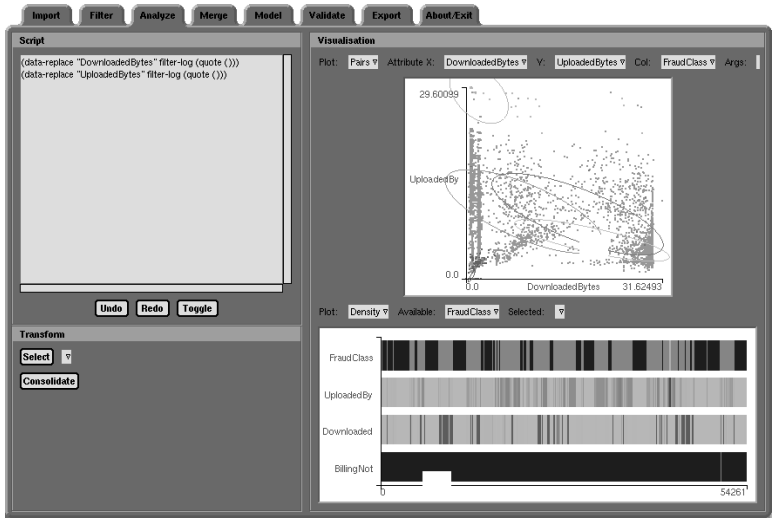

Figure 6.17: Analysis and selection.

the currently selected model. First, we select if the problem is supervised, as in prediction and classification, or unsupervised, as in anomaly detection. We then decide if we would like to model clusters in the data, either in a supervised fashion were one or several attributes in data specify the cluster, or in an unsupervised fashion in which case the a mixture model estimated through Expectation Maximisation will be used.

After this, we specify what type of graphical model we would like to use for each cluster (or for the whole model, if clustering is not selected). The application provides a choice between Naïve Bayes, using the same or separate graphs for each cluster or class, and a quadratic classifier. If separate or the same graph is selected, graphs are generated by calculating pairwise dependencies between all attributes and generating a tree in a greedy manner. The application automatically chooses distribution parameterisations that are suitable for all necessary factors in the graphs.

In the following validation section, shown in figure 6.20, the type of validation scheme can be specified in the leftmost part of the window while test results are shown in the rightmost part. Testing can be performed on a separate data set, or using cross validation. The cross validation procedure can be selected to either n-fold or leave-one-out cross validation. The results are shown in text in the upper right, displaying a multitude of performance measures suitable for the problem and model type, while both the model predictions and the true result can be examined in a scatter plot below.

The final section of the program allows the user to export transformed data

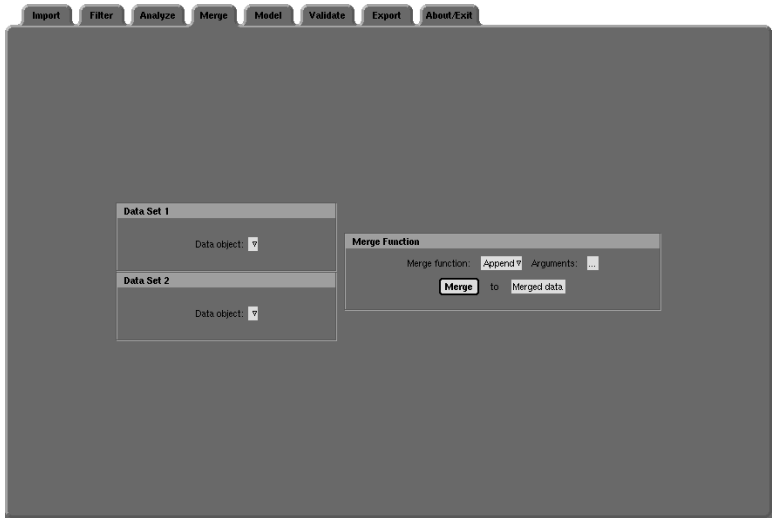

Figure 6.18: Merging data sets.

sets, transformation descriptions, store models, and generate simple applications to perform either a relevant data transformation or a classification, prediction etc. The generated applications can be interacted with either through text from file (standard input and output), or by interprocess communication.

6.6 Conclusions

As developing practical machine learning applications becomes more common, methodology increases in importance. The rather high-level, general descriptions of the process that we have presented helps, but it can only take us so far. Developing data-driven applications is very area dependant, and ideally there should be methodology available for each typical application area, or, at the very least a class of application areas such as process industry or discrete production. This is however way beyond the scope of this text, and is left for future research.

Something that can still help the process significantly is readily available tools and applications for interactive data preparation and modelling suitable for the specific application area. We have here described an example of how to implement such a tool set, allowing for fast specification of data transformation and modelling, but in many situations a tool set that is only usable for this type of interactive data analysis is not enough. An example of this could be an application where the system needs to be fully automatic and data driven, independent of an operator.

Although not necessarily the highest priority while constructing and evaluating solutions, a fast implementation is in fact crucial in many practical situations.

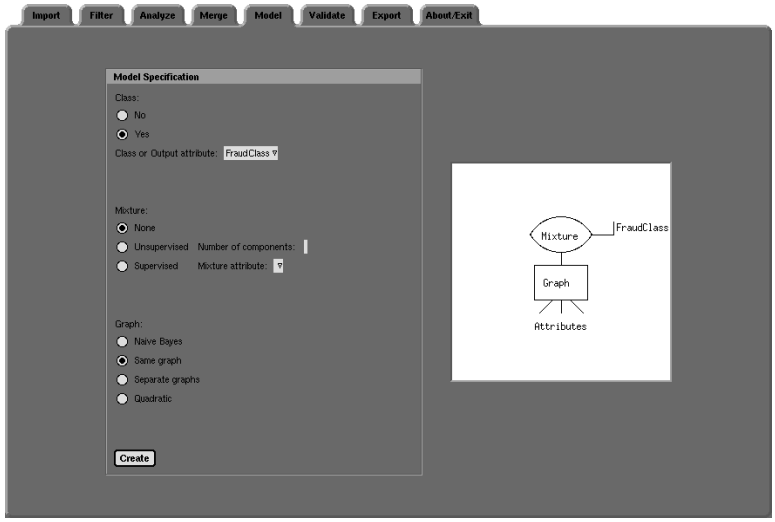

Figure 6.19: Statistical modelling.

A user will only accept a small delay in *e. g.* an interactive diagnosis application before it is regarded as sluggish and unresponsive. A statistical model used to calculate a number of derived values by an optimiser might need to be very fast if we want to arrive at an acceptable solution while it is still valid and usable. This requirement unfortunately rules out most high level languages for implementation of the resource intensive procedures, making their implementation somewhat more difficult. However, when a library of these procedures is implemented and available, accessing them from a high level language allows for simple and fast modelling for the specific application.

It must also be possible to deploy the developed models and data transformations as stand-alone applications or embedded within other systems with relative ease. As it is very difficult to know exactly what systems we would like to interface with in the future and how, it is not possible or perhaps even desirable to construct a large number of interfacing possibilities for a number of standard machine learning applications. Instead, we have opted to create a more self-contained system that can be accessed through a very simple API, intended for rapid construction of specialised applications.

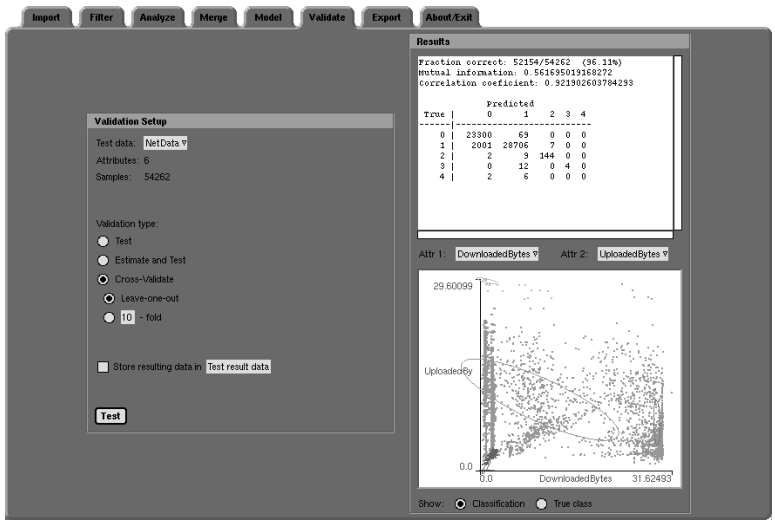

Figure 6.20: Model validation.

Chapter 7

Example Applications

7.1 Examples of Practical Data Analysis

In this chapter we will present a few examples of practical data analysis describe the problems and pitfalls of the application area. The descriptions also serve as examples of practical use of the theoretical models and methodologies presented in earlier chapters, although in all fairness, the experiences from these applications also represent a major influence on both the models and the methodologies. Below is a brief description of the applications we will present.

Sales Prediction Predicting customer demand, manifested as future sales, is an important task for supply chain optimisation. The ability to predict future sales can be used to do more efficient resource allocation and planning, avoiding large stocks and long delays in delivery. Sales prediction is inherently difficult, and can benefit greatly from including other information than historical sales data [Gillblad and Holst, 2002, 2004b]. This application is an example of a fully Bayesian approach to prediction.

Emulating Process Simulators Next, we will explore the possibility of replacing a process simulator with a learning system. This is motivated in the presented test case setting by a need to speed up a simulator that is to be used in conjunction with an optimisation algorithm to find near optimal process parameters [Gillblad *et al.*, 2005]. The presented solution illustrates the importance of suitable data pre-processing and preparation, and compares simple mixtures of graphs to other machine learning approaches.

Prediction of Alloy Parameters The foundry industry today uses software tools for metallurgical process control to achieve less casting defects, a high yield and reduced costs. Using parameters from such a tool that performs thermal analysis,

it is potentially possible to predict properties such as chemical composition and final quality of the alloy [Gillblad and Rögnvaldsson, 2002].

Fraud Detection in an IP Network Media-on-Demand networks that provide *e. g.* movies and television downloads for paying customers are becoming increasingly common. However, these services may invite morally non-scrupulous people to take advantage of the system for their own fraudulent purposes. We will study a couple of approaches for automatic detection of such fraudulent behaviour [Jacobsson *et al.*, 2002]. Both solutions make use of finite mixture models as discussed in chapter 3.

7.2 Sales Prediction for Supply Chains

Introduction

An important component of a supply chain management system is the ability to predict the flow of products through the network and to use this information to avoid large stocks or delays in delivery. While such a tool can potentially be constructed in many ways, the central issue is to predict the future sales of the products. Sales prediction is usually a difficult task, where there might be severe problems finding relevant historical sales data to base a prediction on. Therefore, the use of all relevant information in addition to the historical sales records becomes important. There are several factors that may affect the sales. Most notable are sales activities, advertisements, campaigns and brand awareness of the customers, but also weather, trends, and the emergence of competing products may have a substantial impact. Some of these are very difficult to encode for the use in a prediction system, but others, like brand awareness, marketing investments and sales activities have a clear and direct impact on the future sales while being possible to represent in a reasonable way. Here, some aspects of the forecasting module of the DAMASCOS [Ferreira *et al.*, 2000, 2003] suite is presented, which tries to incorporate this information in the predictions while giving the user information that makes practical usage of the prediction easier. Some important general considerations of sales prediction for supply chain management are also discussed, and some methods for sales prediction are presented briefly.

DAMASCOS is a R&D project supported by the EC, and aims at supporting optimisation and improved information flow in the supply chain. The DAMASCOS software contains modules for sales management, distribution and logistics, supply chain coordination, integration and forecasting. The forecasting module of the DAMASCOS suite, D3S2 (Demand Driven Decision Support System), is a configurable forecasting engine that is integrated with the rest of the DAMASCOS software.

Sales Prediction for Supply Chain Management

Sales Prediction for Decision Support

Sales predictions will most likely be used by sales managers, sales agents, marketing managers and the persons responsible for production planning. A sales manager can use the predictions to get an indication of whether or not the sales target is going to be achieved in order to increase or redirect the sales efforts if needed. The sales agents can use the predictions to *e. g.* promise better delivery times on some products. A marketing manager will be able to use the predictions to early see the effects of marketing strategies and to get an early warning if a change in strategy or additional marketing investments are needed to meet the goals set by the company. The sales prediction is also very useful for the persons responsible for production planning, since capacity can be reserved at the suppliers at an early stage, raw materials can be ordered earlier than otherwise and production scheduled longer in advance.

The sales predictions from the D3S2 module are intended to be used to support manual decisions. In general, sales predictions should not be used in automatic decision processes because of the inherent uncertainty in any prediction method. Only a human operator can decide whether a prediction is plausible or not since it is impossible to encode and use all background information within a prediction system. This means that the design of the module was targeted towards creating a system that delivers as rich and accurate information as possible to its users, both regarding the actual predictions and what they are based on.

In discussions and market surveys, many people express scepticism about sales prediction, comparing it to stock market prediction and questioning whether it is at all possible. Although this is a sound reaction, it builds on the misconceptions that sales prediction generally faces the same problems as prediction of the highly volatile stock market, and that the prediction should provide a highly accurate number of future sold goods instead of being a useful tool that can give an indication of what to expect. This means that a sales prediction module must try to give the user a feel for when to trust the prediction as well as tools that enable full use of this prediction.

Looking at a prediction alone, without any further information, it is very difficult for the user to determine how reliable that prediction is. If an important decision is to be taken partially based on that sales prediction it is necessary to know the accuracy. Therefore, the D3S2 module provides not only the predicted value but also an uncertainty measure to give the user an indication of the expected prediction error.

To further make it easier for the user, the module will provide a short explanation of what the sales prediction is based on and how it has been calculated. If the historical data used to make the prediction would differ greatly from the situation at hand, the user can choose to disregard a prediction that looks good otherwise. The explanation could also help the user decide whether or not to trust the predic-

tion when something out of the ordinary has occurred. Both this explanation and the uncertainty measure are included to increase the users trust in the prediction, and to give a more transparent feel to the module. The user should not be left thinking of the prediction as a black box whose inner workings are concealed, since this greatly reduces the usefulness of the prediction. Also, whenever possible a plot showing the development of the prediction in time as more data becomes available will be presented to the user so that the relevance of the prediction can be more easily evaluated.

Important Considerations

Historical sales data are usually quite noisy and may not be very representative for a company's current and future sales. A prediction model should take this into account, regarding recent sales and marketing events more relevant than older ones, while using as much expert knowledge as possible. The number of parameters in the model must also be kept to a minimum, giving a less accurate prediction on the historical data used for estimating the model but providing for much better generalisation and predictions of future sales.

Additionally, in many cases the rapid changes of a companies range of products gives a need for a human operator to specify relationships to earlier products to make it possible for the prediction to use historical data. When a new article is introduced, the prediction module will benefit from knowledge about what older articles it resembles, *i. e.* what older articles to extract data from and use as the basis for the prediction. The D3S2 module deals with this problem by grouping all articles into *categories* and *segments*. This grouping is made by the user when entering a new item into the database. Categories and segments can be overlapping, and a prediction can be made on all articles, a category, a segment or a combination thereof. No predictions are made on a single article. The user can of course reserve a special category or segment for one article only, but must be aware of the statistical implications of doing so. If the article has been produced and sold for a long time with a decent volume, though, the module will probably have enough historical data to do a reasonable prediction.

Integrating the Customer Demand in the Prediction

Integrating customer demand in the sales prediction can greatly improve the prediction results. Reliable estimation of customer demand is not very easy though. A simple approximation used by a number of media analysis companies is to use the media investments made by the company, which can be viewed as an indicator of future customer demand since all marketing investments generally increase the awareness of the brand. Media investments can be acquired directly from the marketing department of the company. To find the impact of these marketing activities, a method called *brand tracking* can be used. The tracking continuously measures for example advertising awareness and considered brand amongst customers using

telephone interviews with a statistically representative set of people, who answers questions about what brands they have noticed advertising for lately and which trademarks they prefer. If tracking data is available, it is a more reliable indicator of customer demand. It should then be used instead of media investments in the prediction, but both tracking data and media investments can be integrated in the same way in the prediction. Unfortunately, tracking data is not available in many small and medium sized enterprises.

It is very difficult to separate advertising or brand awareness effects on certain articles or groups of articles. The customer demand data must therefore be viewed as having a general impact of the future sales of a brand. In the D3S2 module, customer demand data are used together with historical sales data to estimate the general trend of all sales of a company or brand. This general trend is then used as a basis for all predictions by the module.

The D3S2 Forecasting Module

D3S2 in the DAMASCOS Suite

The D3S2 module is a configurable forecast engine that runs on a centrally located server, most likely at the principal producers headquarter. This forecast engine responds to all prediction requests from all modules within the DAMASCOS suite, making the prediction functionality available to all users of the system without the need to locally store and update the large amount of data used to calculate the prediction. Since prediction is used in several other modules in the DAMASCOS suite, modules often used by different kinds of users, all interaction with the D3S2 module is made through those other modules so that the prediction system is well integrated into the users normal work tool.

Access to constantly updated data is very important to the prediction module, especially when doing short-term predictions. In the DAMASCOS suite, all orders are entered into the SALSA module. All new sales and marketing data that become available are provided without delay by the SALSA module to D3S2. This of course includes new orders and marketing investments, but also information about planned sales activities, start and end dates for sales campaigns and brand tracking results. New orders and sales campaign information are automatically sent to D3S2 as soon they are entered into the SALSA module. Marketing investments and brand tracking results on the other hand are usually kept in a separate system at the company, but the SALSA module will provide interfaces for entering these data as well, not making use of it itself but directly passing it on to the D3S2 module. This design keeps system maintenance costs low, since all users essentially only interact with the SALSA module and there is no need to keep the database of the prediction module up to date separately.

Sales Predictions and the General Trend

Very generalised, two distinct scenarios exist for how customer orders are collected. One is the *sales campaign scenario*, in which sales agents visit a predefined number of potential customers during a sales campaign with a defined start- and end date. The other is the *continuous order scenario*, in which orders arrive continuously at varying rate, either through agents visiting customers to collect orders or from customers that approach the company directly. Many companies operate with a mix of these scenarios. The D3S2 module handles both scenarios, using suitable prediction methods for each case.

Due to the need to use expert knowledge in the sales prediction, the prediction models can be expected to be fairly specialised for each type of company. Below two examples of simple but effective methods for sales prediction are explained briefly. The methods were mainly developed for the pilots industries of the DAMASCOS project; ATECA, an Italian clothes manufacturer that works mostly with season based sales and KYAIA, a Portuguese shoe manufacturer who only uses the continuous order scenario. Although developed for these companies, the methods can be expected to be applicable to all companies operating in a similar way.

Predicting the general trend forms the basis for both season based and continuous order prediction in the D3S2 module. Loosely speaking, the general trend is modelled using linear methods incorporating the sales dependency on time and the media investments or brand awareness at that time. Media investments and brand awareness usually have a time-aggregated effect on the sales, *i. e.* media investments in several weeks usually have larger impact than an investment in just one of those weeks. Therefore, the media investments and brand awareness at a certain point in time are represented as the sum over the last previous weeks with an exponentially decaying weight, lowering the significance of the media investment or brand awareness with time.

Season Based Sales Prediction

To predict how many items that will be sold at the end of the sales campaign when the campaign has just started is obviously very difficult. Many or most of the articles sold during the current campaign might not have been available in the last sales campaign, and most articles that could be bought during the last campaign might be discontinued. Strictly speaking, there is a good chance that we have no relevant statistical information to base our prediction on if we cannot in some way make the approximation that the sales of the current seasons articles will behave like previous seasons articles. A reasonable prediction of the sales, perhaps even the best one, would actually be the last seasons total sales, with compensation for the long term trend. To make a prediction of the sales near the end of a campaign is of course much simpler, since most of the orders will already have been placed by that time.

In some cases, though, we have information that will make the predictions much

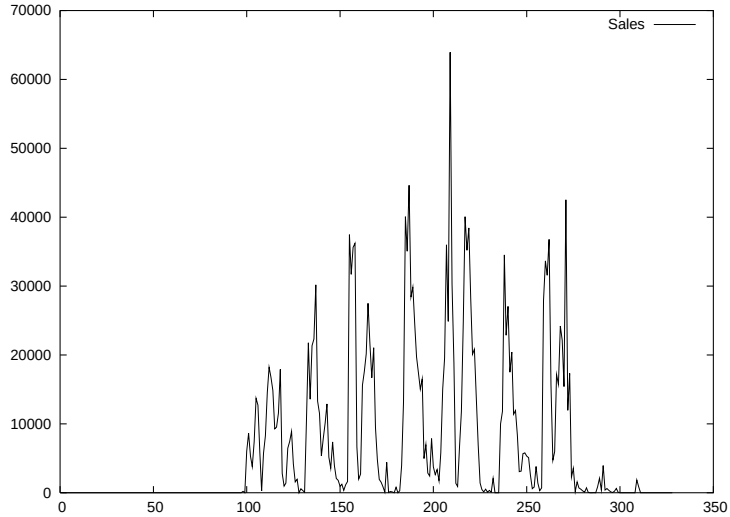

Figure 7.1: Weekly sales during seven seasons, from spring 1997 to spring 2000.

easier. Since there often is a well-defined sales plan, we know how many customers the sales agents will visit. That means, in practise, that we actually know with a rather high certainty how many orders we will get in total during the season, since the mean number of orders from each client is very stable over the years. This information will help us to construct a simple but effective model to handle predictions in the season-based case. Note that the assumption that we fairly well know the total number of orders beforehand is not valid in all season based sales scenarios. It requires that the company operates like the pilot ATECA, with a pre-defined sales plan that is not adjusted too much during the campaign and a customer base that does not show a very erratic behaviour when it comes to the numbers of orders placed. It is likely though that many companies, at least in the clothing business, fit these basic assumptions.

Let us have a closer look at data to explain the situation further. Figure 7.1 show (anonymized) total sales of ATECA on a weekly basis. At first there does not seem to be many invariants in the data. The total sales of each season vary significantly, as do the total number of visited customers and number of orders. The distribution of sales within a season also varies, further complicating matters. However, at a closer inspection, the order size distribution remains rather stable between seasons. Let us now use this fact for our prediction model.

The prediction model itself can be very simple. At a certain point of time t in a campaign, we know the number of orders so far, N_t, and the number of items sold so far, S_t. The prediction should be a sum of the orders we have got so far and the expected sales for the remaining period. Since we assume that we

Figure 7.2: A smoothed histogram of order sizes.

know the total number of orders, we know how many orders that will come in during the rest of the sales campaign, $N_r = N_{tot} - N_t$. This can be used to easily predict the future sales. From the earlier orders, we can estimate a probability distribution over the size of each order, x. Knowing the number of remaining orders, we can estimate the probability distribution of the total remaining sales by simply adding N_r distributions over the size of each order, $P(x)$, together. Now when we have an estimate of the distribution of the remaining orders, we can calculate the expectation and standard deviation of that distribution. These are our sales prediction and uncertainty measure, respectively.

So how do we find the order size distribution $P(x)$? The actual order size distribution can usually be expected to describe a distribution from the exponential family, most likely the gamma distribution. A Gaussian distribution would also be a reasonable fit, since it is approximately symmetric. In fact, the choice between a gamma and a Gaussian distribution is not critical. The attributes of the resulting distribution we are interested in are the expectation and the variance, and the same expressions for summing the distributions are valid both for the gamma and the Gaussian distribution.

Figure 7.2 shows a smoothed histogram of the order sizes. If we ignore the fact that it is truncated at zero, it does in fact look reasonable Gaussian distributed. The tail is somewhat heavier than in a Gaussian, but for this application the approximation should be sufficient.

Here we will use simple Bayesian techniques [Cox, 1946; Jaynes, 1986] to make an estimation of the parameters in the distribution, *i. e.* the expectation and the variance. To start with, we could assume that we more or less know nothing about the initial distributions of the parameters. The assumption that we know nothing at all about the values of the mean and variance of the order size distribution is of course the most careful one; not assuming the data to have any special character-

istics at all. But the fact is, that most of the time we have historical data available that give us valuable prior knowledge about the distribution.

We can use a simple prior information reasoning to find expressions for the expected mean and variance that includes prior information. Let us calculate a prior mean and variance, μ_p and σ_p^2, simply by using the maximum likelihood estimation of the mean and variance of the previous *corresponding* season adjusted with the general trend. If we then estimate the mean and variance in the data we have so far in the current season, μ_t and σ_t^2, we can write the new mean and variance estimates as

$$\mu_x = \frac{\alpha\mu_p + N_t\mu_t}{\alpha + N_t} \tag{7.1}$$

$$\sigma_x^2 = \frac{\alpha\sigma_p^2 + N_t\sigma_t^2}{\alpha + N_t} \tag{7.2}$$

where α denotes a constant that is the weight of the prior. If $\alpha = 1$, then the prior is weighted as one of the data observations, if $\alpha = 2$ it counts as two observations and so on (see section 3.9 for a further explanation of these expressions). Using these estimates the prediction becomes less sensitive to uncommonly large or small orders early in the season. At the end of the season on the other hand, the number of orders N_t is usually much larger than α, so the prior will have little or no effect on the final prediction.

Note that here, the previous corresponding season of for example spring 2000 would be spring 1999, and for winter 1998 it would be winter 1997. The reason for using only the previous corresponding season in the calculations is partly due to changes in available articles and therefore order distribution from year to year, making the previous season more important than earlier seasons, and partly due to the fact that the order distribution usually is quite different between winter and spring seasons.

Now when we have reasonable estimates of μ_x and σ_x^2, we can simply write the total sales prediction S_p at time t as $S_p = S_t + N_r\mu_x$. This prediction model may be simple, but it deals with the fact that we do not have statistical data for all articles sold during the season and is relatively robust to all the noise that is present in the sales data.

However, the prediction model described above has two major flaws. First, it does not provide a reasonable expression of the variance in many situations. Second, it does not explicitly consider time passed since the start of the campaign. This leads to strange effects in the prediction if the sales goals are not exactly met, and the prediction keeps insisting on a higher or lower prediction, with low variance, until the absolute last day of the sales campaign.

To counter this problem, we can extend the model somewhat. We will start by considering the fact that the number of orders varies with how many retailers the company visits from season to season. If we assume that each retailer has a probability q to place an order, we can write the total sales as

$$S_{tot} = \mu_x \cdot n = \mu_x \cdot R \cdot q \tag{7.3}$$

where R is the total number of retailers the company visits, μ_x the mean order size, and n the number of expected orders of the season. This divides the problem into two parts, the random order size which has a relatively invariant distribution between seasons, and the number of orders, which depends directly on how many retailers are visited. Assuming that the probability of a retailer placing an order is stable over the seasons, we can estimate q from previous seasons as

$$q = n_p/R_p \tag{7.4}$$

where n_p is the number of actual orders placed during earlier seasons and R_p the total number of retailers visited during the same period. We can now calculate the expected number of remaining sales of a season and the expected variance using

$$E[S_r] = \sum_n \int_{\mu_r} \mu_r n \cdot P(\mu_r)P(n)d\mu_r \tag{7.5}$$

$$E[S_r^2] = \sum_n \int_{\mu_r} {\mu_r}^2 n^2 \cdot P(\mu_r)P(n)d\mu_r \tag{7.6}$$

$$Var[S_r] = E[S_r^2] - E[S_r]^2 \tag{7.7}$$

where S_r is the remaining sales, n the remaining number of orders, and μ_r the mean of the remaining orders.

To calculate these expressions, we are going to need to specify the distributions over the order size $P(x)$ and the number of orders placed during the season, $P(n)$. Let us start with the distribution over x. Previously, we assumed this distribution to be Gaussian. We will continue to use this assumption, meaning that the mean μ_x and variance σ_x^2 of the order size distribution can be estimated as before in equation 7.1 and 7.2. However, what we actually need is $P(\mu_r)$, *i. e.* the distribution of the mean value of the remaining orders of the season. This distribution is also Gaussian with the same mean μ_x, but with a variance that is n times less, or

$$P(\mu_r) = \frac{1}{\sqrt{2\pi\sigma_x^2/n}} \exp\left(-\frac{(\mu_r - \mu_x)^2}{2\sigma_x^2/n}\right) \tag{7.8}$$

Note that this distribution depends on n, which we need to take into account when we calculate the expectations of equation 7.5 and 7.6.

Let us now turn our attention to the distribution over the remaining orders n of the season. If we knew exactly how many orders we would get during the season this would be trivial, but the number of orders can in fact vary around the expected Rq, and as we get further into the season we can get a better estimation of how many orders we can actually expect. Assume that p, a number between 0 and 1, is the fraction of the season that has passed so far. Denote the number of orders that we have gotten so far as N_t, and the number of orders arriving during the rest of the season as n. There are three possible outcomes for each of the R possible orders. An order has either been placed already, *i. e.* is amongst the N_t orders before p; it

will be placed during the rest of the period, *i. e.* it is part of n; or it will not be placed at all.

We are interested in knowing the probability that n orders will be placed during the rest of the period, given that N_t orders have been placed already. This becomes a binomial distribution, where we distribute $R - N_t$ orders over the order being placed during the rest of the period or not being placed at all, with probabilities $q(1-p)/(1-qp)$ and $(1-q)/(1-qp)$ respectively,

$$P(n) = \binom{R - N_t}{n} \left(\frac{q - pq}{1 - pq}\right)^n \left(\frac{1 - q}{1 - pq}\right)^{R - N_t - n} \tag{7.9}$$

If we now perform the integration in equations 7.5 and 7.6 we get

$$E[S_r] = \mu_x (R - N_t) \left(\frac{q(1-p)}{1 - pq}\right) \tag{7.10}$$

$$Var[S_r] = (R - N_t) \left(\frac{q(1-p)}{1 - pq}\right) \left(\sigma_x^2 + \mu_x^2 \left(\frac{1 - q}{1 - pq}\right)\right) \tag{7.11}$$

These expressions can now be directly used to provide a sales prediction and an estimate of its variance.

This model is more complicated; at least the derivation of it, but it counters most of the problems encountered with the simplistic model and predictions are still easily and very quickly calculated. This is the model actually used in the D3S2 module for season based sales prediction.

Continuous Order Prediction

In the continuous case there is no fixed target for the prediction as in the season-based case. There is no sales campaign that we can predict the total number of orders for. Instead, the user must select a period of interest for which he or she wants a prediction of the sales. To be able to deal with arbitrary prediction periods the model must be very flexible, but the task is much simplified if we can build a reasonable model of the process that results in sales orders at the company. To avoid over fitting problems, this model should have a rather low number of free parameters.

Using the assumption that the sales follow a long-term trend with seasonal variations, which is valid for a wide range of companies, we can construct a simple model for the behaviour of this kind of sales data. The first step is to automatically find the base frequency and phase of the seasonal variations, parameters that are already known in the season based case. Then we must find and model the shape and dynamics of this seasonal variation, using as few parameters as possible to make the model more stable.

In the simplest version, the complete model consists of a base level of sales, the general trend, with an added sinusoidal component representing the basic seasonal variation. There is a very straightforward method of finding the characteristics of

the sine wave. By using the *discrete Fourier transform*, we can transform the sales from the time domain to the frequency domain. Then we can find the strongest low frequency in the data, which probably represents the basic seasonal behaviour we are looking for. The DFT also gives us the phase for all frequencies. In the D3S2 module, the DFT is calculated with the fast Fourier transform, FFT [Brigham and Morrow, 1967] to increase performance. Using the FFT, finding the frequency of the seasonal variations in these relatively small data sets is very fast.

The seasonal trends cannot be expected to be perfectly sinusoidal. We can get a better waveform by using more frequencies, *i. e.* not only using one sine curve but two or more. In fact, if we use all frequencies in the DFT we can of course reconstruct the data perfectly. This is not what we want, however. The reason is that the reconstruction is periodic. If we use all frequencies to reproduce the data, the time period 0 to N, where N is the number of data points, will be perfectly reproduced but the period $N + 1$ to $2N$ will also be an exact copy of the first N weeks. Using this for predictions will simply produce the wrong results. A good compromise is to use only the strongest frequency detected and multiples of that. This means that we can use *e. g.* frequency 8, 16, 24 and so on. Using only multiples of the base frequency will give us a somewhat shaped waveform that is periodic in the base frequency, which is exactly what we want. When more complicated waveforms are necessary to make an accurate prediction, we can benefit from the fact that sales are strictly positive and that we might be able to represent the shape of the seasonal variation as a mixture of Gaussians, positioned using the EM algorithm (see chapter 3).

When we have found the waveform, we need to find its amplitude. Most likely, this amplitude will follow the general trend of the data quite closely, so the best approach is to use a normalised seasonal component that is scaled with the general trend. The resulting model then consists of a base level, the general trend, with an additive waveform. This waveform is found using the DFT and possibly EM, and its amplitude is modulated by a line that also follows the seasonal trend. The model expresses sales as a function of time and media investments or brand tracking. Extrapolating this function into the future, the actual prediction for a specific period of time is calculated as the sum of sales given by the function during that time. This model will produce predictions that are correct on average. This means that a request for a prediction of the sales during only one week or perhaps even one day will usually be off by a relatively large amount, while the variance of the prediction is reduced if it is for a larger time interval.

Test Results

Figure 7.3 shows the prediction results for the spring 2000 season on one particular category of goods, in this case a certain type of pants. The x-axis shows time in days, while the y-axis shows the number of units sold, scaled for anonymization purposes. The graphs depict the development of the prediction over the complete sales campaign, the horizontal line showing the actual total number of orders at

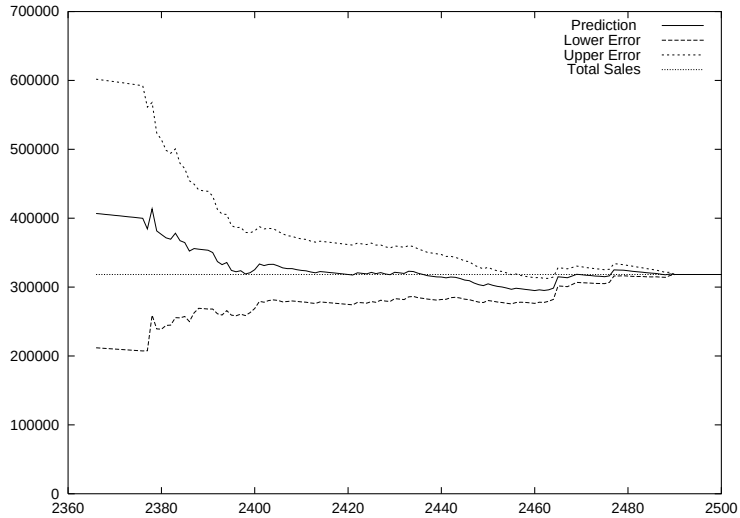

Figure 7.3: Prediction results for the spring 2000 collection.

the end of the campaign. The prediction is plotted for each day, based on the order put so far in the campaign, together with the upper and lower expected prediction error. We can see that the prediction starts out a bit optimistic, predicting total sales quite a lot higher than the correct value. This is because the prediction is in the beginning is only based on the previous corresponding seasons, *i. e.* spring 1999, results. When orders start coming in that reflect the current situation, the prediction adapts rather quickly and the expected margin of error shrinks rapidly. By the time a decision is usually made about reserving capacity in factories, about 5 to 7 weeks into the sales campaign, the prediction is already rather close to the true value. The prediction continues to change in time, dropping slightly to reflect the fact that a number of orders arrive unusually late in the campaign, and finally settles on the correct value when the campaign is over.

The predictions are in general quite useful from rather early on in the sales campaign. Table 7.1 shows the prediction results for a number of seasons at the time of capacity reservation. The errors are expressed as a percentage, where *e. g.* ten percent error means that the system predicted ten percent higher sales at the en of the sales campaign than the correct value. The errors for spring and winter 1997 are quite high due to the fact that there were no earlier corresponding seasons in the data to base the prior distributions on, and the predictions in these cases are only based on orders arrived until that point in the season. The tendency to overestimate the final sales is due to one very large client that puts in large orders at the beginning of the seasons, leading to this slightly over optimistic behaviour of the model. In the other seasons, where prior information is available, this tendency

Season	Error (%)
Spring 1997	21.2
Winter 1997	30.0
Spring 1998	3.9
Winter 1998	7.8
Spring 1999	-3.3
Winter 1999	13.5
Spring 2000	6.1

Table 7.1: Results from predicting the total sales of a season at the time of main resource allocation.

is much less pronounced. In general, the sales estimates produced by the model are certainly close enough to the correct value to be highly usable from both a marketing and production planning point of view.

In figure 7.4, the results of a weekly continuous order prediction during one year is shown. The model is estimated on data up until the year shown, and the models sales prediction for each week is shown along the actual sales of the same week. The model was in this case estimated and tested on sales data from a collection of different types of shoes.

Although orders can arrive at any time, the very strong seasonal effect is clearly visible in the diagram. It is also clear that the error of the predicted sales of a specific week can be relatively high, although it can be reduced greatly by selecting a prediction period longer than one week. Still, the results are such that they provide important information to production planning.

Table 7.2 shows the correlation coefficient between the predicted sales and the actual sales, estimated as before on a few different categories. It is clear that the best prediction on average is performed for all categories together, when the importance of random fluctuations of sales in these categories are reduced. Separating on categories reduces prediction performance, and the level of performance actually follows the size of these categories, the smallest category performing worst. Selecting subsets of these categories is possible, but usually reduces prediction performance to a level where the predicted values cannot be used for planning purposes.

Example Interface

In figure 7.5, an example of an interface to the prediction module is shown. This interface is not integrated with the rest of the DAMASCOS suite, but provides a stand alone interface for those situations where the complete system is not available or necessary. In the top section, information about sales, media investments and seasons are selected and loaded. The module will make do without information on media investments, but without relevant sales data predictions are of course

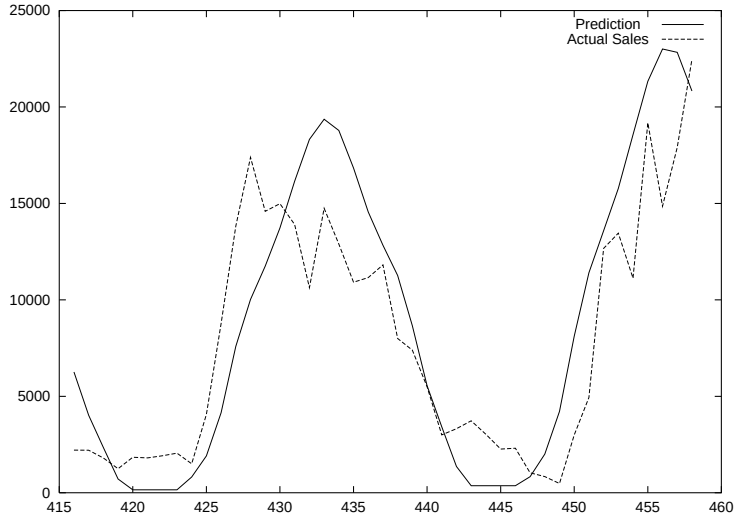

Figure 7.4: The actual and predicted total sales during one year on a weekly basis.

Category	ρ
All	0.86
Casual	0.76
Fashion	0.77
Sports	0.79

Table 7.2: Correlation, measured using Pearson's correlation coefficient ρ, between actual sales and predicted sales in the continuous scenario divided on categories.

impossible.

The middle section allows the user to select the type of prediction to be performed, the segment and category, the target period, and the prediction period. The target period can be set to a certain season, or, in the case of continuous order predictions, an arbitrary time period. The prediction period specifies for which dates the prediction should be calculated, and one prediction for the sales of the target period will be produced using sales information up until each of the dates in this period. This provides the user with the opportunity to visualise how the prediction changes as more data becomes available.

The lower section shows the prediction results of the prediction period as both a graph and in textual format. A comments section provides the user with some information about the assumptions and data used for the predictions.

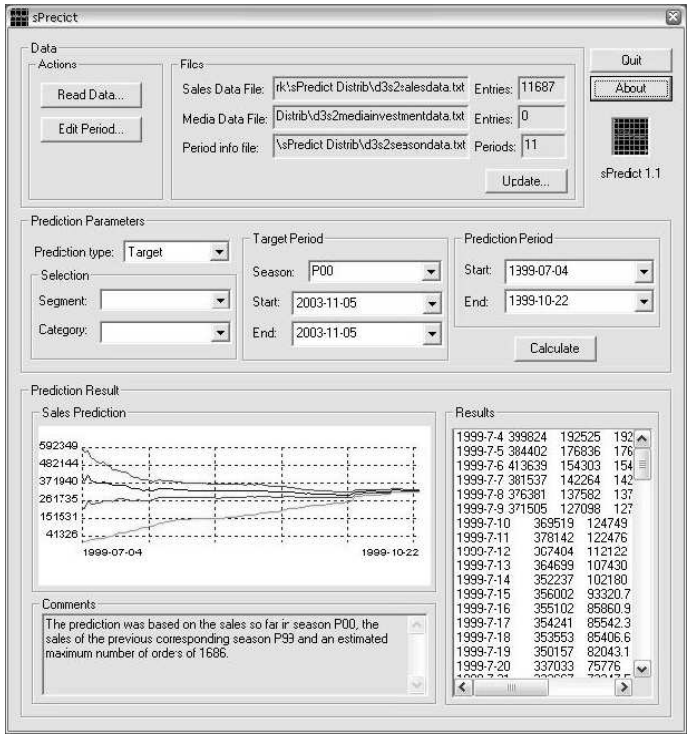

Figure 7.5: A stand-alone interface to the prediction module.

Conclusions

With the increasing demand for both short delivery times and less production for stock, sales forecasting for supply chain management will become more and more important. The work with the D3S2 module within the DAMASCOS suite has shown that by using all available information, it is possible to perform accurate sales forecasting and to provide information to the user that makes the forecasting a very usable tool that fits well into the normal work flow.

7.3 Emulating Process Simulators with Learning Systems

In the process industries there is often a need to find optimal production parameters, for example to reduce energy costs or to improve quality or production speed. Many of the parameter settings are scheduled some time in advance, *e. g.* to produce necessary amounts of different product qualities. A parameter schedule that is sufficiently near optimal as evaluated by a cost function can possibly be found using an optimiser that iteratively tests different scenarios in *e. g.* a first principles

simulator, *i. e.* a simulator that tries to mimic the physical properties of the process, gradually converging to an optimal solution. An initial state for the simulator must be retrieved from the actual process, and the final scheme is suggested to the operators as an effective way to control the real process.

Unfortunately, although the simulator in question is faster than real time, it might still not be fast enough. The number of iterations that is required for the optimisation might easily stretch into the thousands, meaning that even a relatively fast simulator cannot be used to reach a conclusion before the optimisation horizon is well over. If this is the case, some way to speed up the simulations is critical.

Learning Systems and Simulators

Let us consider two fundamentally different ways to model process behaviour. One is to build a first principles simulator of some sophistication. It has to consider how *e. g.* flows, temperatures, pressures, concentrations, etc. varies as material flows through components. The other approach is to approximate the input-output mapping in the process with some mathematical function, *without* considering the actual physical path. This is essentially what is done by a learning system, in which at least a subset of the parameters are estimated from examples.

If we want to replace the simulator with a learning system we have a choice of either modelling the actual outputs of the simulator, *i. e.* training the system to map simulator inputs to corresponding outputs, or to associate simulator states to corresponding objective function values. The latter approach is very elegant and could probably yield very good results, but it is highly dependant on the specific outline of the objective function. In our test case it was not possible to try this direct modelling of the objective function, since all data necessary was not available. Instead, the first approach of mapping simulator inputs, consisting of a description of the initial state and a proposed schedule, to corresponding outputs was used.

We also have to make a choice of either using real process data, or to generate data with the simulator. There are benefits and drawbacks with both approaches, but using a simulator is actually very attractive mainly because of two reasons. First, most simulators are not only free of random measuring noise and drifting sensors, they also lack the stochastic nature of real data in the sense that we do not need several samples from identical input states to reliably estimate the mapping. Second, and perhaps more importantly, is the fact that real processes are kept within only a fraction of the total state space by the operators, following best practises known to produce good results. Most states are simply worthless from the process point of view. Nevertheless, the learning system does not know that these states are worthless unless the training data contain examples showing this, and will probably not produce very realistic results when the optimising algorithm moves out of the region covered by training data.

With a simulator we can cover a larger part of the state space in a controlled manner, but the actual generation of this training data now becomes somewhat of a problem. At first, this might seem like an ideal case: We should be able to generate

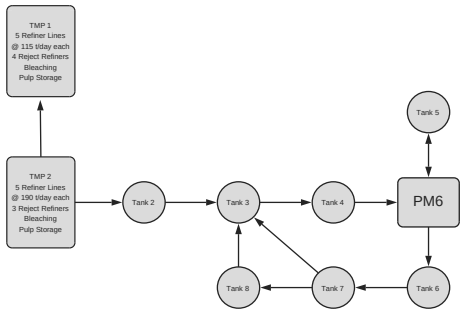

Figure 7.6: A highly simplified overview of the refiner line connected to PM6.

arbitrary amounts of relatively noise free data. Unfortunately, disregarding the effects of finite precision calculations and time resolution in the simulator, there is still one problem remaining: Having a large amount of training data is still effectively useless if it does not reflect the data the learning system will encounter in use. When generating data, we would like to cover as much as possible of all possibly relevant states of the process. Fortunately, this can be rather straightforward using *e. g.* a statistical modelling approach, as we will describe later for the test case.

The Paper Mill Test Case

A simulator of a part of the system at the Jämsänkoski paper mill in Finland was used to test the approach. In the Jämsänkoski mill, thermo mechanical pulp refiners are used to grind wood chips, resulting in pulp that is fed to a number of production lines through a complex system of tanks and filters. There are two separate refiner lines, each consisting of five refiners, and three paper machines, PM4–6. Refiner line one is connected through two production lines to PM4 and PM5, while the second refiner line is only connected to PM6. The second refiner line and production line is largely representative for the whole system and was chosen for the test case. The state of the system is mainly represented by a number of tank levels, and the external control parameters by production rates, refiner schedules and production quality schedules. A simplified overview of the system is shown in figure 7.6.

The cost function for the optimisation problem is constructed so that electricity costs for running the refiners are minimised while maintaining consistency of schedules and tank levels. It can be expressed as

$$C_{tot} = C_E + C_C + C_{\Delta_{tot}} + |V_\Delta| \tag{7.12}$$

where C_{tot} represents the total cost, C_E the cost of electricity, C_C and $C_{\Delta_{tot}}$ consistency terms related to the refiner and set points schedules, and V_Δ the difference between desired and actual tank levels at the end of a schedule. For a further explanation of these terms, see [Gillblad *et al.*, 2005].

To generate training and validation data from the simulator, we modelled the joint distribution of the "normal" conditions over all inputs to the simulator, and then sampled random input values from this distribution. By "normal" states, we refer not only to the states and conditions the process normally encounters in daily use, but rather conditions that do not produce obviously faulty or unacceptable behaviour. This might seem complicated at first, but there are reasonable assumptions that simplify the procedure considerably. As a first approach, we can consider all inputs to be independent. This means that the joint multivariate distribution is simply the product of all marginal distributions. We only have to describe these marginal distributions over each variable, which simplifies the task significantly. Common parameterisations such as uniform distributions, normal distributions etc. were used, preferably making as few assumptions about the variable and its range as possible. When we *e. g.* from explicit constraints know that two or more variables are dependant, we model these using a joint distribution over these variables.

For the external control variables the situation is a little bit more difficult, since we need to generate a time series that to at least some degree reflects a real control sequence in the plant. We model these control sequences using Markov processes, each with transfer probabilities that will generate a series with about the same rate of change and mean value as the real control sequences. Great care was taken to assure that the statistical model and its parameters reflected actual plant operations. In total, around ten million samples were generated, representing about six years of operations.

Test Results

A natural approach to emulate the simulator with a learning system is by step-by-step recursive prediction. However, initial test results using this approach were not encouraging. The state space is large and complex, making the error in the prediction add up quickly and diverging from the true trajectory.

Fortunately, we can actually re-write the data into a form that does not involve time directly. From the cost function we know that we are not interested in intermediate tank levels, but only the final levels at the end of the optimisation horizon. However, we do not want to overflow or underflow any tanks during the time interval, as this disturbs the process and does not produce optimal quality. We also know that apart from the internal control loops, all parameter changes in the process are scheduled. This allows us to re-code the data as *events*, where an event occurs at every change of one of the scheduled external control parameters, *i. e.* one data point describing the momentary state is generated for each external change to the control parameters. If we assume that the process stays in one state, *i. e.* that flows or levels are stable or at least monotonous during the whole event or after a shorter stabilisation period, it should be possible to predict the difference in tank levels at the end of an event from initial levels, flows, quality and perhaps event length.

The previously generated raw data was transformed to this event form and

		Events					
		MLP		k-NN		NB	
Tank	σ	ρ	RMS	ρ	RMS	ρ	RMS
2	22.5	0.50	20.3	0.59	18.2	0.52	19.3
3	16.0	0.70	11.7	0.63	12.4	0.71	11.2
4	15.9	0.48	16.0	0.64	12.1	0.57	13.2
5	14.9	0.34	14.0	0.35	14.0	0.35	13.9
6	15.8	0.61	12.8	0.55	13.2	0.55	13.2
7	22.0	0.54	18.7	0.57	18.0	0.47	19.3
8	19.4	0.69	14.1	0.75	12.7	0.54	16.4

Table 7.3: Results from predicting the difference in tank levels on event data. σ denotes the standard deviation of the data, and results are displayed using both the correlation coefficient (ρ) and the root mean square error (RMS) for the multi-layer perceptron (MLP), k-Nearest Neighbour (k-NN), and Naïve Bayes (NB) models.

three types of learning systems were tested: A Multi-Layer Perceptron trained with backward error propagation in batch mode using sigmoid activation functions [Rumelhart *et al.*, 1986], a k-Nearest Neighbour model using a euclidean distance measure on normalised inputs [Cover, 1968], and a mixture [McLachlan and Peel, 2000] of Naïve Bayes models [Good, 1950], one for each production quality and all using normal distributions for all attributes. One separate model was constructed for each tank. The parameters of each model were chosen experimentally to values producing good results on a separate validation data set. These data were also used to perform backwards selection of the input attributes for each model, resulting in 4 to 10 used inputs out of 10 available depending on model and tank. Testing was performed by predicting the difference in a tanks level at the end of an event from the initial level in the beginning of the event on a test data set separate from the training and validation set. The results can be found in table 7.3. For a detailed description of the models used and the results, see [Gillblad *et al.*, 2005].

The different models' performances are reasonably similar, but not particularly good. Although there might be a reasonable correlation coefficient in some cases, the square errors are still much too high for the predictions to be seen as very useful. Also note that for optimisation horizons consisting of more than one event, which usually would be the case, these errors will accumulate and make the predictions unusable. Since all three types of models perform similarly, we can conclude that the problem probably is ill posed.

So why does it actually go wrong? A closer look at data reveals a likely reason for the poor performance, so let us study an example. Figure 7.7 shows that the tank levels behave very nicely during the first part of the event. Then, one tank overflows and the regulatory systems in the simulator change behaviour. The tank volumes start to oscillate and behave in an unstable manner, the overflow affecting almost the entire tank system. These oscillations in the tank levels are very difficult

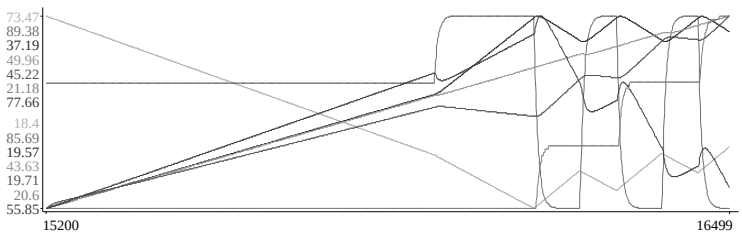

Figure 7.7: An example of the characteristic oscillations of tank levels.

for a learning system to predict, since it in essence has to learn how to predict the phase, frequency and shape of these oscillations. However, we can now actually try to use this observation to change the representation of the problem in a manner that would make it easier for a learning system to solve.

Overflows or underflows are not desired from the cost functions perspective, which means that we have an opportunity to restrict the state space we train the learning system on to data when this does not occur. The model would of course only be valid when there are no overflows or underflows, but since we in practise predict a derivative for the tank levels we can easily estimate whether we would have an overflow or not in any of the tanks during the event. We transformed the data as earlier, but with the exception that an overflow or an underflow of any tank also constitutes the end of an event, although not the start of a new since the process then is in an unstable state. The models used in testing is as before, and the results can be seen in table 7.4. An improvement of the results compared to earlier tests was observed, but some tank levels are still very difficult to predict. The main storage line of tanks 2, 3 and 4 show decent results on the test data set, but the tanks with a direct connection to the paper mill (5 and 6) are very difficult to predict accurately. The differences in these tanks are actually usually zero, only occasionally changing drastically when a control loop needs to use these tanks for temporary storage. Predicting when and to what degree this happens is very difficult.

The differences between the different models performance are again not that high. Usable predictions could possibly be made for the tanks in the main line and some tanks used for temporary storage. However, if the exact levels of the tanks connected more directly to the process itself are necessary, then there is a question of whether the predictions produced by the models are good enough. The correlation coefficient is definitely very low, but the tanks do not usually fluctuate much, meaning that the mean absolute error of the predictions still could be kept rather low.

		Monotonous Events					
		MLP		k-NN		NB	
Tank	σ	ρ	RMS	ρ	RMS	ρ	RMS
2	37.8	0.89	33.3	0.89	17.4	0.71	26.6
3	7.5	0.82	5.18	0.81	4.85	0.84	4.29
4	16.4	0.77	12.5	0.72	12.4	0.53	15.5
5	15.6	0.44	6.66	0.44	15.1	0.33	15.9
6	14.2	0.59	8.13	0.57	11.9	0.51	12.8
7	21.2	0.63	12.8	0.58	16.5	0.44	18.3
8	20.1	0.76	14.9	0.72	14.5	0.51	17.9

Table 7.4: Results from predicting the difference in tank levels on the monotonous event data. As in table 7.3, σ denotes the standard deviation of the data, and results are displayed using both the correlation coefficient (ρ) and the root mean square error (RMS) for the multi-layer perceptron (MLP), k-Nearest Neighbour (k-NN), and Naïve Bayes (NB) models.

Conclusions

The idea of replacing a slow simulator with a faster learning system is certainly attractive. The neural network and Naïve Bayes models are at least 100 to 1000 times faster than the simulator in the test case. However, as the results showed, it is by no means an easy process and not necessarily an effective solution. The generation and representation of data require quite a lot of work, which might easily make it more effective to develop a simpler, faster simulator instead.

It can also be argued that learning systems often are not a suitable solution for approximating a process simulator. The reason is that most "real" processes are described by a system of non-linear differential equations. Such systems will display chaotic behaviour, *i. e.* small changes in input data are quickly amplified, and lose correlation with the input. The time horizon for accurately predicting the output from input data is likely about as short as the time span within which the non-linear differential equations can be approximated by linear differential equations. However, this might not be a problem if we are not interested in the actual output values after a longer time period, but rather a mean value over a certain time or similar.

Even if we need these actual output values, it might still be possible to re-formulate the problem so that it is solvable. It might potentially also be possible to divide the simulator into smaller parts and replacing some or all of these parts with fast learning systems, overcoming the problem of non-linearity for these systems. This division will, unfortunately, require a substantial amount of expert knowledge, as there is no reliable way to perform the division automatically today. Method development might need to get further before we can expect the learning system simulator replacement to become directly viable as an alternative in the general

case.

7.4 Prediction of Alloy Parameters

Thermal analysis studies the solidification from liquid metal into solid iron or an alloy. It is based on recording temperatures at certain time intervals during the solidification progress, and from them constructing a cooling curve. The cooling curve is essentially a plot of the temperature of the metal as a function of time.

By thermal analysis of a sample from the furnace, it is possible to extract information that can be used to predict properties of the alloy produced by the contents of the furnace. In the thermal analysis, a sample from the furnace is cooled and the cooling curve is recorded. Several parameters can then be extracted from the curve, describing important properties of the cooling process. The parameters include, among other things, plateau temperatures and cooling rates in the different states. Using this, information about when different state transitions occur in the furnace can be extracted, which in turn gives an opportunity to predict properties such as chemical composition and final quality of the alloy.

Predicting properties of the final alloy from a sample taken from the furnace is an important task. The ability to make such predictions reliably could potentially help in the reduction of scrap material and defects in the foundry. We have studied the problem on two separate data sets. The focus of the work presented here is on predicting and understanding one of these properties, the oxygen content.

The Thermal Analysis Data

The available data set consists of measurements from two different places in the process, furnace data and ladle data. The furnace data set contain 45 samples, and the ladle data 46 samples. The data sets have been treated as completely separate. Although both data sets contain roughly the same attributes, they must be regarded as behaving significantly different from each other. There are six different kinds of measurements in both data sets:

1. Grey unioculated
2. 12mm cup
3. Grey inoculated
4. Tellurium cup
5. Second grey unioculated
6. Second 12mm cup

The first and second 12mm cups and the first and second grey unioculated measurements are duplicate samples, and should be highly correlated. All of the

Name	Explanation
TL	Liquidus temperature in the cooling curve
TES	Start eutectic cooling
dT/dTES	Cooling rate at the start of eutectic solidification
TE Low	Lower eutectic temperature
TE High	Upper eutectic temperature
R	Recalescence
Max Rate	Max R rate
T1	Temperature 1
T2	Temperature 2
T3	Temperature 3
TL Plata	Liquidus temperature plateau (Only in ladle data)
dT/dtTS	Cooling rate at the solidus temperature
TS	Solidus temperature
GRF2	Grafite factor 2

Table 7.5: The attributes included in all measurements except the tellurium cup.

Name	Explanation
TL	Liquidus temperature in the cooling curve
TES	Eutectic temperature

Table 7.6: The attributes included in the tellurium cup.

different kinds of measurements except the tellurium cup include the attributes listed in table 7.5 along with short explanations of some of them. The tellurium cup measurements contain just 2 attributes. These attributes are listed in table 7.5. All the attributes are continuous.

The data also contain thirty attributes that might be interesting to predict. All the these possible output attributes are continuous, and describe for example chemical composition and hardness. However, the most important output attribute to predict is the oxygen content.

Oxygen Content Prediction using a Mixture Model

One natural approach to predict a continuous parameter, in this case the oxygen content, is to create a mixture of Gaussians over the whole input space and the output space. The mixture model parameters, such as the means and the covariance matrices of the Gaussians are estimated from the training data set, using for example expectation maximisation (EM). The marginal of the output space given an input pattern can then be calculated. When we know the marginal distribution of the output variables, the expectation or the maximum of this distribution can be used as a prediction, depending on whether we want to minimise the mean square

error or the absolute error.

The performance of a mixture model over all input attributes and the output attribute were tested. All the tests used the furnace data set, and all possible input attributes were used, a total of 68. The number of available training samples were 44. The expectation of the resulting marginal distribution for a test pattern was used as the prediction. Table 7.7 *a* and *b* show the test results, table *a* showing the results when the model was tested on training data and table *b* with leave one out cross-validation. With leave one out cross-validation, a model is estimated from all entries in the data except one, which the model is tested on. This is done for all patterns in the data set. The first column in the tables shows the number of Gaussians used in the mixture. The second column the resulting root mean square error (RMS), and the third and fourth column show the fraction of patterns that are within one and three standard deviations of the predicted pattern. The third and fourth column can be viewed as a measure of how many of the predictions that are, in one sense, reasonable. The standard deviation of the oxygen content is 0.26.

The results are reasonably good, both on training and test data. The mean square error is rather low, at least for some of the tested models. It is obvious from the test results that while using only one Gaussian, the mean square error is rather high. On the other hand, for the results with cross validation, the number of patters within one standard deviation is high, suggesting that a simple linear predictor might be sufficient to produce good results if this is considered to be the most important property. Increasing the number of Gaussians though leads to lower mean square errors, and the number of predictions that fall within three standard deviations rise up to 100%, although the number of patterns within one standard deviation is lower. This probably makes for a more practically useful prediction.

Note that the results both on the training set and with cross-validation are not completely consistent, in the respect that the quality of the results do not follow the number of Gaussians in a very organised way. This is due to random effects. When generating the prediction model, the initial model that is trained is generated at random with some considerations to the data.

Dependency Structure Analysis

To gain insight and knowledge of a data set, a dependency structure analysis can be very useful. The dependency graph will show the dependencies between attributes, and what attributes that affect the prediction the most. Even if not used for predictions or some other application, the creation of a dependency graph can still be very valuable for getting a feel for the relationships in the data.

When constructing the dependency graph, we need to keep in mind what we want to use it for. By calculating all pairwise correlations between attributes and then showing the strongest ones in a dependency graph, we might get useful information about the general dependency structure and what attributes that might be redundant. On the other hand, dependencies that might be interesting for a certain task might not be visible using this approach. Often the strongest dependencies

a) Training and testing on all data

Gaussians	RMS	σ	2σ
1	0.35	70.5%	88.6%
2	0.20	77.2%	100.0%
3	0.23	86.4%	100.0%
4	0.20	77.2%	100.0%
5	0.17	86.4%	100.0%
6	0.14	88.6%	100.0%
7	0.18	77.2%	100.0%
8	0.51	81.8%	100.0%

b) Leave one out cross-validation

Gaussians	RMS	σ	2σ
1	0.57	77.2%	84.1%
2	0.35	52.2%	95.5%
3	0.32	59.1%	95.5%
4	0.40	86.4%	95.5%
5	0.32	65.9%	95.5%
6	0.30	50.0%	97.7%
7	0.26	61.4%	100.0%
8	0.27	61.4%	100.0%

Table 7.7: Oxygen content prediction results on furnace data. RMS denotes the root mean square error, σ the fraction of which the true result is within one standard deviation and 2σ within 2 standard deviations.

in data are between similar input attributes, not between input attributes and the output attribute we are interested in. If we want to visualise the dependencies to a specific output attribute, we must use another approach.

The dependency graph in figure 7.8 was generated keeping the relevant output attribute, the oxygen content, in mind. All pairwise linear correlations, *i. e.* the correlation coefficients, between attributes in the ladle data were calculated. Then the strongest correlations between input attributes and the oxygen content were selected, as well as all the stronger correlations between these input attributes. All other correlations were discarded. The graph in the figure was then generated by running a greedy tree construction algorithm on the selected dependencies. The attributes in the graph are denoted with the attribute name and a number in parenthesis describing from what kind of measurement the attribute belongs to (see section 7.4). When studying figure 7.8, keep in mind that the oxygen content has a strong correlation to all the input attributes in the graph. The tree structure shown is most of all an aid to understand the relationship between the input variables.

Two general observations can be made by examining figure 7.8. First, most input attributes shown belong to either Grey unioculated or Second grey uniocu-

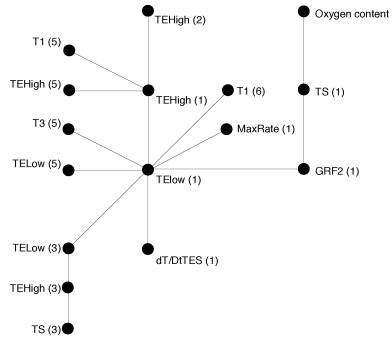

Figure 7.8: Dependency graph for the ladle data

lated. This means that these measurements might contain more useful information about the oxygen content than the other measurements. To be fair though, the grey inoculated is also rather common while the tellurium cup and 12mm cup are barely represented. Second, most attributes are temperatures or the maximum or minimum value of temperatures. This is perhaps not surprising since most of the inputs in fact represent different temperatures, but for example TEHigh and TELow are clearly over represented, indicating that these attributes might be important for the oxygen content.

Also note that after the tree generation, the oxygen content ended up as a leaf, with only one connection. This is a result of the fact that the dependencies between the input attributes are generally stronger than to the oxygen content.

On the whole, though, the dependency graph must be looked upon with some scepticism. The number of examples are low, only 44, while at the same time the data must be considered to be fairly noisy. This means that reliable correlation estimation is difficult, and is also the reason why the simple correlation coefficient was used instead of some more advanced measure more prone to suffer from the low number of available examples. Also, the tree generation algorithm used is sensitive to random effects and noise in the correlation estimations, but it still provides useful information as a suggestion of what the actual dependency structure looks like.

Conclusions

The predictions produced by the very straightforward method of using a Gaussian mixture model over the whole input and output space are promising. The results are reasonably good, and there is probably room for significant improvement using similar but more specialised models. The data set is very small, though, and it is hard to tell whether this data set correctly reflects all the properties of the data.

7.5 Fraud Detection in a Media-on-Demand System

Problem Description

The Swedish telecommunications company TeliaSonera has been performing field tests on a prototype *Media-on-Demand* (MoD) system for distributing movies and television programmes in an IP network. Content can be distributed to the user at any time, a long as a corresponding payment has been registered. As many other digital media distribution systems, it is naturally subjected to fraudulent behaviour from people wanting to take advantage of the system for their own purposes. There is of course a very large number of possible frauds that the system could be subjected to, but for practical purposes a smaller number of well known frauds that are feasible in terms of detectability were selected to be studied in detail. These fraud types are

Illegal redistribution A user downloads movies or other material (in a legal or illegal way) and later redistributes them to other users.

Excess download A user has tampered with the billing system, making it possible to download movies or TV without paying for the service.

Subscription fraud A user registers himself under false identity and manages to use the services of the media-on-demand system without paying for them.

In addition to detecting these frauds, there is an interest in detecting new fraud types. This might be performed by detecting behaviours deviating from normal behaviour, *i. e.* modelling the normal behaviour of the system and perform anomaly detection.

Fraud Indicators

There is a number of possible indicators for the three main kinds of frauds being studied. If the ratio between downloaded and uploaded data volumes is suspiciously high in general, this may be an indicator that the user redistributes movies. If the user is uploading a lot of data at the same time as he/she is downloading a live event, this is an even stronger indicator. If the user is spending a lot of money on an unreasonable number of movies, it is reasonable to believe that the user might be financing this by selling them. Since the redistribution is more likely to be focused on a specific service, an abnormal increase in usage of a service together with an increased traffic is considered as an indicator. If the received and transmitted data are of the same kind (*e. g.* by showing similar statistics or burst patterns), this may indicate redistribution fraud.

Similarly, if more data are transmitted than the user is billed for, excess download is probably in progress; alternatively, a billing system error has appeared. Finally, if a new user deviates substantially from the average new user and uses

services for an excessive amount it may be an indicator that the user ID is used by someone who will not pay for the services.

Not all of the indicators above are directly supported by the available data, but the extraction and interpretation of such indicators belongs to the task that should be solved by the learning system.

The Available Data Sets

As there was limited amounts of real test data available, a simulator in which thousands of users can be simulated in a controlled manner was constructed. Data generated from this simulator was used in conjunction with the real pilot test data to further evaluate the fraud detection mechanisms.

To test the fraud detectors, 518 simulated users for training of which 18 are fraudulent (6 of each fraud type) were created. The same number of users is used for the test set. The simulator was run for 6 months of simulated time, and the fraudulent users where programmed to behave as normal users until they start their fraudulent behaviour some time during the 3 first months. As the simulator does not log *when* a user turns fraudulent, the problem was in practise reduced to classifying a user as fraudulent or not in this data set.

Supervised and Unsupervised Fraud Detection

Fraud detection can be performed in two fundamentally different ways. Either the fraud types are known, and data classified into fraudulent, perhaps with a fraud type tag, or non fraudulent is available. It is then at least hypothetically possible to construct a classifier that will classify new data patterns into fraudulent or non fraudulent, perhaps even with a very simple set of rules specified by someone who knows how these frauds are usually committed. The other possibility is that there is no knowledge about what fraudulent behaviour may look like, or perhaps that we want to be able to detect new types of frauds. This makes the task much harder, since we cannot extract any information on what the frauds might look like from data. Instead, we have to try to build a model of normal, non fraudulent behaviour. When new data patterns deviate significantly from this model, we can classify it as a possible fraud.

In the first scenario, where we have labelled data available for training of a learning system, we can use a supervised learning algorithm to construct our classifier. Supervised learning is only possible when labelled data are available. When this is not the case, we have to rely on unsupervised learning, *i. e.* there are no labelled examples to be learned by the system. This is the situation in the second scenario, where we just want to build a general model of the data that represents normal behaviour. The second scenario can be regarded as generally more difficult and sensitive to the choice of model. On the other hand it is, if it is performing with sufficient precision, perhaps even more useful to a company than a normal classifier trained on labelled data.

The supervised training method uses a mixture of Gaussians, with one Gaussian for each fraud type. The model is trained by estimating each Gaussians parameters on the corresponding data, *i. e.* the Gaussian that represents fraud type one is estimated from all data categorised as fraud type one, the Gaussian that represents fraud type two is estimated from all data belonging to fraud two and so on. When a classification is made, the input pattern is presented to each Gaussian. The corresponding fraud type of the Gaussian with the highest likelihood for the pattern is the classification result.

In the unsupervised case, a mixture of Gaussians is trained on non fraudulent data by the Expectation Maximisation algorithm. The resulting distribution is an estimate of the real distribution of non fraudulent data. If a new pattern has a very low likelihood of being drawn from this distribution, it can be considered to be a possible fraud or at least deviant behaviour.

The Data Used to Train the Models

The pilot test data set did contain frauds, but without any good indication of when and by whom the frauds were committed. The data were therefore labelled for all users and time steps using a simple scheme based on hints given about when the different frauds took place. The data were labelled as

1. Inactive
2. Active
3. Redistribution
4. Excess download
5. Subscription fraud

Patterns were only labelled as frauds at the time they could be considered to be committed, resulting in a rather low number of fraudulent patterns. This and the fact that each type of fraud is committed only by one user makes it impossible to divide the data set into one training set and one testing set for cross validation. One separate data set was constructed though for training of the unsupervised model. This data set contains every second day from all users, with all fraudulent behaviour filtered out.

Not all available data were used for training the models. Only order requests, delivery notification, billing notification, downloaded bytes and uploaded bytes were used, both on pilot test data and on simulated data. In both cases, the uploaded and downloaded bytes were also transformed by the logarithm to the power of three to compensate for dynamics in the data.

In the simulated data case, the amount of data was simply too large to train the models in a reasonable amount of time. Therefore, a subset of the users in the training set were selected for the training data. This subset includes all six

	Inactive	Active	Redistr.	Excess download	Subscription fraud
Inactive	23292	1995	0	0	0
Active	81	28730	5	0	0
Redistribution	0	0	153	0	0
Excess download	0	0	0	4	0
Subscription fraud	0	0	0	0	2

Table 7.8: Supervised prediction results on pilot test data

fraudulent users of each fraud type and 18 randomly selected normal users, *i. e.* the training data set consists of 18 fraudulent users and the same number of normal users. For the unsupervised training, only the non-fraudulent users were used.

Results on Pilot Test Data with Supervised Models

A mixture model was trained supervised over all pilot test data as described above. The model was then tested on the same data set, with the results shown in table 7.8. The rows of the table show the classification made by the model, the columns the true value, and the each value show the number of patterns that fall into that category. Thus, if the prediction was perfect, all values except the diagonal would be zero. From the table we can see that the classifier sometimes confuses active and inactive behaviour. This is not much of a problem in this experiment though, since both types are non-fraudulent. The classifications of subscription frauds and excess download are without errors. Some slight mistakes are made for the redistribution fraud, classifying redistribution as normal active behaviour.

Results on Pilot Test Data with Unsupervised Models

The unsupervised model contained three Gaussians and was trained on every second day of data from all users, with all fraudulent examples removed from the training data set. The reason for training on every second day of data was to test if the model could generalise the information about the normal distribution to the whole data set. The model was then tested on the complete data set, with the results shown in figure 7.9. The upper plot shows the log likelihood of each pattern, while the lower shows the fraud type. The plots show that fraud types 3 and 4 (excess download and subscription fraud) causes significantly lower log likelihood than the non-fraudulent patterns. Redistribution fraud on the other hand does not show any lower log likelihood and could not be detected by the system. This is probably not a problem with the model, but rather a problem with the labelling of the data. The patterns marked as redistribution fraud are based on very basic assumptions that might very well prove to be wrong. In fact, later information suggested that there were probably no real redistribution fraud in the used data set at all.

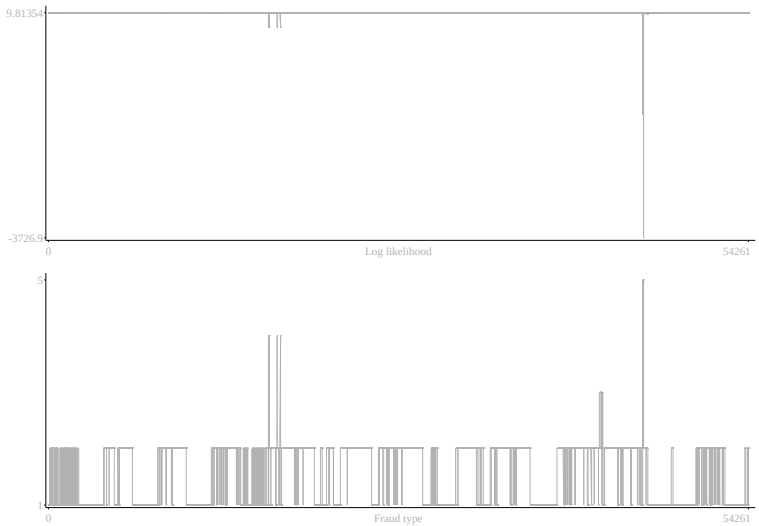

Figure 7.9: Log likelihood and fraud class type for pilot test data

Results on Simulated Data with Supervised Models

A mixture model was trained as described above on the reduced data set. The classification is made for each time step, not one general classification for one user. The results of the classifications are shown in table 7.9. The layout of the table is the same as for table 7.8, except that the active and inactive classes in table 7.8 have been joined to one no fraud class.

If we assume that a user is fraudulent if he or she has been classified as having fraudulent behaviour at any point in time, we can translate the classification per time step to a classification per user. We can then calculate sensitivity and specificity, as described earlier, for all types of fraud. The result is shown in table 7.10.

The table does not show any results for the excess download fraud. It turned out after the experiments that in the test set, all data labelled as excess download frauds were incorrectly labelled and should really belong to the illegal redistribution fraud. The results in table 7.10 compensates for this.

Results on Simulated Data with Unsupervised Models

The unsupervised model again contained three Gaussians and was trained on the selected normal users. The model was tested on the whole data sset and the results are shown in figure 7.10. As before, the upper plot shows the log likelihood and the lower plot the fraud type, here denoted 1 for no fraud and 2 to 4 for the different

	No fraud	Re-distribution	Excess download	Subscription fraud
No fraud	2310670	0	105	56
Redistribution	694	584	2	0
Excess download	0	0	0	0
Subscription fraud	21	0	0	22

Table 7.9: Supervised prediction results on simulated test data

	Sensitivity	Specificity
No fraud	0.96	1.0
Redistribution	1.0	1.0
Subscription fraud	1.0	0.24

Table 7.10: Supervised prediction results on simulated test data, as sensitivity and specificity.

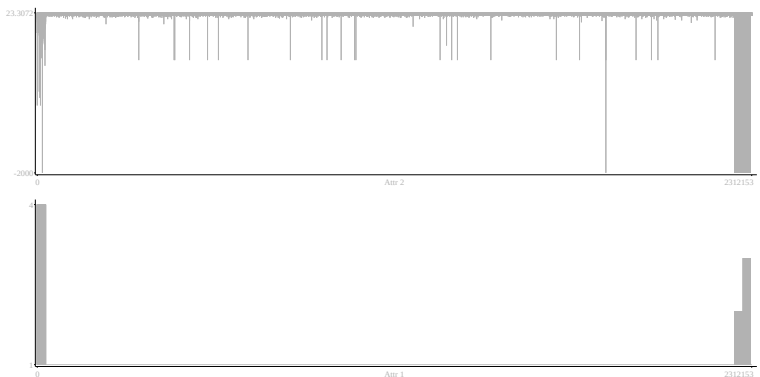

Figure 7.10: Log likelihood and fraud class type for simulated test data

fraud types. All fraud types show significantly lower log likelihood in the model. The model also produces some dips in the log likelihood even when there is no fraud, but they are smaller than the ones produced by fraudulent behaviour. This has one exception though. In the beginning of the last quarter of the data there is one large dip in the log likelihood, but the pattern is not labelled as fraudulent. The reason is that at that time, the user is billed for a movie that has not been ordered or delivered. This makes the pattern stand out from normal behaviour, although it is not the customer that behaves differently but the movie provider.

Conclusions

After all experiments were completed, the apparent failure of the model to separate between Excess download and Illegal redistribution was considered somewhat strange, since there is very little similarity between those two fraud types. This lead to some investigation into the cause of the problem, during which it was discovered that all cases labelled as Excess download actually were generated as Illegal redistribution in the simulator. In the light of these facts, all results are very promising. The fact that some of the data was simulated may be cause for concern, but it is important to remember that real data would most certainly contain the same characteristics as the simulated data.

Chapter 8

Discussion

8.1 Machine Learning and Data Analysis in Practise

After applying several machine learning and data analysis methods to practical problems, often with the direct intent of creating a working application of some sort, we can come to a few general conclusions. They have been stated repeatedly before, both in this thesis and other texts, but their importance should not be underestimated.

The first is that in all data analysis projects data collection, preprocessing, and getting to know the application area and what the problem really is about, almost always take a huge and by far the largest effort. It is also an iterative process that must be repeated a few times before it, hopefully, gets right. Some of the things that often go wrong are related the available data itself. There can *e. g.* be too few effective data samples, or the data have been selected in an unfortunate way. After a few iterations of studying and acquiring new data sets, it may also suddenly turn out that the initial formulation of the problem is not a good one, and that it has to be reformulated.

Another general conclusion is that, once these initial obstacles have been overcome, the exact choice of which learning system to use is often not that important, as many methods perform roughly the same. The hardest part is the preprocessing and representation, and once data have been turned into something reasonable, it may suffice with rather simple methods to solve the actual task. This also means that it often turns out that once the preprocessing is done, the results of a simple linear model was close to those of the non-linear models. This is true partly because, with the limited amounts of independent data that we frequently have to make do with, the number of free parameters has to be kept low.

Finally, the perhaps most important issue is the fact that in practise, each application is unique. New applications often, due to specifics in the application or available data, require specialised solutions and model formulations. This has direct consequences on the possibility to evaluate a certain solution, as the exact problem

formulation never or seldom has been approached earlier. To a certain degree we have to rely on a different evaluation criterion, namely whether the solution is "good enough" to provide performance or financial gains of some sort within its application. If so, our proposed solution has value.

8.2 The Applicability of Data-Driven Methods

Highly data-driven systems are desirable for a number of reasons in many practical applications, often because they potentially can deliver a useful application with a limited investment in describing and understanding the system that generates the data. Unfortunately, this advantage is at the same time often the main drawback when using data-driven systems: The dependency on relevant historical data.

This historical data are vital for model construction and validation. However, the available data are often very scarce, making validation in particular a very delicate process. Data usually also contains large amounts of noise, effectively reducing the number of useful data points available. These problems are common and can sometimes be dealt with using specialised models with relatively few parameters. This approach can unfortunately not help us in another relatively common scenario, where there is absolutely no relevant historical available examples.

This is not quite as uncommon as it may first appear. Quite often, there is a desire to *e. g.* predict or classify parameters relating to a newly designed system that has not yet been taken in use. There is no data relating to it available, and collecting it might not be straightforward. The completely data-driven approach is not useful here, but we would still like to create a model that is useful from the start and that adapts to data as it becomes available.

This means that in practise, there is a need for hybrid data-driven and knowledge based models. By this, we refer to models that are constructed based on available knowledge about a system, *e. g.* through studying its physical properties, but that will still adapt to data when it arrives. These models can potentially be constructed in a number of ways. Bayesian belief networks and other graphical models provide us with the opportunity to encode previous knowledge into the graphical structure of the model, while the parameters of the distributions are free to adapt to incoming data. Bayesian statistics also provide a framework for expressing a previous belief of the properties of a system that is then modified as more data arrives.

Previous knowledge about a system can also be used to construct prototypical data, *i. e.* data that represents not just an outcome, but a typical expression of *e. g.* a certain class. These prototypes can be constructed using known physical properties of the system, available diagnosing diagrams or similar, and then be used for the estimation of a data driven model as in chapter 5. However, the prototypical data does have a slightly different interpretation than ordinary case data, and this should be taken in account when constructing model.

All in all, the creation of hybrid knowledge based and data-driven models, perhaps rather expressed as the incorporation of as much previous knowledge about

a system as possible into the data-driven models, is very important for practical applications. The common lack of relevant data can hardly be solved in any other practical way.

8.3 Extending the Use of Machine Learning and Data Analysis

First, we probably need to accept the fact that for the foreseeable future, most machine learning applications are going to be highly customised for a specific task and application. The day when we can describe to a computer in relatively vague and simple terms what we would like it to perform, leaving the machine to decide how and what data to collect, how to transform it in a suitable way and then learn from it is unfortunately still far away. Therefore, we need to concentrate our efforts not only on creating better learning algorithms but also on supporting the rather tedious task of developing machine learning models for practical use.

In practise, this mainly means suitable methodologies for data transformation and model development, and practically usable tools. We also need to find methods or guidelines for model validation, so that we do not over estimate the actual generalisation performance.

Perhaps somewhat counterintuitive considering our discussion in the section above, is that more effective data driven methods are critical, as this would make model development significantly less labour intense. There may not be enough information available in data for a completely data driven system, but what structure there is we should be able to find and make use of. This includes efficient methods for finding potential clusters and the dependency structure in the data, but also, if possible, the causal directions. Knowing these, we can both create more efficient models and significantly increase our understanding of the system generating the data.

8.4 Final Conclusions

Modern systems, whether it is a communication network, an industrial plant, or even our future homes, are growing extremely complex. There are many potential sources of both faults and disturbances but also gains, often unanticipated during the system design, although the effects of these problems or potentials can be very complex and difficult to recognize or diagnose. In addition, the quality, performance, and environmental demands on how systems should be run are increasing, while disturbances are becoming extremely costly due to just-in-time operations and the degree to which modern society relies on the function of its complex systems. Robustness is rapidly becoming a critical issue in the operation of complex systems.

To gain this robustness, we need *e. g.* systems that autonomously detect deviations from normal behaviour and diagnose and point to the cause of the problem

so that appropriate action can be taken. These tasks, and others, can be solved in natural ways by learning systems. As we can now often access large amounts of data produced by these systems, we also have an opportunity to increase our understanding of them through efficient analysis of these data. Together, this makes the potential impact of machine learning and data analysis in modern systems huge. Creating algorithms and methods for practical applications should therefore remain a worthwhile exercise for the foreseeable future.

Bibliography

Aamodt A. and Plaza E. (1992). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *Artificial Intelligence Communications* **7**: 39–59.

Adriaans P. and Zantinge D. (1996). *Data Mining*. Addison-Wesley, Harlow, England.

Aggarwal C. and Yu P. (2001). Outlier detection for high dimensional data. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pp. 37–46. ACM Press.

Aha D., Kibler D., and Albert M. (1991). Instance-based learning algorithms. *Machine Leaning* **6**: 37–66.

Aji S. M. and McEliece R. J. (2000). The generalized distributive law. *IEEE Transactions on Information Theory* **46**: 325–343.

Akaike H. (1974). A new look at the statistical model identification. *IEEE Transactions on Automatic Control* **19**: 716–723.

Anthony M. and Biggs N. (1992). *Computational Learning Theory: An Introduction*. Cambridge University Press, Cambridge, UK.

Ash R. (1967). *Information Theory*. Interscience Publishers, New York.

Bach F. and Jordan M. (2001). Thin junction trees. In *Advances in Neural Information Processing Systems 14*, pp. 569–576. MIT Press.

Barnett V. and Lewis T. (1994). *Outliers in Statistical Data*. John Willey, NY USA.

Barto A. G., Sutton R. S., and Andersson C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Trans. Systems, Man, and Cybernetics* **13**: 834–846.

Basseville M. and Nikiforov I. V. (1993). *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, New Jersey.

Baum L. E. (1997). An inequality and associated maximisation technique in statistical estimation for probabilistic functions of a markov process. *Inequalities* **3**: 1–8.

Bellman R. (1961). *Adaptive Control Processes: A Guided Tour*. Princeton University Press, New Jersey.

Beni G. and Wang J. (1989). Swarm intelligence. In *Proc. of the Seventh Annual Meeting of the Robotics Society of Japan*, pp. 425–428. Proc. of the Seventh Annual Meeting of the Robotics Society of Japan.

Berry M. J. A. and Linoff G. (1997). *Data Mining Techniques for Marketing, Sales and Customer Support*. John Wiley & Sons, New York.

Bishop C. M. and Svensen M. (2003). Bayesian hierarchical mixtures of experts. In *Proceedings of the Nineteenth Conference on Uncertainty in AI (UAI)*, pp. 57–64. Morgan Kaufmann.

Breiman L. (1994). Bagging predictors. Technical report, Department of Statistics, University of California, Berkely, CA.

Breiman L., Friedman J. H., Olshen R. A., and Stone C. J. (1984). *Classification and Regression Trees*. Wadsworth, Belmont, CA.

Brigham E. O. and Morrow R. E. (1967). The fast fourier transform. *IEEE Spectrum* **4**: 63–70.

Buntine W. (1996). A guide to the literature on learning probabilistic networks from data. *IEEE Trans. Knowledge And Data Engineering* **8**: 195–210.

Burgess C. J. C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery* **2**: 1–47.

Cheeseman P. and Stutz J. (1996). Bayesian classification (autoclass): Theory and results. In *Advances in Knowledge Discovery and Data Mining*, pp. 153–180. AAAI Press.

Chow C. K. and Liu C. N. (1968). Approximating discrete probability distributions with dependency trees. *IEEE Trans. Information Theory* **14**: 462–467.

Clare A. and King R. D. (2003). Data mining the yeast genome in a lazy functional language. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*. Springer-Verlag, London.

Clifton C. and Thuraisingham B. (2001). Emerging standards for data mining. *Computer Standards and Interfaces* 23.

Cochran W. G. (1977). *Sampling Techniques*. Wiley, New York.

Collobert R., Bengio S., and Mariéthoz J. (2002). Torch: a modular machine learning software library. IDIAP Technical Report IDIAP-RR 02-46, IDIAP Research Institute, Martigny, Switzerland.

Cooper G. F. (1990). The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence* **42**: 393–405.

Cover T. M. (1968). Estimation by the nearest neighbour rule. *IEEE Trans. Information Theory* **14**: 50–55.

Cover T. M. and Hart P. E. (1967). Nearest neighbor pattern classification. *IEEE Trans. Information Theory* **13**: 21–27.

Cover T. M. and Thomas J. A. (1991). *Elements of Information Theory*. John Wiley and Sons, New York.

Cowell R., Dawid P., Lauritzen S. L., and Spiegelhalter D. (1999). *Probabilistic Networks and Expert Systems*. Springer Verlag, New York.

Cox R. T. (1946). Probability, frequency and reasonable expectation. *American Journal of Physics* **14**: 1–13.

Cramer N. L. (1985). A representation for the adaptive generation of simple sequential programs. In Grefenstette J. J. (ed.), *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pp. 183–187. Lawrence Erlbaum Associates, Mahwah, NJ, USA.

Crowe C. (1996). Data reconciliation - progress and challenges. *Journal of Process Control* **6**: 89–98.

Dagum P. (1993). Approximating probabilistic inference in bayesian belief networks is np-hard. *Artificial Intelligence* **60**: 141–153.

Date C. J. and Darwen H. (1987). *A Guide to the SQL Standard*. Addison-Wesley, Reading, Mass.

Daubechies I. (1990). The wavelet transform, time-frequency localization and signal analysis. *IEEE Transactions on Information Theory* **36**: 961–1005.

Dempster A. P., Laird N. M., and Rubin D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statistical Society B* **39**: 1–38.

D.J. Newman S. Hettich C. B. and Merz C. (1998). UCI repository of machine learning databases.

Duda R. O. and Hart P. B. (1973). *Pattern Classification and Scene Analysis*. Wiley, New York.

Dudani S. A. (1976). The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man and Cybernetics* **6**: 325–327.

Duraurajan T. M. and Kale B. K. (1979). Locally most powerful test for the mixing proportion. *Sankya B* **41**: 91–1000.

Eskin E. (2000). Anomaly detection over noisy data using learned probability distributions. In *ICML '00: Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 255–262. Morgan Kaufmann Publishers Inc.

Fayyad U., Grinstein G., and Wierse A. (2001). *Information Visualization in Data Mining and Knowledge Discovery*. Morgan Kaufmann, New York.

Fayyad U., Piatetsky-Shapiro G., and Smyth P. (1996a). Knowledge discovery and data mining: towards a unifying framework. In Simoudis E., Han J., and Fayyad U. (eds.), *Proceedings of the Second International Conference on Knowledge Discover and Data Mining*, pp. 82–88. AAAI Press, Menlo Park, CA.

Fayyad U., Piatetsky-Shapiro G., Smyth P., and Uthurusamy R. (1996b). *Advances in Knowledge Discovery and Data Mining*. MIT Press, Cambridge.

Ferreira D., Ferreira H., Ferreira J., Goletz T., Geißler W., Levin B., Holst A., Gillblad D., Ferraz R., Lopes F., Klen N. A. A., Rabelo R., Ericson K., Karlsson J., Medon C., Lugli D., and Nasi G. (2000). A workflow-based approach to the integration of enterprise networks. In *Proceedings of the 16th International Conference on CAD/CAM, Robotics & Factories of the Future (CARS&FOF 2000)*.

Ferreira J. J. P., Antunes N. S., Rabelo R. J., Klen A. P., and Gillblad D. (2003). Dynamic forecasting for master production planning with stock and capacity constraints. In *E-Business Applications*, pp. 377–382. Springer, Germany.

Fisher N. I., Mammen E., and Marron J. S. (1994). Testing for multimodality. *Computational Statistics and Data Analysis* **18**: 499–512.

Fisher R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of Eugenics* **7**: 179–188.

Fowlkes E. B. (1979). Some methods for studying the mixture of two normal (lognormal) distributions. *Journal of the American Statistical Association* **89**: 1027–1034.

Frawley W. J., Piatetsky-Shapiro G., and Matheus C. J. (1992). Knowledge discovery in databases: An overview. *AI Magazine* **92**: 57–70.

Freund Y. and Schapire R. E. (1997). A decision-theoretic generalization of online learning and an application to boosting. *Journal of Computer and System Sciences* **55**: 119–139.

Frey B. J. (1997). Continuous sigmoidal belief networks trained using slice sampling. In *Advances in Neural Information Processing Systems 9*. MIT Press.

Friedman N., Geiger D., and Goldszmidt M. (1997). Bayesian network classifiers. *Machine Learning* **29**: 131–163.

Friedman N. and Koller D. (2003). Being Bayesian about Bayesian network structure: A Bayesian approach to structure discovery in Bayesian networks. *Machine Learning* **50**: 95–125, Full version of UAI 2000 paper.

Gelman A., Carlin J. B., Stern H. S., and Rubin D. B. (2003). *Bayesian Data Analysis*. CRC Press.

Geman S. and Geman D. (1984). Stochastic relaxation, gibbs distributions and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **6**: 721–741.

Gillblad D. and Holst A. (2001). Dependency derivation in industrial process data. In *Proceedings of the 2001 International Conference on Data Mining*, pp. 599–602. IEEE Computer Society Press, Los Alamitos, CA.

Gillblad D. and Holst A. (2002). Sales prediction with marketing data integration for supply chains. In *Proceedings of the 18th International Conference on CAD/CAM, Robotics & Factories of the future (CARS&FOF 2002)*, volume 1, pp. 41–48. INESC Porto, Porto, Portugal.

Gillblad D. and Holst A. (2004a). A brief introduction to hierarchical graph mixtures. In *SAIS-SSLS Joint Workshop*. Artificial Intelligence Group, Department of Computer Science, Lund University, Lund, Sweden.

Gillblad D. and Holst A. (2004b). Sales prediction with marketing data integration for supply chains. In *E-Manufacturing: Business Paradigms and Supporting Technologies*, pp. 173–182. Springer, Germany.

Gillblad D. and Holst A. (2006). Incremental diagnosis for servicing and repair. In *Proceedings of the 16th International Congress on Condition Monitoring and Diagnostic Engineering Management (COMADEM 2006)*, pp. 131–140. Cranfield University Press, Cranfield, UK.

Gillblad D., Holst A., Kreuger P., and Levin B. (2004). The gmdl modeling and analysis system. SICS Technical Report T:2004:17, Swedish Institute of Computer Science, Kista, Sweden.

Gillblad D., Holst A., and Levin B. (2005). Emulating first principles process simulators with learning systems. In *Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005. Lecture Notes on Computer Science*, volume 3696, pp. 377–382. Springer, Germany.

Gillblad D., Holst A., Levin B., and Aronsson M. (2003). An english butler for complex industrial systems. In *Proceedings of the 16th International Congress on Condition Monitoring and Diagnostic Engineering Management (COMADEM 2003)*, pp. 405–413. Växjö University Press, Växjö, Sweden.

Gillblad D., Holst A., Levin B., and Gudmundsson M. (2005). Approximating process simulators with learning systems. SICS Technical Report T:2005:03, Swedish Institute of Computer Science, Kista, Sweden.

Gillblad D., Holst A., and Steinert R. (2006). Fault-tolerant incremental diagnosis with limited historical data. Technical Report T2006:17, Swedish Institute of Computer Science (SICS).

Gillblad D., Kreuger P., Levin B., and Rudström A. (2005a). Preparation and analysis of multiple source industrial process data. Technical Report T2005:10, Swedish Institute of Computer Science (SICS).

Gillblad D., Kreuger P., Rudström A., and Levin B. (2005b). Data preparation in complex industrial systems. In *Proceedings of the 18th International Congress on Condition Monitoring and Diagnostic Engineering Management (COMADEM 2005)*, pp. 543–551. Cranfield University Press, Cranfield, UK.

Gillblad D. and Rögnvaldsson D. (2002). NovaCast: Prediction of alloy parameters. In Holst A. (ed.), *The DALLAS project. Report from the NUTEK-supported project AIS-8: Application of Data Analysis with Learning Systems, 1999–2001*, chapter 6. SICS Technical Report T2002:03, SICS, Kista, Sweden.

Good I. J. (1950). *Probability and the Weighing of Evidence*. Charles Griffin, London.

Hall D. L. and Llinas J. (1997). An introduction to multisensor data fusion. *Proc. IEEE* **85**: 6–23.

Hand D. J. (1981). *Discrimination and Classification*. Wiley and Sons, New York.

Hand D. J. and Yu K. (2001). Idiot's bayes - not so stupid after all? *International Statistical Review* **69**: 385–399.

Hartigan J. and Mohanty S. (1992). The RUNT test for multimodality. *Journal of Classification* **9**: 63–70.

Haykin S. (1994). *Neural Networks: A Comprehensive Foundation*. Prentice Hall, New Jersey.

Heckerman D. E., Horvitz E. J., and Nathwani B. N. (1992a). Towards normative expert systems: part i, the pathfinder project. *Methods of information in medicine* **31**: 90–105.

Heckerman D. E., Horvitz E. J., and Nathwani B. N. (1992b). Towards normative expert systems: part ii, probability-based representations for efficient knowledge acquisition and inference. *Methods of information in medicine* **31**: 106–116.

Hernandez M. A. and Stolfo S. J. (1997). Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery* **2**: 9–37.

Hestenes M. R. and Stiefel E. (1952). Methods of conjugate gradients for solving linear systems. *J. Research Nat. Bur. Standards* **49**: 409–436.

Holst A. (1997). *The use of a Bayesian neural network model for Classification Tasks*. PhD thesis, dept. of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden.

Holst A. and Kreuger P. (2004). Butler, Fallanalys 1 - Outocumpu. Technical Report T2004:07, Swedish Institute of Computer Science SICS. In Swedish.

Holst A. and Lansner A. (1993). A flexible and fault tolerant query-reply system based on a Bayesian neural network. *Int. J. Neural Systems* **4**: 257–267.

Hopfield J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proc. of the National Academy of Sciences of USA* **79**: 2554–2558.

Howard R. A. (1966). Information value theory. *IEEE Transactions on Systems Science and Cybernetics* **SSC-2**: 22–26.

Ihler A. T., Fisher J. W., Moses R. L., and Willsky A. S. (2005). Nonparametric belief propagation for self-localization of sensor networks. *IEEE Journal on Selected Areas in Communications* **23**: 809–819.

Jaakkola T. (2000). Tutorial on variational approximation methods. In Saad D. and Opper M. (eds.), *Advanced Mean Field Methods: Theory and Practice*, chapter 3. MIT Press, Cambridge, CA.

Jacobs R. A., Jordan M. I., Nowlan S. J., and Hinton G. E. (1991). Adaptive mixtures of local experts. *Neural Networks* **10**: 231–241.

Jacobsson H., Gillblad D., and Holst A. (2002). Telia: Detection of frauds in a Media-on-Demand system in an IP network. In Holst A. (ed.), *The DALLAS project. Report from the NUTEK-supported project AIS-8: Application of Data Analysis with Learning Systems, 1999–2001*, chapter 8. SICS Technical Report T2002:03, SICS, Kista, Sweden.

Jain A. and Zongker D. (1997). Feature selection: evaluation, application, and small sample performance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **19**: 153–158.

Jain A. K. and Dubes R. C. (1988). *Algorithms for Clustering Data*. Prentice-Hall, New Jersey.

Jaynes E. T. (1986). Bayesian methods: General background. In Justice J. H. (ed.), *Maximum Entropy and Bayesian Methods in Applied Statistics*, pp. 1–25. Cambridge University Press, Cambridge, MA. Proc. of the fourth Maximum Entropy Workshop, Calgary, Canada, 1984.

Jensen F. V. (1996). *An introduction to Bayesian networks*. UCL Press, London, UK.

Jolliffe I. T. (1986). *Principal Component Analysis*. Springer-Verlag, New York.

Jordan M. I. (1999). *Learning in graphical models*. MIT Press, Cambridge, MA.

Jordan M. I., Ghahramani Z., Jaakkola T. S., and Saul L. K. (1999). An introduction to variational methods for graphical models. *Machine Learning* **37**: 183–233.

Jordan M. I. and Jacobs R. A. (1994). Hierarchical mixtures of experts and the em algorithm of adaptive experts. *Neural Computation* **11**: 181–214.

Kappen B. (2002). The cluster variation method for approximate reasoning in medical diagnosis.

Kappen B., Wiegerinck W., and ter Braak E. (2001). Decision support for medical diagnosis.

Keehn D. G. (1965). A note on learning for gaussian properties. *IEEE Trans. Information Theory* **11**: 126–132.

Kelsey R., Clinger W., and Rees J. (1998). The revised^5 report on the algorithmic language scheme. *ACM SIGPLAN Notices* **33**: 26–76.

Kindermann R. and Snell J. L. (1980). *Markov random fields and their applications*. American Methematical Society, Providence, RI.

Kohonen T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics* **43**: 59–69.

Kohonen T. (1989). *Self-organization and associative memory*. Springer-Verlag, Berlin. 3rd edition.

Kohonen T. (1990). The self-organizing map. *Proc. of the IEEE* **78**: 1464–1480.

Kosko B. (1993). *Neural Networks and Fuzzy Systems*. Prentice-Hall, New Jersey.

Koza J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Kschischang F. R., Frey B. J., and Loeliger H. A. (2001). Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory* **47**: 498–519.

Kullback S. (1959). *Information Theory and Statictics*. Wiley and Sons, New York.

Kullback S. and Leibler R. A. (1951). On information and sufficiency. *Annals of Mathematical Statistics* **22**: 79–86.

Langley P. (1996). *Elements of Machine Learning*. Morgan Kaufmann, San Francisco, CA.

Lauritzen S. L. and Spiegelhalter D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society* **50**: 157–224.

Lawrence J. (1991). Data preparation for a neural network. *AI Expert* **6**: 34–41.

Lee M. L., Lu H., Ling T. W., and Ko Y. T. (1999). Cleansing data for mining and warehousing. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications*, pp. 751–760. Springer-Verlag, London, UK.

Lee W. and Xiang D. (2001). Information-theoretic measures for anomaly detection. In *Proc. of the 2001 IEEE Symposium on Security and Privacy*.

Li W. (1990). Mutual information functions versus correlation functions. *J. Statistical Physics* **60**: 823–837.

Liu H. and Motoda H. (1998). *Feature Selection for Knowledge Discovery and Data Mining*. Kluwer Academic Publishers, Boston.

Louis T. A. (1982). Finding the observed information matrix when using the EM algorithm. *J. Royal Statistical Society B* **44**: 226–233.

Martinez-Bejar R., Ibanez-Cruz F., and Compton P. (1999). A reusable framework for incremental knowledge acquisistion. In *Proc. of the fourth Australian Knowledge Acquisition Workshop*. Proc. of the fourth Australian Knowledge Acquisition Workshop.

McLachlan G. (1992). *Discriminant Analysis and Statistical Pattern Recognition*. Wiley and Sons, New York.

McLachlan G. and Peel D. (2000). *Finite Mixture Models*. John Wiley, New York.

McLachlan G. J. and Basford K. E. (1988). *Mixture Models: Inference and Applications to Clustering*. Marcel Dekker, New York.

Meila M. and Jordan M. I. (2000). Learning with mixtures of trees. *Journal of Machine Learning Research* **1**: 1–48.

Metropolis N., Rosenbluth A. W., Rosenbluth M. N., and Teller A. H. (1953). Equation of state calculation by fast computing machines. *Journal of Chemical Physics* **21**: 1087–1092.

Minsky M. L. and Papert S. A. (1969). *Perceptrons*. MIT Press, Cambridge, MA.

Mitchell T. M. (1997). *Machine Learning*. McGraw-Hill, New York.

Neal R. M. (1993). Markov chain monte carlo methods based on"slicing" the density function. Technical report 9722, University of Toronto, Dept. of Statistics, Toronto, CA.

Parzen E. (1962). On estimation of a probability density function and mode. *Annals of Mathematical Statistics* **33**: 1065–1076.

Pearl J. (1986). Fusion, propagation, and structuring in belief networks. *Artificial Intelligence* **29**: 241–288.

Pearl J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, CA.

Pearl J. (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge.

Pearson K. (1894). Contribution to the mathematical theory of evolution. *Philosophical Transactions of the Royal Society of London* **185**: 71–110.

Piatetsky-Shapiro G. and Matheus C. J. (1992). Knowledge discovery workbench for exploring business databases. *International Journal of Intelligent Systems* **7**: 675–686.

Polymenis A. and Titterington D. M. (1999). A note on the distribution of the likelihood ratio statistic for normal mixture models with known proportions. *Journal of Statistical Computation and Simulation* **64**: 167–175.

PROMEDAS. (2002). PROMEDAS: A probabilistic decision support system for medical diagnosis. Technical report, Foundation for Neural Networks, Nijmegen, The Netherlands.

Pyle D. (1999). *Data Preparation for Data Mining*. Morgan Kaufmann, New York.

Quinlan J. R. (1983). Learning efficient classification procedures and their application to chess end games. In Michalski R. S., Carbonell J. G., and Mitchell T. M. (eds.), *Machine Learning: An Artificial Intelligence Approach*, chapter 15. Tioga Publishing Company, Palo Alto, CA.

Quinlan J. R. (1986). Induction of decision trees. *Machine Learning* **1**: 81–106.

Quinlan J. R. (1993). *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Rabiner L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE* **77**: 257–285.

Reinartz T., Wirth R., Clinton J., Khabaza T., Hejlesen J., Chapman P., and Kerber R. (1998). The current CRISP-DM process model for data mining. In Wysotzki F., Geibel P., and Schädler K. (eds.), *Proceedings of the Annual Meeting of the German Machine Learning Group FGML-98*, pp. 14–22. Germany Technical Report 98/11 Berlin: Technical University, Berlin.

Rish I., Brodie M., Ma S., Odintsova N., Beygelzimer A., Grabarnik G., and Hernandez K. (2005). Adaptive diagnosis in distributed systems. *IEEE Transactions on Neural Networks* **16**: 1088–1109.

Rumelhart D. E., Hinton G. E., and Williams R. J. (1986). Learning internal representations by error propagation. In Rumelhart D. E., McClelland J. L., and the PDP Research Group (eds.), *Parallel Distributed Processing, Explorations in the Microstructure of Cognition*, volume 1, chapter 8. MIT Press, Cambridge, MA.

Russel S. J. and Norvig P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, New Jersey.

SAS Institute Inc. (1998). From data to business advantage: Data mining, the semma methodology and the sas system. SAS Institute White Paper, SAS Institute Inc., Cary, NC.

Sattler K. and Schallehn E. (2001). A data preparation framework based on a multidatabase language. In *Proc. of Int. Database Engineering and Applications Symposium (IDEAS 2001)*.

Schmidt R. and Gierl L. (1997). The roles of prototypes in medical case-based reasoning systems. In *Proc. of 5th German Workshop on Case-Based Reasoning, GWCBR-97*.

Schölkopf B. (1997). *Support Vector Learning*. R. Oldenbourg Verlag, Munich.

Schwartz G. (1978). Estimating the dimension of a model. *Annals of Statistics* **6**: 461–464.

Shannon C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal* **27**: 379–423, 623–656.

Shannon C. E. (1951). Prediction and entropy of printed english. *Bell System Technical Journal* **30**: 50–64.

Simon J. C. (1986). *Patterns and Operators: The Foundation of Data Representation.* McGraw-Hill, New York.

Smyth P., Heckerman D., and Jordan M. I. (1997). Probabilistic independence networks for hidden markov probability models. *Neural Computation* **9**: 227–269.

Spiegelhalter D. J. (1986). Probabilistic reasoning in predictive expert systems. In Kanal L. N. and Lemmer J. F. (eds.), *Uncertainty in Artificial Intelligence*, pp. 47–68. North Holland, Amsterdam.

Stensmo M. (1995). *Adaptive Automated Diagnosis.* PhD thesis, dept. of Computer and Systems Sciences, Royal Institute of Technology, Stockholm, Sweden.

Stensmo M., Lansner A., and Levin B. (1991). A query-reply system based on a recurrent Bayesian artificial neural network. In Kohonen T., Mäkisara K., Simula O., and Kangas J. (eds.), *Artificial Neural Networks*, pp. 459–464. North-Holland, Amsterdam. Proc. of the 1991 International Conference on Artificial Neural Networks, Espoo, Finland, June 24–28, 1991.

Stone M. (1974). Cross-validatory choice and assessment of statistical predictions. *J. Royal Statistical Society B* **36**: 111–147.

Sudderth E., Ihler A., and Freeman W. (2003). Nonparametric belief propagation. In *Proceedings of the 2003 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 605–612.

Sudderth E., Mandel M., Freeman W., and Willsky A. (2004). Distributed occlusion reasoning for tracking with nonparametric belief propagation. In *Advances in Neural Information Processing Systems 17.* MIT Press.

Sutton R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning.* PhD thesis, University of Massachusetts, Amherst, MA.

Swayne D. F., Cook D., and Buja A. (1998). Xgobi: Interactive dynamic data visualization in the X window system. *Journal of Computational and Graphical Statistics* 7.

Teyssier M. and Koller D. (2005). Ordering-based search: A simple and effective algorithm for learning bayesian networks. In *Proceedings of the Twenty-first Conference on Uncertainty in AI (UAI)*, pp. 584–590. Edinburgh, Scottland, UK.

Thiesson B. and C. Meek D. Chickering D. H. (1999). Computationally efficient methods for selecting among mixtures of graphical models. In *Bayesian Statistics 6*, pp. 569–576. Oxford University Press, Oxford.

Thiesson B., Meek C., Chickering D. M., and Heckerman D. (1997). Learning mixtures of bayes networks. Technical Report MSR-POR-97-30, Microsoft Research.

Titsias M. K. and Likas A. (2002). Mixture of experts classification using a hierarchical mixture model. *Neural Computation* **14**: 2221–2244.

Titterington D. M., Smith A. F. M., and Makov U. (1985). *Statistical Analysis of Finite Mixture Distributions*. Wiley and Sons, New York.

Tukey J. and Tukey P. (1990). Strips displaying empirical distributions: I. textured dot strips. Bellcore Technical Memorandum, Bellcore.

Tukey J. W. (1977). *Exploratory Data Analysis*. Addison-Wesley, Reading, MA.

Vapnik V. (1995). *The Nature of Statistical Learning Theory*. Springer Verlag, New York.

Veneris A., J. Liu M. A., and Abadir M. (2002). Inremental diagnosis and correction of multiple faults and errors. In *Proc. of Design Automation and Test in Europe Conference and Exhibition (Date'02)*.

Venkatasubramanian V., Rengaswamy R., and Kavuri S. N. (2003). A review of process fault detection and diagnosis part II: Qualitative models and search strategies. *Computers & Chemical Engineering* **27**: 313–326.

W. R. Gilks S. R. and Spiegelhalter D. J. (1995). *Markov Chain Monte Carlo in Practice*. CRC Press.

Waltz E. and Llinas J. (1990). *Multisensor data fusion*. Artech House, Boston.

Weiss Y. (2000). Correctness of local probability propagation in graphical models with loops. *Neural Computation* **12**: 1–41.

Wichert A. (2005). Associative diagnosis. *Expert Systems* 22.

Wiegerinck W., Kappen H., Braak E., Burg W., Nijman M., and Neijt Y. (1999). Approximate inference for medical diagnosis.

Williams P. M. (1996). Using neural networks to model conditional multivariate densities. *Neural Computation* **8**: 843–854.

Witten I. and Frank E. (1999). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Artech House, Boston.

Witten I. H. and Frank E. (2005). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco.

Zadeh L. A. (1965). Fuzzy sets. *Information and Control* **8**: 338–353.

Zhang S., Zhang C., and Yang Q. (2003). Data preparation for data mining. *Applied Artificial Intelligence* **17**: 375–381.

Ziarko W. (1991). The discovery, analysis, and representation of data dependencies in databases. In Piatetsky-Shapiro G. and Frawley W. J. (eds.), *Knowledge Discovery in Databases*. MIT Press, Cambridge, MA.

Swedish Institute of Computer Science

SICS Dissertation Series

1. Bogumil Hausman, Pruning and Speculative Work in OR-Parallel PROLOG, 1990.

2. Mats Carlsson, Design and Implementation of an OR-Parallel Prolog Engine, 1990.

3. Nabiel A. Elshiewy, Robust Coordinated Reactive Computing in SANDRA, 1990.

4. Dan Sahlin, An Automatic Partial Evaluator for Full Prolog, 1991.

5. Hans A. Hansson, Time and Probability in Formal Design of Distributed Systems, 1991.

6. Peter Sjödin, From LOTOS Specifications to Distributed Implementations, 1991.

7. Roland Karlsson, A High Performance OR-parallel Prolog System, 1992.

8. Erik Hagersten, Toward Scalable Cache Only Memory Architectures, 1992.

9. Lars-Henrik Eriksson, Finitary Partial Inductive Definitions and General Logic, 1993.

10. Mats Björkman, Architectures for High Performance Communication, 1993.

11. Stephen Pink, Measurement, Implementation, and Optimization of Internet Protocols, 1993.

12. Martin Aronsson, GCLA. The Design, Use, and Implementation of a Program Development System, 1993.

13. Christer Samuelsson, Fast Natural-Language Parsing Using Explanation-Based Learning, 1994.

14. Sverker Jansson, AKL - A Multiparadigm Programming Language, 1994.

15. Fredrik Orava, On the Formal Analysis of Telecommunication Protocols, 1994.

16. Torbjörn Keisu, Tree Constraints, 1994.

17. Olof Hagsand, Computer and Communication Support for Interactive Distributed Applications, 1995.

18. Björn Carlsson, Compiling and Executing Finite Domain Constraints, 1995.

19. Per Kreuger, Computational Issues in Calculi of Partial Inductive Definitions, 1995.

20. Annika Waern, Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction, 1996.

21. Björn Gambäck, Processing Swedish Sentences: A Unification- Based Grammar and Some Applications, 1997.

22. Klas Orsvärn, Knowledge Modelling with Libraries of Task Decomposition Methods, 1996.

23. Kia Höök, A Glass Box Approach to Adaptive Hypermedia, 1996.

24. Bengt Ahlgren, Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption, 1997.

25. Johan Montelius, Exploiting Fine-grain Parallelism in Concurrent Constraint Languages, 1997.

26. Jussi Karlgren, Stylistic experiments in information retrieval, 2000.

27. Ashley Saulsbury, Attacking Latency Bottlenecks in Distributed Shared Memory Systems, 1999.

28. Kristian Simsarian, Toward Human Robot Collaboration, 2000.

29. Lars-åke Fredlund, A Framework for Reasoning about Erlang Code, 2001.

30. Thiemo Voigt, Architectures for Service Differentiation in Overloaded Internet Servers, 2002.

31. Fredrik Espinoza, Individual Service Provisioning, 2003.

32. Lars Rasmusson, Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design, 2002.

33. Martin Svensson, Defining, Designing and Evaluating Social Navigation, 2003.

34. Joe Armstrong, Making reliable distributed systems in the presence of software errors, 2003.

35. Emmanuel Frecon, DIVE on the Internet, 2004.

36. Rickard Cöster, Algorithms and Representations for Personalised Information Access, 2005.

37. Per Brand, The Design Philosophy of Distributed Programming Systems: the Mozart Experience, 2005.

38. Sameh El-Ansary, Designs and Analyses in Structured Peer-to-Peer Systems, 2005.

39. Erik Klintskog, Generic Distribution Support for Programming Systems, 2005.

40. Markus Bylund, A Design Rationale for Pervasive Computing - User Experience, Contextual Change, and Technical Requirements, 2005.

41. Åsa Rudström, Co-Construction of hybrid spaces, 2005.

42. Babak Sadighi Firozabadi, Decentralised Privilege Management for Access Control, 2005.

43. Marie Sjölinder, Age-related Cognitive Decline and Navigation in Electronic Environments, 2006.

44. Magnus Sahlgren, The Word-Space Model: Using Distributional Analysis to Represent Syntagmatic and Paradigmatic Relations between Words in High-dimensional Vector Spaces, 2006.

45. Ali Ghodsi, Distributed k-ary System: Algorithms for Distributed Hash Tables, 2006.

46. Stina Nylander, Design and Implementation of Multi-Device Services, 2007.

47. Adam Dunkels, Programming Memory-Constrained Networked Embedded Systems, 2007.

48. Jarmo Laaksolahti, Plot, Spectacle, and Experience: Contributions to the Design and Evaluation of Interactive Storytelling, 2008.