Course: DD2427 - Exercise Set 10

Exercise 1: *Non-linear SVM*

Consider training data of 1-dimensional points from two categories:

$$\omega_1 : -5, 5$$
$$\omega_2 : -2, 1$$

a) Plot these points. Are they linearly separable ?

b) Consider the transformation $\psi : \mathcal{R}^1 \rightarrow \mathcal{R}^2$ defined by $\psi(x) = (x, x^2)$. Transform the data using $\psi()$ and plot these transformed points. Are these transformed points linearly separable ?

c) What is the optimal separating hyper-plane in the transformed space ? This separating hyper-plane results in a non-linear discriminant function in the original space. Plot this non-linear discriminant function.

For the next lecture: 3rd May

Bring your hand written solution to this exercise.

Exercise 2: *SVM and Digit Recognition (optional)*

In this exercise you will write code to use the package `libsvm` to implement a SVM classifier on the digit data. Your first task is go to the webpage
`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
and download the *Matlab* package in the list `Interfaces to LIBSVM`. Unzip the file you download and then move to the directory `libsvm-mat-2.91-1`. Here you may have to edit the file `Makefile`. This depends on which operating system etc you are using. Basically, you have to ensure that the code knows where the *Matlab* libaries are on your machine. Thus for my `csc` machine I changed the line

```
MATLABDIR ?= /usr/local/matlab
```

in `Makefile` to

```
MATLABDIR ?= /pkg/matlab/r2008b
```

Afterwards, while you are sitting in the directory `libsvm-mat-2.91-1` run the command.

```
$ make
```

If this goes without a hitch then you should have generated the commands for SVM training and evaluation that can be used by *Matlab*.

Now you are ready to get going. Upon starting `Matlab` add the directory containing the digit figures, you used in the previous exercise set, and the directory containing the SVM code to your path.

To begin with load all the training images and store them as columns in a data matrix `X` of size `784` × `1000`. Make sure you **normalise each image** to have a minimum pixel intensity value of 0 and a maximum intesnity value of 1. This is required for numerical reasons and allows the SVM code to run more efficiently. Also keep a record of the labels of the images. Note that the feature vector you are using is just the raw pixel data. Once you have loaded the data you will try and find a separating hyper-plane between one digit and the others. You will use an SVM to do this.

To summarize you will need to write functions to perform the following:

- Load the training data. Normalize each image and construct the data row `X` of size `784` × `1000`. Each column of `X` is a training image. Also keep a record of the image labels which correspond to the digit in the image. This can be obtained from the name of the image.

- Write a function that calls the package `libsvm` to train and return a SVM structure to find a seperating hyperplane between one digit and the rest. This function will have the form

  ```
  function svm = SVMLearning(X, labs, d1)
  ```

  The inputs to the function are the training data `X`, the labels of the images and an integer `d1` representing the digit you want to recognise. You will find a separating hyperplane between digits of type `d1` and the other digits. (These will be the two classes and will have labels `t= -1` and `1`.) Then you can train your svm with the command:

  ```
  svm = svmtrain(ts, X', '-t 0 -w1 1 -w-1 9 -q -c 1');
  ```

  Most importantly the flag `'-t 0'` indicates you are fitting a linear SVM, the flag `'-w1 1 -w-1 9'` indicates that you have unbalanced training data and therefore you should penalize a misclassification of a training example from class `-1` with a weight of `9` and from class `1` with a weight of `1` (note this is the ratio of the number of class 1 examples to class -1 examples). Finally the flag `'-c 1'` controls the penalty term added to the cost term when a hyperplane misclassifies an example. This last parameter we will return to later.

  Type the command

```
>> svmtrain
```

within *Matlab* to get a summary of what these parameters represent and other available parameter settings

- Once you have learnt `svm`, load the test images into another data matrix `X1` and the labels `labs1`, of course, remember to normalize. Then see how many of the digits of type `d1` you can classify correctly, the number of true positives `tp`, in conjunction with the number of true negatives `tn` as well as the accuracy of the classifier. Do all this in a function called

  ```
  function [tp, tn, acc] = TestHyperPlane(X1, labs1, d1, svm)
  ```

  In `TestHyperPlane` you will call the function

  ```
  [pred_labels, acc, d_values] = svmpredict(ts, X1', svm);
  ```

  where `ts` are the labels of the test examples which have value `-1` or `1`. The useful outputs of this function are the predicted labels of the test examples and the accuracy of the classifier, which is if you remember `(tp+tn)/(tp+fp+tn+fn)`.

Run the code you write and print out `tp`, `tn` and `acc` for the digits 0, 4 and 9. The numbers I got when I learnt a separating hyperplane for each of the digits against all the others are shown in the table.

Classified Digit	`tp`	`tn`	`acc`
0	89	882	.9710
1	97	896	.9930
2	69	841	.9100
3	66	876	.9420
4	40	867	.9070
5	75	839	.9140
6	67	858	.9250
7	71	876	.9470
8	51	851	.9020
9	46	856	.9020

Exercise 3: *SVM and Digit Recognition II (optional)*

There was one parameter `c` which we set in training the SVM. We gave it value `1`. However, it is not clear what the *best* setting of `c` is. A process called **k-fold cross validation** is a way to explore how to set this parameter. Here is a brief description of **k-fold cross validation**

> First split the set of positive training examples into k subsets of equal size such that each example appears in only one subset. Similarly split the set of negative training examples into k subsets of equal size. Fix the value of `c`.
>
> Use the first $k-1$ subsets of the positive and negative examples to train an SVM. Apply this learned SVM to the kth subset of positive and negative examples that you omitted from the training process and record the accuracy of the classifier. Then retrain the SVM using all the training data minus the $(k-1)$th subset. Use the retrained SVM on the $k-1$ subset and record the new classifier's accuracy. Repeat the process with the $(k-2)$th subset omitted, then the $(k-3)$rd and so on until you've omitted the 1st subset. Record the average accuracy of the classifiers you have constructed. This average accuracy is an estimation of how good the parameter settings of `c` was.
>
> Repeat the process for different settings of `c`. Then choose the setting which produces the best average accuracy.

Luckily, for you `svmtrain` has a flag which performs cross validation. So if you call the command as follows

```
acc = svmtrain(ts, X', '-t 0 -w1 1 -w-1 9 -q -c 1 -v 5');
```

Then it will perform `5`-fold cross validation and return the average accuracy of the classifiers learnt given a `c` setting of `1`. Therefore, your task is to run the command `svmtrain` with the `-v` flag and different values of `c`. Record the value of `c` that produces the best accuracy. Given, `c_best`, the best value of `c`, train your SVM using all the training data with the command

```
mstr = ['-t 0 -w1 1 -w-1 9 -q -c ', num2str(c_best)];
svm = svmtrain(ts, X', mstr);
```

Typically a loose search is performed with `c=`$2^{-5}, 2^{-3}, \ldots, 2^{15}$ and then afterwords you can try and improve upon these values by performing a finer search around the local maximum you found.

Once you have re-learned your SVM, then you can test how well it performs on the test data. So does for the digits `0, 4` and `9` and record the results.

Once you have re-learned your SVM, then you can test how well it performs on the test data. So does for the digits `0, 4` and `9` and record the results.

The package `libsvm` also allows you to map your data to higher dimensional spaces via various kernel functions. If you are interested you can play around with these and see if you can improve your recognition rates. However, with different kernels you'll have to redo your search for a good value of `c`.