Design and Implementation of a Probabilistic Word Prediction Program

Abstract

Word prediction is the problem of guessing which words are likely to follow a given segment of text. A computer program performing word prediction can be an important writing aid for physically or linguistically handicapped people. Using a word predictor ensures correct spelling and saves keystrokes and effort for the user.

The goal of this project was to design and implement a word predictor for Swedish that would suggest better words, and thus save more keystrokes, than any other word predictor available on the market. The constructed program uses a probabilistic language model based on the well-established ideas of the trigram predictor developed at IBM. By using such a model, the program can be easily modified to work with languages other than Swedish. In tests, the program has been shown to save more than 45 percent of the keystrokes for the user. The report focuses on the technical aspects of designing an efficient algorithm and optimizing it to save as many keystrokes as possible.

Design och implementering av ett probabilistiskt ordprediktionsprogram

Sammanfattning

Ordprediktion är att gissa vilket ord som sannolikt följer en given sekvens av ord. Ett program som utför ordprediktion kan vara ett bra hjälpmedel för motoriskt och språkligt handikappade vid datorarbete och kommunikation. Användningen av ett ordprediktionsprogram underlättar stavning

och besparar knapptryckningar och därmed ansträngning för användaren.

Målet med projektet var att utforma och implementera ett ordprediktionsprogram för svenska som föreslår bättre ord, och därmed besparar användaren fler knapp-tryckningar, än något annat ordprediktionsprogram på marknaden. Det konstruerade programmet använder en probabilistisk språkmodell baserad på den välkända trigram-prediktorn som utvecklats vid IBM. Detta val av språkmodell medför att programmet enkelt kan modifieras för att fungera för andra språk än svenska. Utförda tester visar att programmet kan spara mer än 45 procent av knapptryckningarna för användaren. I denna rapport har tyngdpunkten lagts på de tekniska aspekterna av att skapa en effektiv algoritm och optimering av denna.

Acknowledgments

This paper is a report of my master s project which was done at the Department of Numerical Analysis and Computing Science (NADA) and at the Department of Speech, Music and Hearing (TMH) at the Royal Institute of Technology (KTH). First of all I would like to thank my supervisors associate professors Sheri Hunnicutt at TMH and Viggo Kann at NADA for giving me the opportunity to do this work, and for all their encouragement and support. I am also truly grateful to my girlfriend Ellen Wangler and associate professor Rand Waltzman for inspiration and suggestions.

My thanks also go to my cousin Alice Carlberger, Christina Magnuson, Henrik Wachtmeister, Gunnar Strömstedt, Anders Holtzberg and Johan Bertenstam who all have been participants in the project from time to time. Last, but not least, I want to thank everyone else at TMH for all their help and for making my time at TMH a great and memorable experience.

Contents

2.

Introduction

1. Word prediction

Word prediction is the problem of guessing which words are likely to follow a given segment of text. A computer program performing word prediction is called a *predictor*. To construct a good predictor is, indeed, a difficult task, since natural language is immensely complex and does not lend itself to being described by simple rules. Even the best predictors are easily outperformed by humans. Nonetheless, word prediction as an inherent part of natural language processing has many useful applications in areas such as speech recognition and text proof-reading.

2. Word prediction as a typing aid

A useful application of word prediction itself is as a typing aid for computer users. Figures 1a-c show a typical situation in which a word predictor is used together with a word processor. While the user is typing, the predictor continuously displays a list of words just under the current cursor position. If the intended word is found in the list, the user can press the associated function key, and the word is automatically inserted into the text. The benefits of using a word predictor this way is that it ensures correct spelling and saves keystrokes and effort for the user.

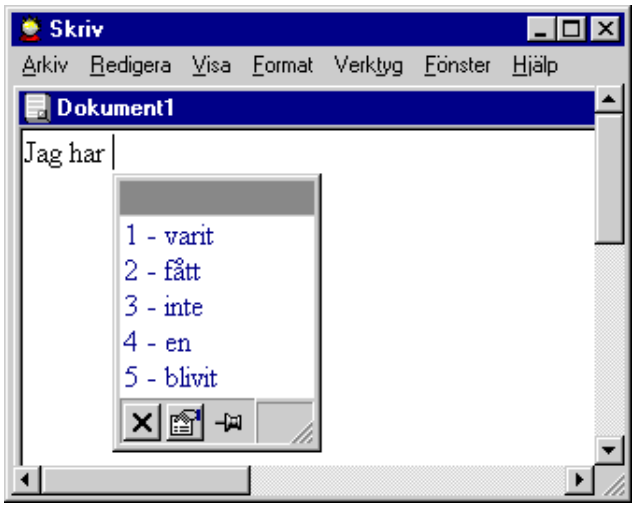

Figure 1a. A word predictor used with a word processor. The predictor suggests five words to follow the typed sequence ***Jag har****.*

3. Word predictor users

There are two main groups of users who may benefit from using a word predictor. The first group is composed of physically disabled individuals who have typing

difficulties. They can save effort, type for longer periods of time, and sometimes speed up their typing by using a word predictor. In many cases, these users have special input devices or user interfaces to suit their needs. The other group consists of linguistically handicapped persons who have difficulties in spelling and composing. Naturally, the levels of difficulties vary within the groups, and some persons belong to both groups.

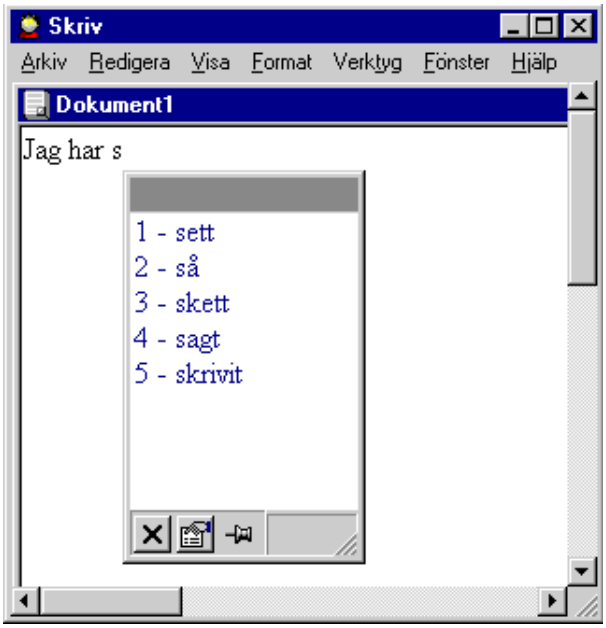

Figure 1b. As letters are typed, the suggestions change to match the first letters of the last word.

*Figure 1c. The user selected the word **skrivit** with the F5 function key. The predictor automatically inserts the word and a space into the text, and then presents five new suggestions.*

In all forms of communication, the information rate with which a person can communicate with other people is very important for the quality of a conversation. A too slow dialog is very stressful and annoying for the participants. Thus, a word

predictor can be an important communication aid when it speeds up typing. Indeed, word prediction programs have proved successful in assisting persons from both groups to communicate and to use word processors [Magnuson, 1994].

All people, however, are by necessity subject to an increased cognitive load when typing with the help of a word predictor, simply because they must inspect the predictor s suggestions. For users with little or no typing or spelling difficulties, the drawback of the increased cognitive load normally outweighs the benefits of saving keystrokes. Thus, most "normal" computer users will prefer not to use a word predictor. Nonetheless, since ten percent of the Swedish population can be classified as dyslexics [von Euler, 1997], the number of possible word predictor users is very large.

4. The Prophet word predictor

A word prediction program for Swedish called *Prophet* has been developed at the *Department for Speech, Music and Hearing* at KTH [Hunnicutt, 1986], and revisions of the program have been made continuously for the last ten years. The program has also been localized into English, Norwegian, Danish and French. In a recent project funded by the National Labor Market Board, the Swedish Handicap Institute, and the National Social Insurance Board, the program has been rebuilt from scratch. However, many of the good solutions of past programs have been incorporated in this version. In this report, I will refer to these two versions of Prophet as the *old* and the *new* version.

My contribution to this project was to design and implement the platform independent part, *Predict*, which manages the language model and performs the actual word prediction. Predict itself does not contain a user interface; it will work together with user interfaces for PC and Macintosh, also developed in the project. It should also be possible to use Predict as a plug-in with other writing-aid programs.

5. Design goals for the new predictor

The major design goal for the new predictor was that it should suggest better words than the old predictor did. When given the word sequence *Det är en lätt up...*, (*It is an easy up...*), the old Prophet program suggests *upp, uppgift, uppgifter, uppfattning* and *Uppsala*, (*up, task, tasks, apprehension, Uppsala* (*a town*)), to complete the uncompleted word *up*. The second word, *uppgift*, is a good suggestion because it well matches *en lätt*. However, when given the similar sequence *Det är ett lätt up...*, the old predictor suggests exactly the same five words. In this case none of the suggestions fit well in the given context.

The reason for giving the same suggestions in these two cases is that the old Prophet program at each prediction simply suggests the most common words. Only when no letters of the last word are yet typed, the predictor considers the previous word, and in this case suggests words which are common successors to the previous word.

It is clear that the new predictor would need a more powerful language model in order to suggest more appropriate words in the previous example. Ideally, the new

predictor must be able to discern grammatically "correct" sequences from "incorrect" ones. Then the predictor should only suggest words which are syntactically and semantically "correct" in the given context. This is, however, a very difficult, if not impossible, problem. Even if the predictor could decide which words are "incorrect" and not suggest these, the user might be deprived of desired suggestions, because most writers do not write correctly all the time. Thus, a word predictor which limits the suggestions to only "correct" alternatives is not only practically unachievable, it is also undesirable. A more preferable and realistic goal is a word predictor which suggests the best fitting words before the more unlikely words. Especially words that are in concord with the previous words should be suggested before words that are not. For example, given the words "*I will*," a predictor suggesting *see* before *saw* is to be preferred before one doing the opposite.

Another important design goal is that the predictor must be dynamic; it should learn new words the user is typing and it should adapt to the user s choice of words. This property is crucial, since the vocabulary and the level of writing skill of different users vary immensely, and one configuration of the predictor cannot possibly serve all users optimally.

6. **Scope of report**

This report will cover the design and implementation of the language model for the new Prophet program. It will focus on the technical aspects, and will only partly cover the design of the user interface and other user-oriented issues. This choice of outline does not reflect the relative importance of these two aspects; it is simply because my work concerns the platform-independent part, Predict, which is separated from the user interface.

This chapter gave a brief description of the word prediction problem and an informal description of the requirements of the word predictor we have designed. The next chapter will cover practical aspects of language modeling for word prediction. Chapter 3 will then describe the language model and the prediction function developed for the new version of Prophet. Chapter 4 concerns the implementation of this language model. The resulting program is then evaluated in Chapter 5, and future improvements are discussed in Chapter 6. The last chapter contains conclusions drawn from this work.

3.

Language Modeling

In the introduction we concluded that we wanted a probabilistic word predictor capable of making high-quality predictions. Any word predictor requires a language model to be able to make predictions. An excellent treatment of language modeling for word prediction in practice was done by a group at IBM for a speech recognition system [Jelinek, 1989]. Their models have formed the basis of the language model we have designed for Prophet.

1. **Basic approach**

The principle of a probabilistic language model is simple: A large training text is used to extract information about the language in question. This information is used by the language model in order to predict successive words, given a sequence of words. The quality of the language model is then evaluated by using it to predict the words of a small test text. The problem is to decide what information to extract, and how the information should be used for making predictions.

2. Practical considerations of language modeling

Suppose we have a language *L* composed of strings of words *w*, and that our word predictor is trying to estimate the probability of the next word, given all previous words of a text, that is

$$P\left(w_i \middle| w_1, w_2, \ldots, w_{i-1}\right) \text{. (1)}$$

We require that the sum of the probabilities over all words is one:

$$\sum_w P(w) = 1 \text{.}$$

Since the number of possible histories, w_1, ..., w_{i-1}, grow exponentially with *i*, most histories will never occur, and very few will occur more than once. Thus, the magnitude and sparseness of the histories make it practically impossible to extract, or store, the probabilities given by (1), even if *L* is small.

A solution to this problem is to define a mapping *S* from the possible histories to a manageable number *M* of equivalence classes, and to approximate (1) by

$$P\left(w_i \middle| S\left[w_1, w_2, \ldots, w_{i-1}\right]\right) \text{. (2)}$$

The simplest way to obtain such a mapping is to consider all histories ending in the same *m* words to be the same. Another way is to use a conventional finite state grammar based on parts-of-speech classifications. As we will see later, such a mapping can be ambiguous, and therefore the probability of the next word is averaged over the possibility that the model was in state *J*:

$$\sum_{J=1}^{M} P\left(w_i \middle| S_{i-1} = J\right) P\left(S\left[w_1, \ldots, w_{i-1}\right] = J\right) \text{. (3)}$$

The probability $P\left(w_i \middle| S_{i-1} = J\right)$ can be estimated by running the training text through the grammar and count the number of times the grammar is in state *J* and the number of times word *w* occurs when the grammar is in state *J*, and use the relative frequency as an estimate:

$$P\left(w_i \middle| S_{i-1} = J\right) = \frac{C\left(w_i, J\right)}{C(J)} \text{. (4)}$$

The choice of the number *M* of equivalence classes will always be a compromise. Choosing only a few classes gives a manageable model, but much information will be lost in the mapping. Employing many classes, on the other hand, preserves more information from the histories, but the higher resolution demands larger training texts to give reliable data.

3. **Evaluation of word predictors**

The concepts of information theory are applicable for evaluating the performance of word predictors. The performance of a word predictor, given a sequence *H* of words, can be measured by quantifying the *surprise* it experiences when the word *w* following the sequence is revealed. The surprise is equal to the amount of information *I* that the revelation of the word gives to the predictor:

$$I = -1\log_2 P(w|H).$$

To see that this measure is adequate, consider the case in which the predictor is certain, beyond any doubt, that the next word will be *w*. Then $P(w|H) = 1$, and thus $I = 0$, that is, if *w* in fact followed *H*, the predictor experienced no surprise at all. On the other hand, if the predictor is equally certain that the word *w* will not follow the sequence, then $P(w|H) = 0$ and $I = \infty$. In this case, if *w*, despite the predictor s conviction, follows*H*, the predictor is said to be "infinitely surprised".

Given a whole text of *n* words, the average surprise that the predictor will experience is equal to the average amount of received information

$$\bar{I} = -\frac{1}{n}\sum_{i=1}^{n} 1\log_2 P(w_i|w_1,\ldots,w_{i-1}).$$

Another related performance measure, proposed by Jelinek, is the *perplexity* of the predictor with respect to a given text:

$$PP = 2^{\bar{I}} = p(w_1,\ldots,w_n)^{-1/n},$$

where

$$p(w_1,\ldots,w_n) = \prod_{i=1}^{n} P(w_i|w_1,\ldots,w_{i-1}).$$

Both the perplexity and the average information are functions of the predictor s estimation of the probability of the text to occur. The perplexity gives the average number of words the predictor must choose between at every prediction.

The perplexity is also related to another performance measure, namely the *rank* of the predictor. The rank *R* is defined as the average number of guesses the predictor needs to correctly guess the following word. The rank turns out to be a monotone function of the perplexity [Hutchens, 1995]. The rank is computed by

$$R = \frac{1}{n}\sum_{i=1}^{n} r_i ,$$

where r_i is the rank of each word.

A common measure for commercial word predictors is the percentage in keystroke savings a perfect user would get from using the predictor. The *letter* keystroke savings in percent, K, can be computed from the rank r_i of each word by

$$K = 100\frac{\sum_{i=1}^{N} \max\left(0, l_i - \left\lfloor (r_i - 1)/F \right\rfloor\right)}{\sum_{i=1}^{N} l_i} , \quad (5)$$

where l_i is the number of letters of the i^{th} word of the text and F is the number of suggestions in the suggestion list. The definitions above make it clear that keystroke savings is a cruder measurement of prediction quality than perplexity, and keystroke savings cannot be said to be a monotone function of $1/R$. Nonetheless, in general it is true that the lower the perplexity, the higher the keystroke savings.

The keystroke savings measure K is the *letter* keystroke savings, because it concerns letter keystrokes only. The *overall* keystroke savings considers all keystrokes, including spaces, punctuation marks, and function keys for selecting words in the suggestion list, as well as letters. Since the predictor (normally) adds a space when a selection is inserted into the text, the user gains one (space key) and loses one (function) keystroke at each selection. Thus the number of saved keystrokes is the same as the number of saved letter keystrokes. This means that we only need to increase the denominator of (5) appropriately when computing the overall keystroke savings.

4. Markov models

A Markov model is an effective way of describing a stochastic chain of events, for example a string of words. Such a model consists of a set of states and probabilities of transitions between them. The last m words in a string are effectively remembered by an m^{th} order Markov model. A simple example of a second order Markov model is shown in Figure 2.

The transition probabilities represent the conditional probabilities for the next word given the previous words. For example, the probability of a transition from state AA to state AB represents the probability that B is written when the two previous words were AA.

Sequences of words extracted from the training texts are called n-grams. In this report, 1-, 2-, and 3-grams are named *uni*-, *bi*-, and *tri*-grams, respectively. The probabilities for the transitions in a second order Markov model can be estimated

simply by counting the number of bigrams and trigrams in the training text, and by using the relative frequency as an estimate. Thus

$$P(w_i|w_{i-2},w_{i-1}) = f(w_i|w_{i-2},w_{i-1}) = \frac{C(w_{i-2},w_{i-1},w_i)}{C(w_{i-2},w_{i-1})}, \quad (6)$$

where C is the count of n-grams. It follows that an m^{th} order Markov model requires m+1-grams to be extracted from the training texts in order to calculate (6). In the example above, the probability of a transition from AA to AB can be estimated by $C(A,A,B)/C(A,A)$.

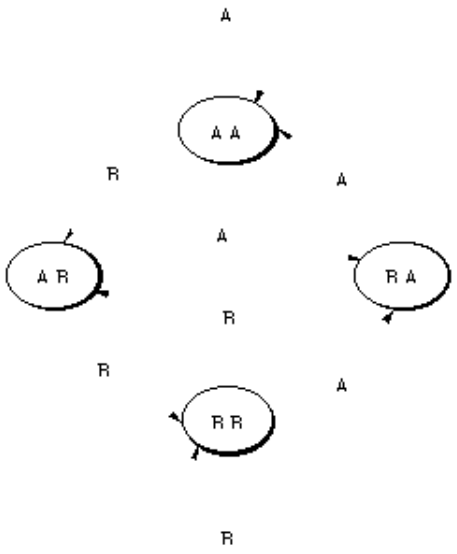

Figure 2. A second order Markov Model for the language {A | B}.*

5. **The interpolated trigram formula**

The trigram formula (6) was used in the language model at IBM, but experiments showed that many trigrams occurring in test texts did not occur in the training text. The consequence was that when such trigrams occurred in a test text, the probability for the third word given by (6) was zero, which is obviously not desirable. Therefore Jelinek proposed the interpolated trigram formula:

$$P(w_3|w_1,w_2) = q_0 f(w_3|w_1,w_2) + q_1 f(w_3|w_2) + q_2 f(w_3), \quad (7)$$

where $f(w_3|w_2)$ is the straightforward bigram variant of (6) and

$$f(w_3) = \frac{C(w_3)}{C},$$

where C is the total word count. The non-negative weights q_i sum up to 1, which holds for all occurrences of q_i in the equations in the rest of this report. Using (7) instead of (6) gave a significant decrease in perplexity for the language model at IBM [Jelinek, 1989].

6. **Limitations of Markov models**

   Naturally a longer scope would yield a better model, but the main limitation of Markov models is that their scope must be very small, otherwise the number of states and transitions would be unmanageable. For example, if the language consists of only a thousand words, the number of possible bigrams and trigrams are one *million* and one *billion* respectively, and although most of them will never occur, the number of *n*-grams quickly grows out of proportion with *n*. Even if it were possible to store all *n*-grams up to a desired *n*, the data would be so sparse that the information would not be useful to compute the corresponding variant of (6) anyway. This is exactly the problem of deciding a suitable value for *M*, the number of mappings of histories into equivalence classes, discussed earlier.

   Using short scope Markov models has obvious consequences. Consider the sentence "*The man who lives on the third floor shot himself yesterday*." The eleventh word *himself* depends on the first two words "*The man*". This dependence can only be captured by an eleventh order Markov model, or better.

7. **Conclusions**

This chapter has made clear that an equivalence classification of the histories of previous words is necessary to get a manageable language model. One such mapping is simply to regard all histories ending in the same *n* words as being equivalent, which enables the use of Markov model probability estimations for the following word. Another way of achieving a mapping is to combine this approach with the use of part-of-speech classification of words, as we will see in the next chapter.

4.

The Language Model of Predict

Markov models as described in the previous chapter will be the basis of the language model developed for the new version of Prophet. This chapter will give a detailed description of this language model. To enable comparisons, the language model of the old version of Prophet will be briefly described first.

1. The old language model

The old predictor worked by employing a main lexicon containing 7 000 words, ranked by commonality. At each prediction, the predictor selected the highest ranked words which matched the initial letters typed, and these words were suggested to the user. The suggestions were sorted by rank and there could be at most nine suggestions in the list.

To make the suggestions fit better with the previous word, the old version had a lexicon with 7 300 bigrams. When a word was completed in the text, and before any letters of the next word were typed, bigram words, if existing, were suggested to the user. As soon as the user typed a letter of the next word, only words from the main lexicon were suggested. There was also a special lexicon with the most common words beginning a sentence, used in the same way as the bigram lexicon.

The predictor learned new words by inserting them into a "subject" lexicon. Then the predictor could suggest these new words in addition to the main lexicon words. Giving the new words a rank of 1 000 ensured that these words were suggested at a preferable rate.

2. The new language model

The old language model gave fairly good suggestions, but often they did not match the context very well, as in the example given in the introduction. This is obvious; the predictor considered only the previous word when no letters of the next word were yet typed. When letters of the next word were typed, the previous word was disregarded and consequently the predictor simply suggested the most common words.

The new language model will always consider the previous word when predicting the following, even after letters of the following word have been typed. Even better would be to consider the two previous words, but if a Markov model is used, it would require trigrams to be extracted from the training text. As mentioned, the number of trigrams is enormous, and moreover, it seems that many will be redundant. Consider all sequences like "*en lätt uppgift*", "*en svår uppgift*", "*någon svår uppgift*" ("*an easy task*", "*a difficult task*", "*some difficult task*"). After the two first words we want our predictor to suggest *uppgift* (*task*) before *uppgifter* (*tasks*), but to do so, it suffices for the predictor to know that the two previous words were a determiner for singular and an adjective. This can be accomplished by using a mapping from words to a much smaller number of part-of-speech classes. The number of part-of-speech trigrams will be manageable, at the cost of lost information.

1. Using part-of-speech tags

To get part-of-speech information as in the example above we need a training text where the words are *tagged*, which means that they are classified according to their part-of-speech. "*This* <article> *is* <verb> *a* <article> *tagged* <adjective> *sentence* <noun>." The process of assigning tags to the words of a text is called *tagging*, and a computer program tagging a text is consequently called a *tagger*. Much research about tagging is currently being done in the field of natural language processing.

For this project, we have access to a small, tagged corpus (text collection) used to train a tagger currently being developed at the Universities of Stockholm and

Umeå [Ejerhed, 1992]. The tag set consists of roughly 150 part-of-speech tags. The level of detail of these tags is illustrated by nouns, which are subdivided according to gender, plurality, definiteness and case. For example, the word *åker* (*crop field*) is tagged <*noun neutre singular indefinite*>.

Since many words in natural language have different grammatical functionality depending on the context, tagging cannot be done unambiguously before a sentence has come to a full stop, even by a perfect tagger. For example, the word *åker* above, can also mean *travel*, and is in this case tagged <*verb present tense*>. Consequently, no tagger can unambiguously tag a sentence starting "*En åker"* (*One travels / One field*) without information about the next words, since both meanings of the word *åker* are still possible.

The tagged corpus consists of one million words, but given the large number of tags, this is an insufficient amount of text to obtain very reliable statistics from. For example, the average tag trigram count of occurring trigrams is a mere 12.8. Nonetheless, we have used this corpus for the language model, since presently, it is the only Swedish tagged corpus available.

2. **Two Markov models**

The new language model of Predict is based on two Markov models; one for words and the other for tags. The two models interact, but the separation enables the predictor to work with lexicons of either tagged or untagged words, without any changes to the program. This will facilitate future localizing of the new Prophet into languages other than Swedish, for which no tagged texts may be available.

We have chosen to employ a second order Markov model for tags and a first order model for words, simply because they require a reasonable amount of implementation work and storage space. Furthermore, this choice assures that the resulting language model will be superior to the one in the old version of Prophet. Future extensions of the scopes are possible and would probably yield a better model.

The idea of this scheme is to first obtain a probability estimation for the tag of the next word, using the tag Markov model, and then use the word Markov model to get a probability estimation for the next word. In the second step, the tag probability estimation is taken into account, in order to promote words with have a likely tag according to the tag Markov model.

3. **The tag model**

The second order Markov model for tags uses the interpolated trigram formula (7) to estimate the probabilities of the next tag:

$$P(t_3|t_1,t_2) = q_0 f(t_3|t_1,t_2) + q_1 f(t_3|t_2) + q_2 f(t_3) \quad (8)$$

Since the tagging performed by the predictor is ambiguous, the state of the Markov model for tags is also ambiguous, and (8) is therefore averaged over the

probabilities of the current state of the model, using (3).

When a word *w* is completed and the model makes a transition, there are two cases: the word *w* is either a known or an unknown word. In the first case, if *w* is known, the estimation from (8) can be modified, since after the revelation of *w*, new information is available. Thus a new tag probability estimation for the tag of *w* can be computed by:

$$P(t_3) = q_0 P(t_3|t_1,t_2) + q_1 P(t_3|w) \text{ . (9)}$$

Equation (9) shows that the tag for the completed word *w* is given by the weighted sum of the probability for *t*, given by the tag model, and the conditional probability that word *w* will have tag *t*. This conditional probability is obtained by

$$P(t|w) = \frac{C(w,t)}{C(w)} \text{ , (10)}$$

where $C(w,t)$ is the number of times word *w* was tagged with tag *t* in the training text.

In the other case when *w* is unknown, the predictor has no information about this word, and thus no counts *C*(*w*, *t*) are available. This means that the second term of (9) cannot be computed for this word the first time it occurs. But, the probability estimation obtained by (8) can be used to get preliminary counts *C*(*w*, *t*) for this word for future occurrences. A detailed description of how unknown words are treated follows later in this chapter.

4. **The word model**

The first order Markov model for words uses the interpolated bigram formula to estimate probabilities of the next word:

$$P(w_2|w_1) = q_0 f(w_2|w_1) + q_1 f(w_2) \text{ . (11)}$$

The probability distribution for the next tag, computed by (8), is used to get two additional terms to (11):

$$P_t(w_2|w_1) = q_0 f(w_2|w_1) + q_1 f(w_2) + q_2 f(w_2|w_1) p_t(w_2) + q_3 f(w_2) p_t(w_2) \text{ , (12)}$$

where

$$p_t(w_2) = P(t_2|w_2) P(t_2|t_0,t_1) = \frac{C(w_2,t_2)}{C(w_2)} P(t_2|t_0,t_1) \text{ .}$$

Thus $P_t(w_2|w_1)$ is the probability of the next word, given the previous word and the probability estimation of the tag of the next word. Since this estimation is a

probability distribution over the tag set, the last two terms are averaged over all tags using (3).

5. **Punctuation marks are treated as words**

   An improvement from the old version of Prophet is that punctuation marks: periods, commas, colons etc., are treated as words by the model. This simplification means that punctuation marks may be parts of *n*-grams, and thus no special rule for the beginning of a sentence, as the old version had, is needed. Furthermore, special tags are associated with punctuation marks, which makes them a natural part of the tag model. This assures that the predictor s suggestions after punctuation marks in the text are automatically optimized without extra work.

6. **Finding optimal weights for the equations**

   The optimal values for the weights q_i in equations (8), (9) and (12) can be found by an iterative, statistical method described in [Jelinek, 1989]. These values will be optimal with respect to the test text used. Within this project, we have decided to optimize the weights simply by testing different values in order to maximize the keystroke savings.

   It is worth noting that if a term in these equations does not contribute positively to the performance of the predictor, the value for the corresponding weight will approach zero, and the term will not affect the sum. Hence, there is no danger in introducing additional terms to the equations. The worst that can happen is that the added terms will have no effect.

7. **The prediction function in practice**

   In this and the following chapters, a *prefix* of a word will refer to the first *n* letters of the word; it can be the empty string and it can be the whole word. For example, the prefixes of *jag* are *""*, *"j"*, *"ja"* and *"jag."* When the user starts to type a word, the predictor regards the sequence of letters as a prefix. Not until a prefix is completed during typing, either by a space or a punctuation mark, the prefix will be regarded as a word.

   The *prediction function* takes as input the current context, the cursor position, and an integer *n* which is the desired number of suggestions wanted by the user. The output is the *n* most probable words which match the prefix at the cursor position.

   As a prefix is completed during prediction, a new word is encountered, and the predictor makes a transition to a new state. After the transition, the predictor computes the probabilities for all words to occur next, using (12), and returns the best *n* words for suggestion. Now, if the user types the initial letter of the next word, the predictor looks up the best *n* words with matching initial letter and returns these words. As long as the user types additional letters of the word, the predictor continues to suggest words matching the prefix. Since the probabilities of all words are already computed, the selection process will now only involve finding the best words matching the prefix.

8. Managing unknown words

The words known to the predictor when a session starts have much statistical information associated with them. There are counts for unigrams and bigrams and there are counts for how many times each word has been tagged with different tags in the training text. Obviously, no such information is available for the new words which are encountered during a prediction session. The predictor must therefore try to extract this information for new words as they are used, just as information was once retrieved from the training text. However, the problem is that we do not want to change the database constructed from the training text, since it would be very troublesome to keep it consistent. New words will therefore be stored separately from the pre-known words of the database. This led to the decision to limit the amount of information associated with new words.

The only information associated with each new word w is the frequency $C(w)$ and one tag t, which means that $C(w, t) = 1$ for one t and 0 for all others. Since no new words are parts of bigrams, the interpolated bigram formula (11) will be reduced to a unigram formula, and hence the model cannot take into account the previous word, when the probability for a new word is estimated. However, since a tag is associated with each new word, the last term of the bigram formula (8) for words will make sure that the new words match the context when they are suggested.

Another problem with learning unknown words is the estimation $f(w) = C(w)/C$, which is the only non-zero term of the word bigram formula when w is unknown. If the count C from the training text is used, $f(w)$ will be extremely small until a word has been typed many times. Another approach is to use another count C which counts the number of words typed during use, but that would mean that $f(w)$ will be very large in the beginning of a text and then diminish as more words are typed. Instead we decided to use a value for C such that

$$\max_{new\,word\,w} f(w) = F,$$

where F is a constant. This ensures that the most common word among the new words is equally probable at all times. By changing F, the influence of the new words can be monitored. The constant F is chosen to maximize keystroke savings.

9. Tagging unknown words

Our first approach to associate a tag with each unknown word as it occurred was to take the tag most probable according to equation (8). The selection was improved by only choosing among tags which are likely for unknown words. Basically, these tags are nouns, adjectives, verbs and adverbs. Not surprisingly, the tags selected for new words were not very accurate because only the left-hand side context of the word is considered.

Of course, the tagging can be improved by looking at the right-hand side context of the unknown word, as new words are typed. But that would require modifications of the program, since it is not designed to analyze the right-hand context of words. Instead, I investigated a method to deduce a likely tag by

statistically analyzing the morphology of words with respect to their tags. Intuitively, the last few letters of a word reveals much information about its part-of-speech classification. The problem is how to find a limited set of word-endings which gives as much information as possible about the probability distribution of tags associated with words with common endings. An elaborate algorithm to find a set of suffixes was devised, but unfortunately there was no available resources within the project to evaluate it. However, keystroke savings improved slightly using morphological tagging, which indicates that it may improve the language model.

10. **Alternatives to part-of-speech tagging**

    Part-of-speech tagging is one way of achieving a mapping from the histories of words to a more manageable number of states, as described in the previous chapter. However, there are no indications that a conventional set of part-of-speech tags is an optimal equivalence classification for word prediction purposes. Indeed, experiments conducted by Jelinek showed that a classification derived by statistical means outperformed a conventional classification [Jelinek, 1989]. Another interesting approach on how to infer a classification automatically is described in [Hutchens, 1995].

    By using part-of-speech tags for classifying words, some of the syntactic patterns of the training texts are captured, but none of the semantic patterns. This means that the predictor given the sentence beginning "*I met a talkative old*" will consider *man* equally probable as *stone* or *flu*. Obviously, a classification of words that capture both syntactic and semantic patterns would be preferable. An attempt to extend the syntactic tags with semantic categories was performed, but it did not result in an improvement of the language model.

11. **Selecting words from the training text**

The extracted data will serve as the base for the language model and thus the quality of the training text is crucial to the quality of the language model. In addition to the small tagged corpus we have access to an untagged corpus of 100 million words, mainly taken from newspapers and fiction. This corpus is large enough, but, as the tagged corpus, it may not contain texts of the kind a typical Prophet user would produce; short letters and notes, etc. A careful selection of texts would certainly be useful, but compiling it requires much work, and is beyond the scope of the project.

In the 100 million word corpus there are hundreds of thousands of unique words. It is unfeasible to store all of the unique words, so there is a problem in deciding which words should be selected. The straight-forward approach is simply to select the most frequent words. However, the most frequent words in the training corpus is not necessarily the words which will give the best coverage for other texts. Typically, certain names can occur very frequently in one text, but not at all in other texts. Therefore, a better approach would be to divide the corpus into a small number of groups, and to select words which are common in most groups using some criterion for ranking the words. Such an approach is yet to be implemented, and consequently the word unigrams and bigrams selected for the database are those with the highest frequency in the training texts.

3. Heuristic improvements of the predictor

Many easily observed features of natural language are not captured by the simple language model we have developed so far. Some of these features can be accounted for by simple heuristic methods described in this section.

1. **Strategy for incorporating modifications to the language model**

   The obvious way of modifying the language model to capture more of the language is to add weighted terms to the bigram formula (12). This ensures that the added terms will have optimal effect when the model is optimized. However, this requires that the additional terms are normalized, which is not always easy to accomplish. Moreover, when there are many new terms, which sometimes correspond to user optional features, it gets complicated to handle the weights vector correctly. For these reasons we decided to give up having a normalized probability distribution of the words. This means that the perplexity will no longer be possible to compute, and that we have to resolve other methods of evaluating the predictor as described in Chapter 5.

2. **Recency promotion**

   One feature of natural language is that an occurrence of a word increases the probability of that word occurring again, soon, in the same text. This is true especially for content words, as opposed to function words. To account for recency promotion equation 12 is given an additional term $r(w)$. This term cannot be determined without doing a thorough statistical investigation beyond the scope of the project, so an experimentally derived term was used:

   $$r(w_2) = F(w_2)R(w_2)\sum_t P(t_2|w_2)P(t_2|t_0,t_1) \quad (13)$$

   In order to implement recency promotion efficiently, each word in the database is associated with an 8-bit value R ranging from 0 to 255. Initially this value is 0 for all words, but as a word occurs, its R value is increased by some small value. As a sentence is completed, the R values for each word is decreased by another small value. By tuning the increasing and decreasing values, the R value can reflect the "recency" of each word.

   The function $F(w)$ has two different values depending on whether w is a function word or a content word. This reflects the observation that content words are more likely to reoccur than function words.

   The last factor was introduced to take into account the tag probability estimation given by the tag Markov model, thereby making the recency promotion sensitive to the previous words. This means that words with appropriate tags are promoted to a greater extent than other words.

   The implementation of this heuristic modification increased the keystroke savings by a few percent, which is a significant improvement. The scheme is very simple, and the introduced overhead is only one byte per word and a constant amount of

time per word to compute $r(w)$.

3. **Case sensitivity**

   Another way to improve the quality of predictions is to monitor the user s employment of capital letters. If the user capitalizes the initial letter of a word, the predictor can promote words that are usually spelled with an initial capital, typically proper nouns. Conversely, if the user does not capitalize the initial letter, the predictor can promote words whose initial letter is normally written in lower case. The promotion is achieved simply by scaling down the probabilities for the capitalized or non-capitalized words.

   Intuitively, altering the probability estimation to get case sensitivity should have a positive effect on the prediction quality. Indeed, with this enhancement added to Prophet, the predictions appear to be better, and a slight increase in keystroke savings was recorded. Thus case sensitivity was included in the new version of Prophet. However, since less advanced users may not use capitals correctly, they might be confused by the behavior of the predictor when the suggestions depend on the case of the initial letter. Therefore, we decided to make case sensitivity optional.

4. **Repetition of suggestions**

   With the prediction function developed to date, the user may get the same word suggested repeatedly, while typing the prefix of a word. This happens when the user types letters which makes the prefix match a word already suggested. For example, while typing the first letters of the word *jagar* (*hunts*) the predictor may well suggest *jag* (*I*) three times, thereby delaying the suggestion of *jagar*. This behavior is preferable in the case in which the user misses selecting a correct suggestion, because he will soon get another opportunity to select the desired word. On the other hand, multiple occurrences of the same word decreases the number of different words suggested, and hence, the possible keystroke savings are not optimized.

   Some users may benefit from having suggestions repeated early, whereas others may not. Therefore, we decided to implement a user option which delays the repetition of suggestions, until all other matching words have been suggested. This is accomplished by scaling down the probabilities of words which have already been suggested within the typing of a word.

5. **Minimal word length for suggestions**

   Just as in the old version of Prophet, the user can set the minimal length required for words to appear in the suggestion list. Typically, the user does not want to have very short words such as *i* (*in*) suggested.

   In the new version, the user can also set the minimum number of letters that would be saved when a suggestion is selected. For example, if this number is *two*, and the user has typed *three* letters of the current word, each suggested word would be at least *five* letters long. This option can be useful for a user without spelling difficulties, who does not want to have suggestions which save only one

letter, since it would require the same effort to type the last letter as to select the suggestion. On the other hand, users who *do* have spelling difficulties may well want to have suggestions which do not save any letters at all, because the suggestions can serve as confirmations of correct spelling.

6. **The order of words in the suggestion list**

   In the old version of Prophet, the suggestions were sorted by frequency. This way of presenting words works well for most users. However, studies show that sorting the suggestions by increasing length may help the user to find the correct suggestion faster. Moreover, users who prefer many suggestions may want to have them sorted alphabetically. We have implemented all three ways of sorting the suggestions, and the user can select the option he prefers.

7. **Deriving inflected forms of words**

   The main lexicon contains both base forms and inflected forms of words; for example the base form *göra* (*do*), as well as the inflected forms *gjorde* (*did*) and *gjort* (*done*) occur. Each word in the lexicon, which is a base form of an adjective, a verb or a noun, is marked with an inflection rule category number. Given the rule category number and the suffix of the word, the correct inflection rule for the word can be found. The rule is used to derive all inflected forms of the base form. The benefit of using this scheme is that many inflected words need not be stored explicitly, yet they can be derived when wanted, thus saving storage space. This scheme was first used in the old version of Prophet.

   The inflected forms of words are derived in two different situations. Just as in the old version, words in the suggestion list which can be inflected are marked, which enables the user to get another list with the inflected forms, when wanted. In the new version, inflected words can also be automatically derived when the predictor gets short of matching words. This feature was found to increase keystroke savings slightly in simulations. A problem is that many words with strong forms have a very small prefix in common with their base form, for example *göra* has only *g* in common with its inflections. This makes the procedure of finding all inflected words with common prefixes very time consuming, since basically all base forms must be investigated. Another disadvantage is that no statistical information of the derived forms is stored.

8. **Avoiding suggestions of misspelled words**

A word predictor which learns new words requires a smaller database and hopefully makes better predictions. A negative consequence of using an adaptive predictor, however, is that it will learn misspelled words, which later will be suggested to the user. Unfortunately, the predictor cannot discern correct new words from meaningless strings of letters accidentally created due to typing errors, and hence such non-words are also suggested to the user. When these non-words are suggested, the user is distracted and desired suggestions are delayed.

Having all new words blindly suggested to the user was a serious problem with the old version of Prophet, but is remedied in the new version in two different ways. The first way is by letting the predictor check all unknown words with a spell-checker, and allow

only accepted words to be suggested. The other way is by demanding that a new word is typed a certain number of times before it is allowed to appear in the suggestion list. This improvement will certainly be appreciated by the users, but both methods have the disadvantage that users are deprived of suggestions they might benefit from. This is the case when a word is correctly spelled, yet not accepted by the spell-checker, or when a word is, in fact, misspelled, but the user intends to correct all occurrences of the word later, using the spell-checker. We have implemented both these methods of avoiding misspelled words as user options.

4. Improvements of the user interface

In this section some of the issues concerning the design of the user interface of the new version of Prophet will be mentioned, although this was not my main task in the project.

1. **The context problem**

   To be able to make full use of its language model, Predict must continuously get the context to the left of the current cursor position. Since the user can change the cursor position arbitrarily, Prophet cannot assume that the text is composed sequentially, i.e. without cursor movements. The old version of Prophet made this assumption for practical reasons; it worked with any application and hence it could not get context information easily. The consequence was that when the user moved the cursor to another point in the text, the suggestions matched the prefix at the previous cursor position. Giving useless suggestions is obviously not a desirable behavior of the predictor.

   The new version of Prophet was required to solve the context problem. This is a difficult task, since Prophet should be able to work with an arbitrary application and there is no standardized way of getting access to its context. One way of obtaining information about the context is to let the user type in an intermediate buffer over which Prophet has complete control. This non-transparent solution is not always desirable, because of the extra load on the user who must shuffle text between the buffer and the target application. It may be helpful for dyslectic users, however, since it limits the working context and provides a simpler user interface.

   Another way of solving the context problem is to enable Prophet to work with a few particular applications, or perhaps to develop versions of Prophet, each capable of communicating with a different application. This solution would be transparent to the user, since no intermediate buffer is needed, but the disadvantage is that it requires an unreasonable amount of implementation work to cover all applications.

   It was decided to construct a Prophet program capable of working optimally with MS Word. It should also be able to work with other applications than MS Word, but then the context problem either demands use of an intermediate buffer or leads to reduced prediction quality. The intermediate buffer will have rudimentary word processing functionality, but as mentioned, it requires the user to shuffle the text to the target application.

2. **The user should always be able to get suggestions**

Perhaps the most significant improvement of the user interface concerns the user s possibility to find the intended word while typing. When the user starts to type a word, he may not be sure of the correct spelling, and thus he may not be able to type enough of the word to get the correct suggestion. Therefore, unlike the old version, the new version enables the user to make exhaustive searches for words without typing additional letters. When the user wants more suggestions, he clicks on a button under the prediction list or uses a short-cut key, and additional suggestions are displayed. This puts the user much more in control of his situation, and will always allow him to find the word he is looking for. Also, for many users, searching in a word list requires less effort than striking letter keys. This improvement will undoubtedly improve the user s appreciation of the program [Heckel, 1991].

Another improvement of the new version is that the list of suggestions, normally containing about five words, can be resized arbitrarily. The old version s suggestion list could contain up to nine words, but there is no reason to limit the number of suggestions. An extremely slow typist, such as physicist Stephen Hawking, is probably able to scan a large number of suggestions between each keystroke.

5. The prediction function

The language model and the heuristic modifications above give the pseudo-code of the prediction algorithm found on the next page. The calls to the procedures in rows 15, 16, 17, 20, and 21 perform the necessary changes to the probabilities of the words as described in this chapter.

1. PredictWords(*context*, *position*, *n*)
2. *prefix* = FindLastPrefix(*context*, *position*)
3. **if** IsCompleted(prefix)
4. *newWord* = LookUpWord(all lexicons, *prefix*)
5. **if** *newWord* = **Null**
6. *newWord*.tag = FindMostProbableTag( )
7. AddWord(appropriate topicLexicon, *newWord*)
8. **else** // *newWord* is known
9. **for** each tag *t* of previous position
10. recompute *t*.prob using (9)
11. **for** each tag *t* for next position
12. compute *t*.prob using (8)
13. **for** each word *w*
14. compute *w*.prob using (12)
15. DoRecencyPromotion( )
16. RequireLength( )
17. DeriveInflectedWords( )
18. **else** // prefix augmented
19. **if** Length(prefix) = 1
20. PromoteCorrectlyCapped( )
21. DelayRepetition( )
22. **return** the *n* most probable words

1. Lexicon management

The vocabulary of any natural language is unlimited, so it is obvious that the predictor only can know a fraction of all words of the language in question. The most common words from the training texts are put in a database called the *main lexicon*. This lexicon is supposed to cover at least 90 percent of the running words in any Swedish text, on average. Experiments have shown that this requirement can be met for Swedish with a lexicon containing about 10 000 words. Words which do not qualify for the main lexicon can be grouped together by topic and stored in *topic lexicons*, which are described later in this chapter.

1. **Main lexicon design**

We have decided to use a static, rather than a dynamic main lexicon which was used in the old version. Using a static lexicon has many advantages:

- No misspelled words can enter the main lexicon and its data are never corrupted.
- User-specific data is stored separately, which enables easy updates of Prophet without loosing information.
- More efficient data structures can be used, thereby increasing speed and saving memory.
- The predictor need not perform any troublesome updates of the statistical data.

The negative consequence of employing a static main lexicon is that adaptability must be realized by other means than by changing the main lexicon.

1. **Making the predictor adaptive**

   The predictor should learn new words it encounters during prediction. However, the new words cannot be placed in the static main lexicon. Instead, they are placed in topic lexicons from which the predictor also can select words for suggestion.

   The remaining problem in making the predictor fully adaptive is that words in the main lexicon will always have the same frequency, which implies that their individual ranking will always be the same. This result of static ranking is to some extent improved by the recency promotion described earlier in this chapter. Further improvements may be achieved by using some long term recency function capturing the user s choice of words, but this approach is yet to be tested.

   The need for adaptivity can be significantly reduced by using a more appropriate main lexicon in the first place, rather than radically changing an inappropriate one. A set of main lexicons representing different levels of writing skills should be developed, but this does not lie within the scope of this project.

2. **Topic lexicons**

   There can be many topic lexicons stored on file. When the user is writing about one or more topics, the corresponding topic lexicons should be activated. Prophet should then suggest words from these lexicons and enter new words into suitable lexicons. The ideal situation is that the predictor somehow triggers what topics the user is writing about and loads the corresponding lexicons, but this is generally

too difficult to achieve [Jelinek, 1989]. Consequently, the user himself must be responsible for activating the topic lexicons and for deciding which lexicon new words should be added to.

One problem with using topic lexicons is that it will take an unacceptably long time for the user to produce sufficient amounts of text for the topic lexicons to reach adequate sizes [Jelinek, 1989]. Therefore, Prophet was given the ability to scan the user s previously composed texts and produce topic lexicons containing the words unknown to the main lexicon and other topic lexicons.

The old version of Prophet employed only one topic lexicon. The new version is capable of managing an arbitrary number of topic lexicons. An advantage of using many topic lexicons simultaneously is that the user can have a "private" topic lexicon containing non-topic-related words he uses very often: his name and address, names of friends, etc. This lexicon can be used simultaneously with other topic lexicons. Moreover, many users can share a common copy of the predictor without having to duplicate the main lexicon. The topic lexicons are expected to contain on the order of one thousand words each, but there is no limit on how many words they can contain.

3. **Statistical information about topic words**

Ideally, the information about main lexicon words and topic words should be the same. However, there are some practical problems with this approach. Firstly, if topic words can be parts of bigrams, then there will be bigrams with one word in the main lexicon and the other word in a topic lexicon. Secondly, bigrams must be added during prediction, since topic words are not known in advance. There are many ways to store and manage this type of bigram, but in all cases there is too much administration needed for it to be worthwhile implementing within this project. Hence no such bigrams are stored, and this means that the first term of equation (12) is zero, when the previous word is a topic lexicon word. This is already the case for some main lexicon words, since far from all bigrams of the main lexicon words are stored.

Limiting the amount of information associated with new words is of course limiting the predictor s learning ability, but since the predictor tags unknown words, it still "learns" something about new words.

1.

Implementation of the Language Model

This chapter describes the implementation of the language model designed in the previous chapter. It also describes programs developed in order to extract data from the training texts and produce a database, as well as a program for optimizing and evaluating the predictor.

1. A system for word predictor construction

The creation of a word predictor requires processing of information from different sources. The word predictor needs a database containing statistical information about *n*-grams, inflection rules, etc. It also needs efficient means of performance evaluation. For these purposes I have developed an *n*-gram extractor called *Extract*, a database generator called *Generate,* and a simulator called *Simulate*. I have also developed a rudimentary UNIX user interface for Predict, called *UNIX-Prophet*. The relationships between these programs and the sources of information are shown schematically in Figure 3.

1. **The extractor**

   To extract word *n*-grams, as well as tag *n*-grams, from large training texts Extract scans a set of text files and counts occurrences of the *n*-grams of interest. The training texts can be either tagged or untagged. The resulting data are stored in files.

2. **The generator**

   The *n*-gram files extracted by Extract serve as input to Generate. The Generate program prunes the sets of *n*-grams to desired sizes and sorts them. Additional information, such as inflection rules, is added, and the information is stored on file as a main lexicon.

3. **The predictor**

   The main lexicon created by Generate can then, without any further processing, be used by Predict. The predictor can also use an arbitrary number of *topic lexicons* previously created by Predict or generated through the scanning of texts.

4. **The simulator**

The Simulate program is given a text and uses Predict to reproduce the text with as few keystrokes as possible. The simulator represents a perfect user who does not make any typing errors, and who does not miss any correct suggestions. When the text is completed, the simulator responds with the following data:

- Text statistics, such as number of letters and words in the text.

- Text coverage, i.e. the fraction of the running words that was known to the predictor before the simulation started.

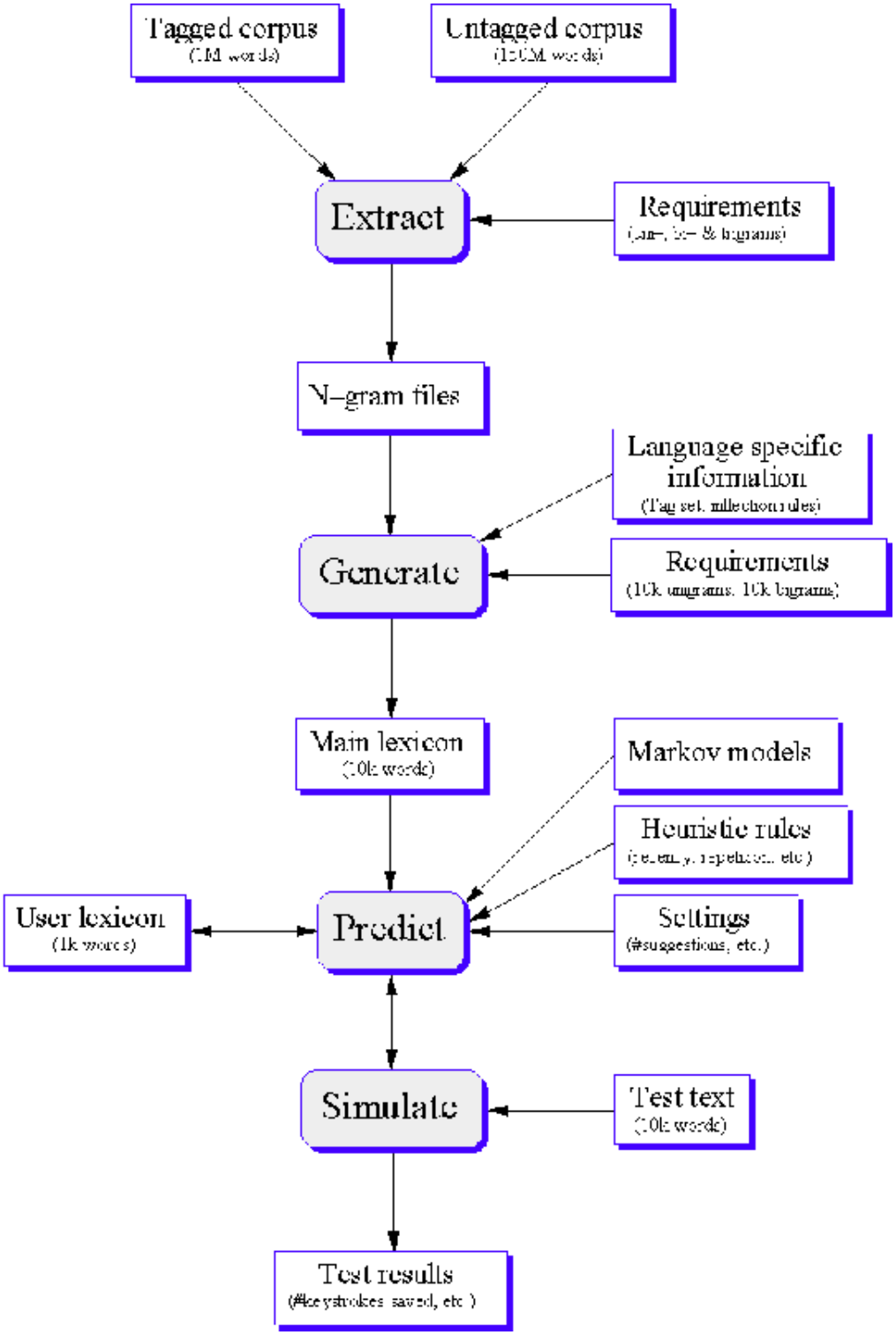

Figure 3. A system for word predictor construction.

-
- Letter and overall keystroke savings.
- The distribution of the positions of the correct suggestions in the suggestion list.

This information is then used to evaluate the performance of different lexicons and different configurations of Predict on selected test texts. Simulate also contains functionality for finding optimum weights of the Markov model formulas and the parameters of the heuristic modifications.

1. **The programs are easy to use**

If too much manual labor is required to generate lexicons and to evaluate the predictor, the chance that these programs will be used again in the future is small. Therefore, I have put substantial effort in making these programs do as much work as possible automatically. For example, lexicons from different training texts and of different sizes can easily be generated with Generate using command line options. The lexicons can be used by Predict directly. Then, the difference in performance between the lexicons can be measured by Simulate, without any additional processing.

1. Modeling of words and lexicons

1. The word class and lexicon class hierarchies

Predict uses words with different sets of information. For example, main lexicon words have bigrams and tags associated with them, as opposed to topic lexicon words which only have frequency and one tag. For this reason it makes sense to use a class hierarchy for words and lexicons.

An abstract class *Entry* represents objects which can be put in lexicons, represented by an abstract class *Lexicon.* The Entry class has a subclass, *Word*, which is in turn a superclass to the *MainWord*, *TopicWord*, and *DerivedWord* classes. These three classes represent words from the main lexicon, from topic lexicons, and words derived by inflection rules, respectively. There is also a hierarchy for lexicons, which mirrors the word class hierarchy.

There are many benefits of using a class hierarchy for words. Most parts of the program need not know if the words originate from the main lexicon or from a topic lexicon. This enables more compact and readable code. Furthermore, changes in the representation of words will not propagate throughout the whole program.

2. Data representation

As we have seen in the previous chapters, the predictor uses a huge amount of statistical information and is very computation-intensive. Therefore, which data structures and algorithms are used is of crucial importance for the performance of the Predict program. Since the program is likely to run simultaneously with word-processors, such as MS Word, which tend to use an extensive amount of the computer s available resources, the importance of the effectiveness of the program must not be underestimated.

The predictor s vocabulary must be quite large, in the order of 10 000 words, perhaps even up to 100 000 words. The predictor must also be able to learn thousands of new words typed by the user. Furthermore, inflected words of base forms can be derived, when needed. All these words, at least in principle, must be examined by Predict, at each transition of the Markov models, in order to calculate probabilities for the next word. This fact requires a set of different data structures for storing the words efficiently.

1. **Representation of the main lexicon**

As mentioned in the previous chapter, the main lexicon is static. Thus, information about words in the main lexicon can be stored efficiently in a fixed size array, with fixed size elements. However, since the actual strings of the words have different sizes, all strings are stored in a common string pool to save space. Further improvement is possible by letting strings with common suffixes share their storage space.

The number of bigrams of a word is varying, so all word bigrams are also stored in one common array in order to save space. Each word has an index into the array where the indices of the other word (and the frequency) of the bigrams are located.

The words in topic lexicons, as well as the main lexicon words, are stored in sorted arrays. The reasons for choosing this structure are the following:

- A word in a lexicon of size *n* can be accessed with binary search in $O(\log_2 n)$ time.
- An interval of words sharing the same prefix can be found in $O(\log_2 n)$ time.
- Iteration over an interval of sorted words is straightforward and fast.
- A minimal overhead for maintaining the structure is necessary.
- The structure and its operators are easy to implement.

The major drawbacks of using arrays are that insertion is performed in $O(n)$ time, and that re-sizing of the arrays may be necessary. However, since the main lexicon is static, insertion is only performed on topic lexicons, which are considerably smaller than the main lexicon.

1. **Representation of the tag Lexicon**

   The number of tags *m* is much smaller than the number of words *n*, typically $m \approx \sqrt{n}$. Since the tag bigram matrix is relatively dense, it is stored in a two-dimensional array enabling fast access. The tag trigram matrix, on the other hand, is sparse, so trigrams are stored in a more compact form. For each tag there is an array of elements containing the second and third tag and the frequency of the trigram. This saves space at the cost of slower access. Access will be especially slow when iterating over all trigrams with the first and third tags fixed, or with the second and third tags fixed. The present version of the program, however, does not access trigrams in this way, so this is not a problem.

2. **Efficient representation of data**

   I have chosen to use data structures which do not require an abundance of pointers and auxiliary structures. As described above, simple arrays are used for storing words. In addition to the benefits of using arrays already mentioned, an important consequence of this choice of representation is that large chunks of memory can be allocated when the lexicons are loaded into the program. This cannot be done

as easily with structures such as lists or trees, which use pointers. The benefit of the chunk-wise memory allocation is that the program will load faster, which is a requirement for the Macintosh version of Prophet. Another benefit is that it saves space, since when many small segments of memory are allocated, the built-in memory manager will always use at least a few bits of extra memory for each allocated piece of memory.

When Prophet is once compiled for one type of machine, the main lexicon, which is common for all types of machines, is loaded into memory from a file. Then the lexicon is stored again just by writing the byte sequences of the big chunks of allocated memory to another file. Now the main lexicon can be loaded much faster in the future by using this file instead. UNIX-Prophet loads ten times faster this way. Predict with a 10 000 word lexicon is loaded in less than a second on a PC.

3. **Implementation of Extract**

The extractor uses self-resizing hash tables for the storing of *n*-grams. As elements are inserted, the hash tables grow if the ratio between the number of elements and the size of the storage array exceeds a certain limit. This assures that lookups and insertions are performed in constant time, whatever the size of the training text.

Since the number of word bigrams can be very large, the texts are scanned twice. During the first scan all information except word bigrams is extracted. Then the set of word unigrams is pruned of low-frequency words. During the second scan only bigrams with both words occurring in the reduced set of unigrams are considered, thus avoiding unnecessary storing of bigrams.

4. **Time complexity analysis of the prediction function**

To look up a word in a lexicon of size n and to find intervals of words matching a prefix both take $O(\log_2 n)$, since binary search is used. Each time a word is completed by the user, the probabilities for all words are re-computed, which takes $O(n)$ for each lexicon. The Markov model for tags makes a transition which takes $O(t^2)$, where t is the number of tags. If the last word was unknown to the predictor, the word has to be inserted into a topic lexicon of size u, which takes $O(u)$. Hence the total time is

$$O\left(\log n + n + \sum_i u_i + t^2\right),$$

which is $O(n)$, with the assumptions made in this and previous chapters.

It is possible to reduce the number of words for which probabilities are computed at each transition by only considering the most common ones from the most likely classes. However, since experiments show that the time spent at each transition is short enough not to cause any irritating delays for the user, we decided not to implement such an improvement. Also, it is possible that such an improvement

would debase the prediction quality if neglected words would have been selected if all words were considered. Moreover, the implementation cost is not negligible, since the words not considered at the transition will have to be investigated later on, and this would inevitably require some administration.

5. **Speed optimization**

   It is possible to reduce the time required for prediction by pre-computing parts of the equations used, or by transforming the equations in order to avoid time-consuming arithmetic computations, such as divisions. At the cost of using more memory, the terms of the Markov model equations were pre-computed, resulting in a 30 percent speed-up of the prediction function.

6. **The source code**

All programs are written in ANSI C++ to ensure portability. A number of generic classes, such as hash tables, lists, arrays, and object pools have been developed to handle sets of objects with varying demands on access to their elements. These classes have been reused extensively in all parts of the programs.

The programs are modular to enable different types of functionality to be well isolated from each other. This facilitates easy implementation of future improvements. For example, by changing a few well isolated parts of the program we can extend the scope of the Markov models.

All programs together comprise 5 600 lines of code. Predict alone comprises 3 800 lines of code.

1.

Evaluation of the new Prophet

A word predictor should be evaluated in its full context. The benefits of saving keystrokes and getting correct spelling must be weighed against the drawbacks of searching the suggestion list for the intended word. The usefulness of the program is also specific to each user and depends on many different aspects of the program. Such an evaluation of Prophet is beyond the scope of this project for my part. In this chapter I will only consider keystroke savings and the quality of the suggestions.

1. **Measuring the performance of the predictor in practice**

   As described in Chapter 2, there are ways of measuring the performance of a word predictor given by information theory. The best way seems to be to calculate the perplexity of the predictor with respect to a test text. However, there are two practical disadvantages using the perplexity as a performance measure.

   The first problem concerns unknown words. When an unknown word is

encountered by the predictor, it is infinitely surprised, resulting in an infinite perplexity which makes comparisons impossible. A solution could be to use a pre-defined number for the surprise instead of infinity in this case. But what number should that be? If a small number, such as zero, is used, predictors with a small vocabulary will score the lowest perplexity, since they will rarely be very surprised. Conversely, if a very large number is used, predictors with large vocabularies will score best, since they will rarely be infinitely surprised, thus collecting the least number of large surprises. Consequently, fair comparisons between predictors with different lexicons are hard to make using perplexity as a performance measure.

The other practical problem is that computing perplexity requires the probability distribution over the words to be normalized. Normalization is a time consuming operation not affecting the rank among the suggestions, thus not improving the prediction quality. Moreover, using perplexity requires that the heuristic complements to the prediction function, such as recency promotion described earlier, must be used more carefully, since they alter the probability distribution in a rather unsophisticated way.

Because of these problems with computing the perplexity, we decided to use the keystrokes savings defined by (5) as a performance measure instead. This measure, in turn, has advantages and disadvantages as well. The advantages are that keystroke savings are easy to compute and that commercial word predictors are evaluated by comparing keystroke savings. Thus, keystroke savings for comparison purposes must be computed anyway.

Apparently, keystrokes savings are a cruder measurement than perplexity, since a change in the probability distribution of the words may not change the rank among the words, and hence minor improvements of the prediction function may be hard to detect. Another disadvantage is that the number of saved keystrokes does not say anything about the position of the correct word in the suggestion list. If the suggestions are sorted by probability, one predictor might score better than another one when five suggestions are used, but it could be the other way around when four predictions are used. This is true when the first predictor happens to have more correct fifth position suggestions, but fewer correct suggestions in the first four positions than the other predictor.

2. **The perceived quality of the predictions**

As mentioned, the performance of word prediction programs cannot be measured by perplexity or keystroke savings alone. The perceived quality of the predictions as experienced by the user is even more important than keystroke savings. Ill-fitting suggestions will distract and slow down the user, and he might even stop using the program, even if it saves many keystrokes. As mentioned in the introduction, it is not possible to avoid ill-fitting suggestions, but it is important to delay them as much as possible.

Also, a slight change of the prediction function might not give a significant improvement of the keystroke savings, yet the change could be valuable if the suggestions given by the predictor appear to be better in some respect.

1. Results of simulations

This section covers some of the tests we have performed to investigate the impact of different features and parameters of Prophet. Four independent 10 000 word texts were used for the experiments one text for optimizing the parameters and the other three for simulation and evaluation.

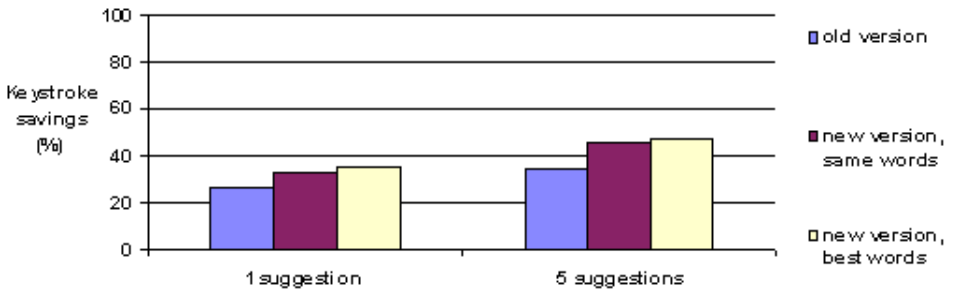

Figure 4. Keystroke savings of old version versus new version of Prophet.

1. **Keystroke savings of the old versus the new version**

   The old version of Prophet used a main lexicon containing 7 014 words and 7 278 word pairs. For the purpose of comparing the two versions, two lexicons of the same size were generated for the new version. The first of these lexicons contained exactly the same words as the lexicon of the old version. The second lexicon contained the 7 014 most common words from the tagged corpus. Because the keystroke savings evaluation program for the old version was error-prone, only one text of 200 words could be used successfully for the simulations. The results are shown in Figure 4.

   The new version saved 33.0 percent of the keystrokes compared to 26.1 percent saved by the old version, when the same words and one suggestion was used. This represents a 26.4 percent improvement. When the second lexicon was used, 35.4 percent of the keystrokes were saved compared to 26.1 percent which is a 35.6 percent improvement. When five suggestions were used the improvements were 33.2 and 37.6 percent, respectively.

2. **Impact of different features**

   The tag Markov model, adaptivity by using topic lexicons, and the heuristic modifications of the language model all contribute to improve the predictions. By comparing the keystroke savings of an optimized Prophet configuration with the keystroke savings of the same configuration, but with one feature removed, the impact of that feature is revealed. Table 1 lists the features and the increase in keystroke savings obtained when each feature is added to an otherwise optimal configuration.

   Table 1. Impact on keystroke savings by different features.

	1 suggestion		5 suggestions	
Feature	**% saved**	**% increase**	**% saved**	**% increase**
-	**32.9**		**46.0**	
learning new words	**29.7**	**10.7**	**40.9**	**12.6**
tagging	**28.7**	**14.9**	**43.2**	**6.4**
recency promotion	**31.9**	**3.1**	**43.9**	**4.8**
repetition delay	**29.2**	**12.9**	**45.4**	**1.3**
case sensitivity	**32.4**	**1.5**	**45.8**	**0.5**
auto-inflection	**32.8**	**0.3**	**45.7**	**0.7**

3. Number of suggestions

The number of words in the suggestion list has of course a big impact on keystroke savings. Naturally, the keystroke savings increases monotonically with the number of alternatives. But the more suggestions, the longer time it takes for the user to inspect them and the greater the chance of missing correct suggestions. Also, as the number of suggestions grows, the increase in keystroke savings diminishes. Most users are able to quickly scan five alternatives, which is also the default setting in Prophet.

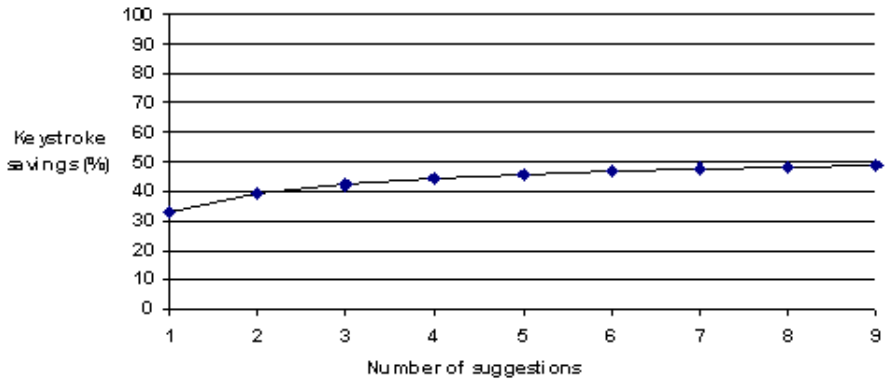

Figure 5. Keystroke savings as a function of the number of suggestions.

4. Comparisons with word predictors for other languages

Prophet is the only commercially available word predictor for Swedish, but there are a number of word predictors for English. Naturally, the keystroke savings measured for English word predictors cannot be directly compared with the keystroke savings for Prophet. However, experiments with the old version of Prophet indicate that higher keystroke savings can be obtained with word predictors for English than for Swedish when lexicons with equal sizes are used on comparable test texts. One plausible reason for this is that it is easier to get a high text coverage for English. Many new Swedish words are composed by concatenating two or more words, which is not the case for

English.

2. Prediction examples

The difference in the quality of the suggestions of the old and new versions is seen in the following example: When given the sequence *Det är en lätt up...* (*It is an easy...*), the old version suggests *uppgifter* (*tasks*) and *Uppsala* (the name of a town) which does not match the two last words very well at all. Only *uppgift* (*task*) and *uppfattning* (*apprehension*) can be considered acceptable suggestions. The new version on the other hand gives only one less acceptable suggestion, *upp* (*up*).

Det är en lätt up...

upp

uppgift

uppgifter

uppfattning

Uppsala

old version

uppgift

uppfattning

uppmärksamhet

upp

upplevelse

new version

The difference in performance between the versions is even more striking when we change the example slightly. When given the sequence *Det är inte **ett** lätt up...* (*It is not an easy...*), the old version suggests exactly the same five words as before, even though there was another determiner *ett* instead of *en*. In this case none of the suggestions can be considered acceptable. But when the same sequence is given to the new version, the suggestions are all acceptable except the last suggestion, *upp* (*up*). This proves that the new version successfully takes into account the two previous words when predicting the next word.

*Det är inte **ett** lätt up...*

upp

uppgift

uppgifter

uppfattning

Uppsala

old version

uppdrag

uppträdande

uppenbart

uppror

upp

new version

It can also be noted that the old version suggested the word *Uppsala*, despite the fact that it is spelled with an initial capital. This is solved by the new version which will (optionally) delay the word *Uppsala* until all other words are suggested.

2. Future Improvements

In the previous chapters the design and implementation of the Prophet word predictor were discussed and evaluated. Improvements such as extending the scope of the Markov models and automatic tag classification and other straightforward enhancements have already been mentioned. In this chapter I will discuss some more elaborate improvements of word prediction tools in general.

1. **Abbreviation expansion**

   A common feature of word processors and typing aids is "abbreviation expansion" or "auto correct". For this purpose, the word processor manages a list of string pairs. When the user types the first string of such a pair, this string is automatically transformed into the other string of the pair. For example, *sth* can be expanded to *Stockholm*, and *ohc* auto-corrected to *och*. This simple feature can save many keystrokes for a user capable of using it correctly, but we have decided not to implement it in Prophet, since such a feature does not lie within the scope of the project. However, it will surely be incorporated in future versions of the program.

2. **Prophet and dyslexia**

   As stated previously, the two main target groups for Prophet are physically handicapped persons and people with spelling disabilities such as dyslexia. Since there are no guarantees that the predictions are semantically or even syntactically correct, the latter group will have two major problems using word predictors.

   The first problem is that the user must be able to decide if a suggested word is the word he intends to type. In fact, this is not always the case. For example, a dyslectic user intending to type the word *kall* (*cold*) may not be able to select the right word when given the two similar suggestions *kal* (*bare*) and *kall*.

   In Prophet, this problem is solved to some extent by incorporating speech synthesis in the program. Speech synthesis will enable the user to listen to a selected word, thereby making it easier for him to decide whether that word is the intended word or not. Still, mistakes are inevitable due to the many homophones in natural language. For example *kål* (*cabbage*) and *kol* (*coal*), are both nouns with the same pronunciation, but with different meaning. The next step would be to include a thesaurus to enable explanations of words.

   The second problem occurs if the user makes a spelling error at the beginning of a word. The spelling error causes the intended word never to show up in the suggestion list. For example, the user might start to type *syk* when he wants to type *cykel* ((*bi-)cycle*), but this word will never be suggested, because it does not match the prefix *syk*.

   It is possible to solve this problem by having a spell-checker generate alternate prefixes to a suspicious prefix (such as *syk*). These alternate prefixes (maybe *suc*, *cyk*, and *psyk*) can then be used as input to the predictor, instead of the original prefix, thereby enabling the generation of suggestions possibly including the intended word.

3. **Cooperation with spell-checker**

The spell-checker can help the predictor by generating alternate prefixes, but the predictor can also assist the spell-checker in finding better alternatives to misspelled words. The predictor makes probability estimations of tags and words in a text. This information could be used by the spell-checker to find suggestions which would fit the context better. For example, if I type "*I have a brown shue*" my spell-checker in MS Word 7.0 suggests *she* as the first alternative to *shue*. If the spell-checker would have consulted Prophet before it made the suggestions, maybe a noun such as *shoe* would have been the first alternative.

Yet a better way to assist spell-checkers could be to let Prophet identify correctly spelled words which do not seem to fit into the context, and warn the spell-checker about those words. In this way the spell-checker could detect words which are misspelled, but happen to form other words that the spell-checker accepts. This is a typical problem with most or all spell-checkers. The same spell-checker as above gladly accepts the sentence "*I have a she.*" Prophet would hopefully find the word *she* improbable given the context. This improvement, however, is difficult to implement successfully, because Prophet might give too many false alarms. Also, this type of proof-reading should rather be done using a more powerful language model which takes into consideration both left and right context of words while

inspecting the text. But if no such program is available, Prophet could easily be modified to do this kind of proof-reading. If this cooperation is made to work successfully, word prediction will not be limited to the current target groups. All computer users would benefit from an improved spell-checker.

3.

4. Conclusions

1. Performance

It is clear that Prophet, in its current configuration is best suited to help physically disabled persons without any linguistic problems. Such users can speed up their typing radically when the program saves almost half of the keystrokes. However, to make full use of the program, the user must master the language he is using. The predictor can only partly assist a linguistically handicapped person, since it cannot give only "correct" suggestions. Furthermore, the program does not aid the user when he misspells the prefix of a word, since no automatic "prefix-correction" is implemented yet.

The tests in the previous chapter revealed that the new version of Prophet saves 47 percent compared to 35 percent of the old version, on an average text, which is a large improvement. When using a larger lexicon the new version achieves keystroke savings of almost 50 percent.

The new version of Prophet is a sound implementation of well-established probabilistic methods and a number of heuristic complements. This work has not come up with any theoretical break-throughs in the field of natural language processing and the program does not comprise any state-of-the-art language modeling. Nonetheless, the resulting application will likely outperform all word predictors on the market, given the features and keystroke savings reported for these programs.

Reports from test users at the department of Speech, Music, and Hearing indicate that the new version gives the impression of being more intelligent, and that it suggests words which fit much better into the context than the old version did.

2. **Limitations**

The major limitation of the language model is the small scope. Presently, the predictor considers the previous word and the two previous tags in order to make predictions. Hence, words outside the scope, which often have an impact on the next word are not considered at all, and consequently the prediction quality suffers.

The scopes can be extended by one word and one tag, respectively, but it seems

impractical to extend them further. Thus, to make improvements beyond the capability of Markov models as used in this project, other methods must be investigated. Interesting approaches are clustering, proposed by Hutchens [Hutchens, 1995] and methods described in [Jelinek, 1989].

3. **Word prediction for everyone**

It is easy to develop versions of Prophet for other languages since the predictor uses mainly statistical information. The language specific rules are kept to a minimum and are well-isolated from the language-independent parts. A rudimentary version can be developed for any language instantly just by extracting a database from a reasonably large corpus.

So it is easy to develop Prophet versions for all languages, but the users are still limited to physically and linguistically handicapped persons. But there is another potential user group that would benefit from using Prophet: persons who are learning a new language. The same problems as with linguistically handicapped persons are likely to concern this group too, but by incorporating a dictionary for the native language of the user, Prophet might be a very useful tool for this group of people. A successful implementation could mean a word predictor tool valuable for virtually everyone, since most of us are learning another language!

Bibliography

Cormen, Leiserson, Rivest (1990) *Introduction to Algorithms*, MIT Press, McGraw Hill.

Ejerhed E., Källgren G., Wennstedt O., Åström, M. (1992) "The Linguistic Annotation System of the Stockholm-Umeå Corpus Project", DGL-UUM-R-33, Department of General Linguistics, University of Umeå, REPORT NO. 33.

Fischer, C. N., LeBlanc, R.J. (1991) *Crafting a Compiler with C*, Benjamin/Cummings Pub. Co.

Heckel (1991) *The Elements of Friendly Software Design*, Sybex.

Hunnicutt, S. (1986) *Lexical Prediction for a Text-to-Speech System in Communication and Handicap: Aspects of Psychological Compensation and Technical Aids*, E. Hjelmquist & L.-G. Nilsson, eds., Elsevier Science Publishers.

Hutchens, J (1995) *Natural Language Grammatical Inference*, University of Western Australia.

Jelinek, F (1989) *Self-Organized Language Modeling for Speech Recognition*, Readings in Speech Recognition, Waibel and Lee (Editors). Morgan Kaufmann.

Lippman, S (1991) *C++ Primer*, Addison-Wesley.

Magnuson, T. (1994). *Evaluation of "Predict": An investigation and a follow-up study of a Swedish word prediction program*. STL-QPSR (Speech Transmission Laboratory Quartely Progress and Status Report) 4/1994, pp. 1-20.

Parsons, T. W. (1992) *Introduction to Compiler Construction*, Computer Science Press.

Stroustrup, B (1991) *The C++ Programming Language*, 2nd Edition, Addison-Wesley.

von Euler, C. (1997) *Dyslexi - ett allvarligt handikapp på biologisk grund, (A serious handicap with a biological basis)*, Socialmedicinsk Tidskrift, vol. 74, nr. 1.