ISBE
Imaging Science and Biomedical Engineering

Machine Learning
Post-course Work

R. S. Schestowitz*
Research Student
Imaging Science and Biomedical Engineering
Stopford Building
University of Manchester
United Kingdom

23rd October 2003

Part I: Support Vector Machines

Polynomial SVM

There are two datasets to be handled with: (1) digits and (2) oil pipeline network. We will deal with each one in turn.

Digits

In this report, details of the algorithms used will be specified with as much details as necessary to reflect on the functionality of the m-files.

We begin by reading in 60 examples of each of the 10 digits – that is 600 in total. We divide them up into two equal-sized sets for training

*Contact: *sch@danielsorogon.com*
Electronic version: *http://www.danielsorogon.com/Webmaster/Research/ML*

and testing purposes, while bearing in mind the later sub-division of the training set for the sake of cross-validation.

We pre-process all of these examples so that each is represented by only 64 dimensions (with up to 4 bits necessary to encapsulate each dimension) and then unite them to obtain two large sets of digit examples[1]. The output vectors are then initialised to correspond to the inputs (a simple correspondence which is driven by the index). These will be two 1x300 matrices, one for training and one for testing (the other two input matrices are 64x300 in size). We then need to divide the training inputs into a number of sets for cross-validation without neglecting to randomise their ordering. By taking a random series of numbers between 1-300 and copying inputs and outputs into new variables we can get a good mixture of digits in each set. I have chosen to divide the training set into 5 sets of 60 (mixed) samples. I then train the SVM using 4 of these [2]and check the performance for the 5th set. This is a standard K-fold cross-validation. For each of the 5 cases, the performance is recorded to infer a possibly optimal polynomial degree (as derived from the confusion matrix). The degree is in fact changed for each one of these cases, or else no concrete comparison will be made. The reason it is required to change the degree value for *different* sets of training and validation is the need to get an unbiased result. In other words, we want to ensure the degree is not too data-specific, that is, suitable only to a certain training and testing sets (think of data that is poorly homogenous. This would have a similar effect). The choice of degree is almost arbitrary and is based on heuristics and the viewing of previous results. This means that if a choice of, let us say, degree=3 was better than that of degree=2, then trying degree=4 is a logically sensible step.

This polynomial degree (which is assumed optimal in our case) is then used to train the SVM with the other parameters set to their default values. The testing set is then used to measure the performance of the SVM created and the confusion matrix is displayed.

Oil Pipeline Network

The procedure is almost identical to the above case. These are a few differences which can be summarised as follows:

- The data is extracted from the data files differently. There is no need to mix the data and union of different sets, as well as division of sets, is more trivial.

[1]The sets are sorted at this stage, starting with examples of '0' and finishing with examples of '9'.

[2]Different sets are used each time so there are 5 cases in total. It is more of a combinatorial problem.

- The size of the whole set given is 1000, 500 for training and 500 for testing.
- The cross-validation uses a K-fold of 5 (k=5), i.e. 100 inputs in each set (400 for training, 100 for validation).
- The outputs require some basic conversion as 1-of-N output representation is not appropriate for an SVM. A simple conversion from [bit,bit,bit] ▷ [number] is carried out. A simple instance would be [0,1,0] becoming a 2.
- The classification has 3 possible distinct outputs, as oppose to 10 in the first case. If we ignore the complexity of the problem which is a major factor, then there is an expectation for substantially better performance (low error rate). Of course this would not have been the case if we tried to distinugish between the letter 'O' and the digit '0'[3] (zero).

The highest degree that I tried usually gave me the best results. This should not be considered surprising because higher polynomial functions are more flexible (although their behaviour can become peculiar). As an example, here is the confusion matrix returned for the oil pipeline network at degree 5:

0.8971	0.1029
0	1

Table 1: Confusion matrix

SVM vs. MLP

The performance of SVM is excellent. MLP results were good, but the confusion matrix shows the SVM's superiority.

1. **For the digits example:** as was proved in a benchmark for postcode analysis, SVM has one of the lowest error rates for tasks of this nature.
2. **For the oil pipeline network example:** the confusion matrix above shows the strength of SVM for this task too.

It is then only left to consider other advantages and drawbacks of each approach, where appropriate criteria would be speed, complexity, robustness, suitability, generalisation, etc.

[3]Contrarywise to a task such as classification of shapes, e.g. lines, rectangles and triangles.

RBF Kernel SVM

The transition from a polynomial SVM to an SVM with an RBF kernel primarily involved altering the function being called. The choice of parameters (C and Gamma) for good performance was the crucial bit because the former SVM worked better than the latter for inadequate values of these two parameters.

The following steps, as suggested by the SVM guide, were carried out to improve the performance of the SVM:

Transformation: data format is an issue which was solves in the previous part.

Scaling: in the case of the digits example, the original binary vectors which describe the image were used to train and test the SVM. This is more efficient because kernel values usually depend on the inner products of feature vectors. Large attribute values may cause trouble and degrade the performance of the SVM.

Kernel: this has been chosen as the SVM type to achieve good results.

Cross-validation: in order to find good values of C and Gamma, cross-validation, as in the previous part, was used. Rather than altering the degree, C and Gamma have been modified to achieve better results (again, in accordance with the confusion matrix). More details on these choices are to be described later (parameter search is an issue which is worth exclusive elaboration).

Parameters: the best parameter values, which were found in the previous stage, are now used to train the SVM using the **whole** training set.

Testing: using the testing set to evaluate the performance of the SVM.

On choice of parameters C and Gamma

A technique called grid-search was used to gradually find better assignments for C and Gamma. The test of performance was applied to different training and validation sets for the reasons explained earlier. Exponentially growing sequences was said to be a good method in practice; that is why this method was used in the code. C = $(2^{-n}, 2^{-n-1}, ..., 2^{n-1}, 2^n)$ and likewise for Gamma was the nature of the series of values to be tried.

An ideal way of practically using the grid-search approach is to have 2 nested loops where C and Gamma are slowly increasing or decreasing. An even better approach would be to view the differences between 2 consecutively returned confusion matrices and adjust the values accordingly. This is similar to the approach of finding global minima, but it is not guaranteed to avoid getting trapped in local minima.

As I am using my 400Mhz, 26MB RAM machine to run these tests, the above methods are far too computationally expensive to be exhaustively applied. I manually tested and tried good values of C and Gamma (based on the confusion matrices I viewed – a somewhat supervised adjustment) and found good performance at C = 2^{-5} and Gamma = 2^{-15} for the digits example. Gamma = 2^{-1} and C = 2 are rather good values for the oil pipeline network example.

Part II: Principal Component Analysis

PCData Problem

Algorithm

At the very start, the 1000x8 matrix is read and stored. Its eigenvalues and eignevectors are then extracted and its dimensionality is reduced to 2D and 3D with the help of the projection rules. The data can then be visualised and the eignenvalues inspected.

The second stage involves noise. This randomly generated noise is applied to the 2D and 3D data and plots are generated as in the above case.

Results

Let us look at some visual results:

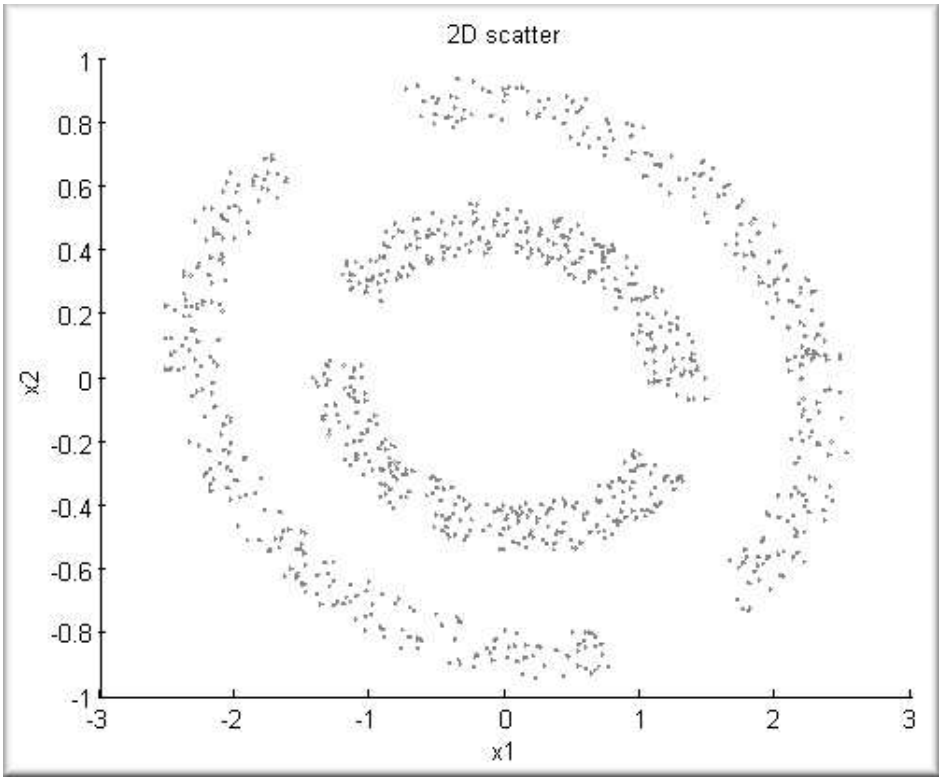

Figure 1: 2D scatter

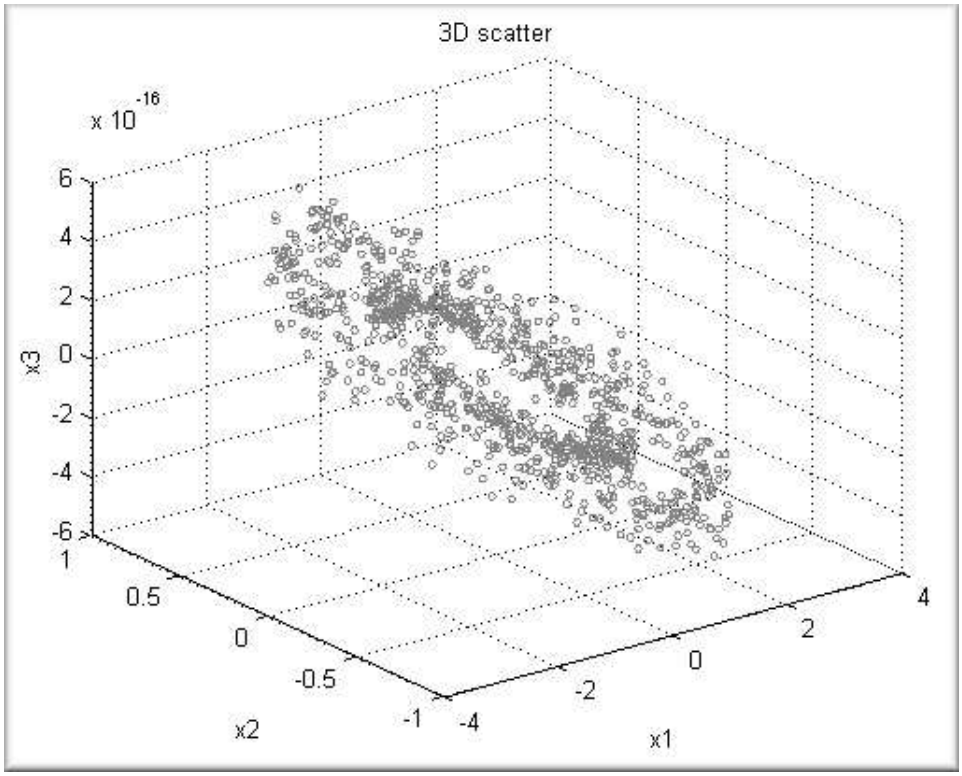

Figure 2: 3D scatter

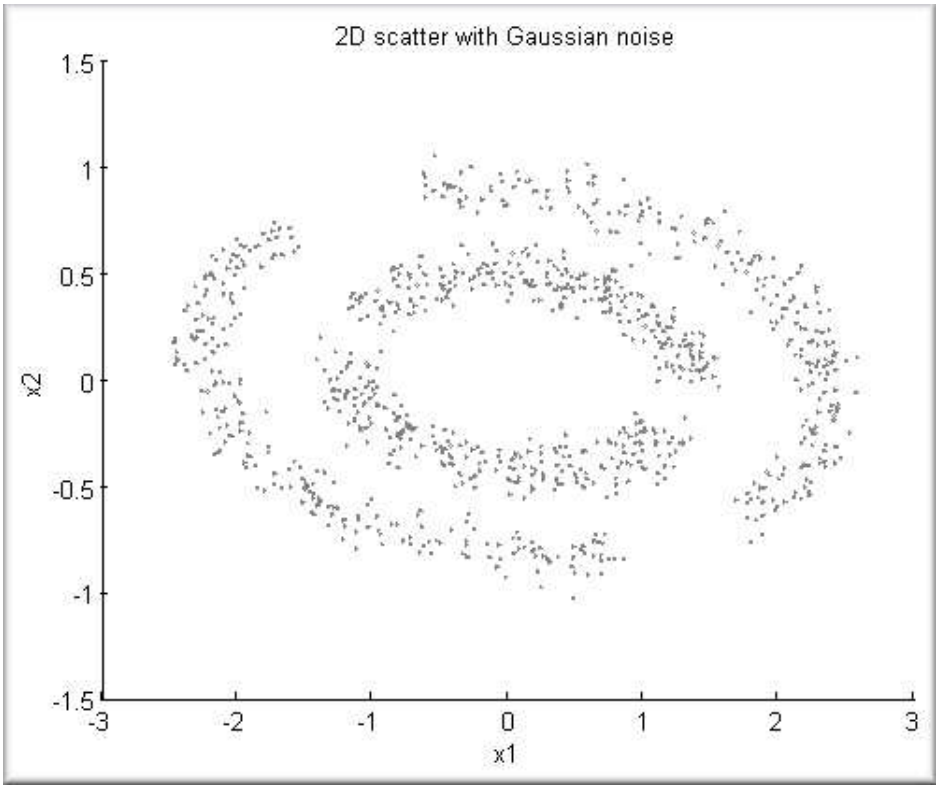

Figure 3: 2D scatter with Gaussian noise

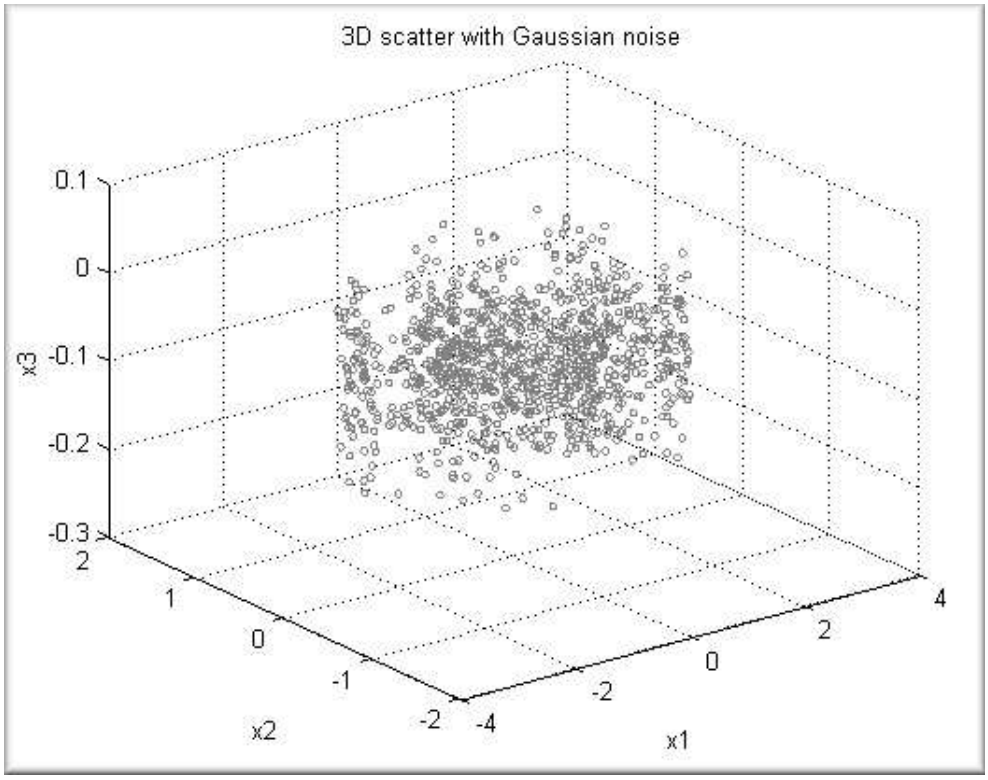

Figure 4: 3D scatter with Gaussian noise

It is clear to see that there are 4 clusters in the data. From the plots to which no noise has been applies it appears as if only 2 dimensions can suitably describe the data. In other words, principal component analysis, which is concerned with dimensionality reduction, allows us to align the data to just 2 axes without significant loss of information.

A quick look at the eigenvalues (listed in descending order) reveals that only 2 vectors are non-zero which confirms the argument above.

The noise applied to the data blurs out the dimensionality property of the data. It also alters the previously solid form of the clusters. The eignenvalues confirm that the data is now better described in 3D as the Gaussian noise perturbed the points so that they no longer lie on the same plain. As the noise is relatively small in extent, the data can still be visualised rather well in 2D.

Crabs Problem

We will begin by discussing the algorithmic solution to this problem, followed by figures that support the conclusions described.

We start by reading in the data from the file **crabs.txt**. The input vectors are stored in a matrix of 200 rows. We hold the sex data and the species data separately in column vectors. We then move on to extraction of the eigenvectors and eigenvalues from our input vectors. These are used as before to reduce the dimension of the data to 2D and 3D.

The part that is more unique to this problem (with respect to the previous one) begins here. We split the 2D and 3D data into 2 sets, once according to species and once according to the sex[4].

A scatter of points is then used to visualise the data, with different colours to distinguish between the two sets as divided by the current criterion (species or sex). One scatter illustrates the 2D nature of this problem and one shows the less pictorially useful 3D scatter.

The following 2 figures show the grouping (or no grouping) of samples belonging to identical species or sex. There are four figures in total, but in order to save space, only a couple are passed on for analysis in this document.

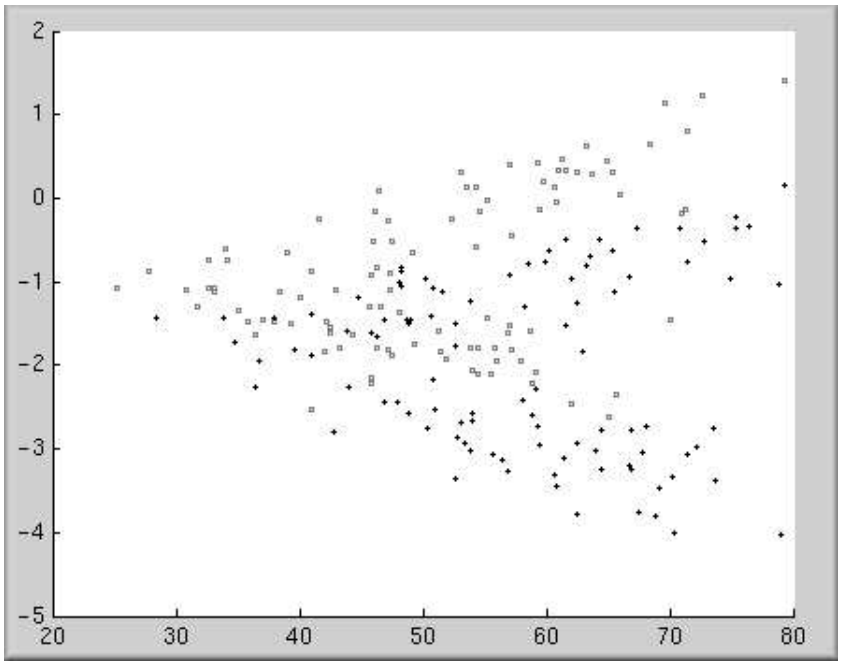

Figure 5: Species classification

[4]This process involves inner counters within a loop as we do not know the size of any of the two sets.

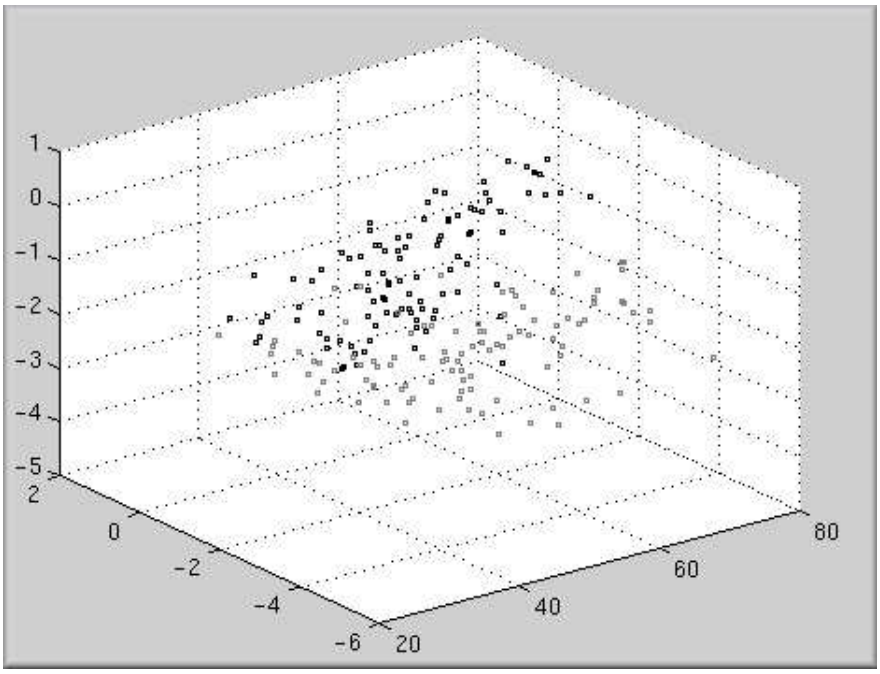

Figure 6: Sex classification

In the first of the two figures, dark (black) dots represent crabs of species O, whereas the gray (shaded) dots represent crabs of species B. In the second, dark (black) dots represent males and gray (shaded) dots represent females.

Analysis

Sex and species were both characteristics that allowed good classification of crabs. In 2D and 3D (quite probably in 5D too) there was a good spread of points from two somewhat separable classes. The spatial location could indicate which crab was of which sex and species. In 2D we could imagine a *line* that divides the two classes, e.g. f(x) = ax+b for a linear classifier and in 3D that would be a *plain*.

Auto-encoder Problem

Algorithm

The data is pulled out from the file **crabs.txt**. We then need to transform the data to have zero mean for each of the inputs – that is – the columns must all add up to zero.

Here is how we go about transforming the data. Firstly we initialise counters, each of which will accumulate the values of one input. We

then find the average value of each input (simple division of the sum by 200). We subtract that average value off each input value to obtain a type of "normalised" or *centred* collection of values. This helps visualisation considerably.

With the inputs in the wanted form, we can now create an MLP and train it using the data's inputs as outputs. When training is completed, examples are passed in to the MLP and we see the activity of the hidden nodes as returned by **mlpfwd**. We vertically append all tuples (1x2 matrices) returned by this function and display the result using a call to **scatter**.

Results

The figure below shows several clusters of data in 2D. Most dots lie near the Y-axis, but there are grouping of points at the top right and the virtual line passing at f(x)=1. It is possible that these clusters somehow represent a commonality which is to do with sex and/or species. It is too hard to draw such conclusions though. The best assumption is that the examples have a distinguishable nature of some sort.

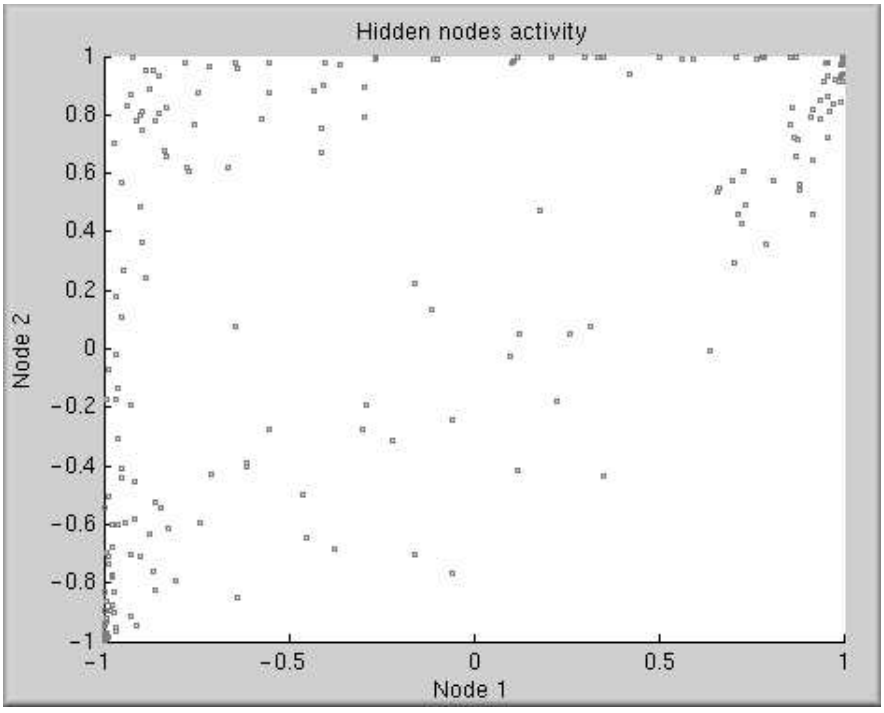

Figure 7: Hidden nodes activity

Note: The data in **crabs.txt** is not homogenous[5] so although I explicitly divided the 200 examples into 2, I trained and tested the MLP using the same data. An alternative implementation would involve randomising the ordering in the file data (quite strictly, we must be careful to just manipulate row numbers, for example using **randperm**) and only then divide it into two.

Part III: Hidden Markov Models

Log-scaled Viterbi Algorithm: Motivation

Algorithm

We shall begin by opening the data file and viewing its contents for general familiarity. Several subsets of the data are then constructed. Each of these *datasubset_x* holds the data numbered 1, 2,..., *x-1*,*x* from the original large file. For realism, I adhered to the probability values used in the lecture notes:

Start_Transition_Fair	100%
Start_Transition_Loaded	0%
Fair_Emission_Set	[1/6,1/6,1/6,1/6,1/6,1/6]
Loaded_Emission_Set	[1/10,1/10,1/10,1/10,1/10,1/2]
Transition_Fair_To_Fair	95%
Transition_Fair_To_Loaded	5%
Transition_Loaded_To_Fair	10%
Transition_Loaded_To_Loaded	90%

Table 2: Model parameters

In the above table, '[' followed by ']' is a representation of a set[6]. We now call the given Viterbi algorithm and store the joint probability values in a vector that can be plotted as shown below:

[5]The data from Brian Ripley's book is sorted by species and sex.
[6]The *n*-th entry in this vector corresponds to die throw of value *n*.

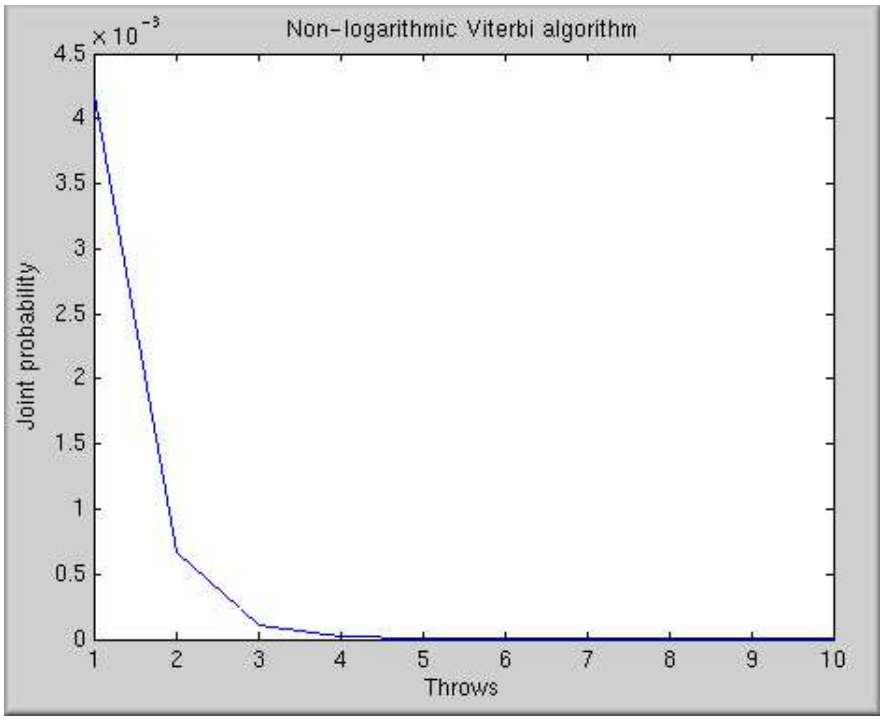

Figure 8: Veterbi algorithm

We can see how quickly the joint probability values shrink[7]. On a standard computer, even floating-point representation of numbers will soon fail and return a flat zero as the joint probability. This makes high-quality comparison a difficult task and log-scaling is one possible solution, which gives inaccurate (even incorrect), nevertheless *manageable* results.

Log-scaled Viterbi Algorithm: Evaluation

A new function **log_viterbi** was implemented. Its description was available in the exercise scripts and need not be repeated. It was merely a mutation of the given algorithm **viterbi**. The aforementioned procedures are called yet again, but the new Viterbi function is used instead. The results can be seen in Figure 9.

[7]Joint probabilities are a product of multiplication, hence they usually grow with respect to $1/K^n$ as a function of the number of probabilities n. K is the number of transitions and the number of emission as they alternate. $1/e_m^n$ * $1/t_m^n$ is another way of expressing this relation, where e_m is dependent on the number of emissions and their relative probabilities. The same argument applies to t_m.

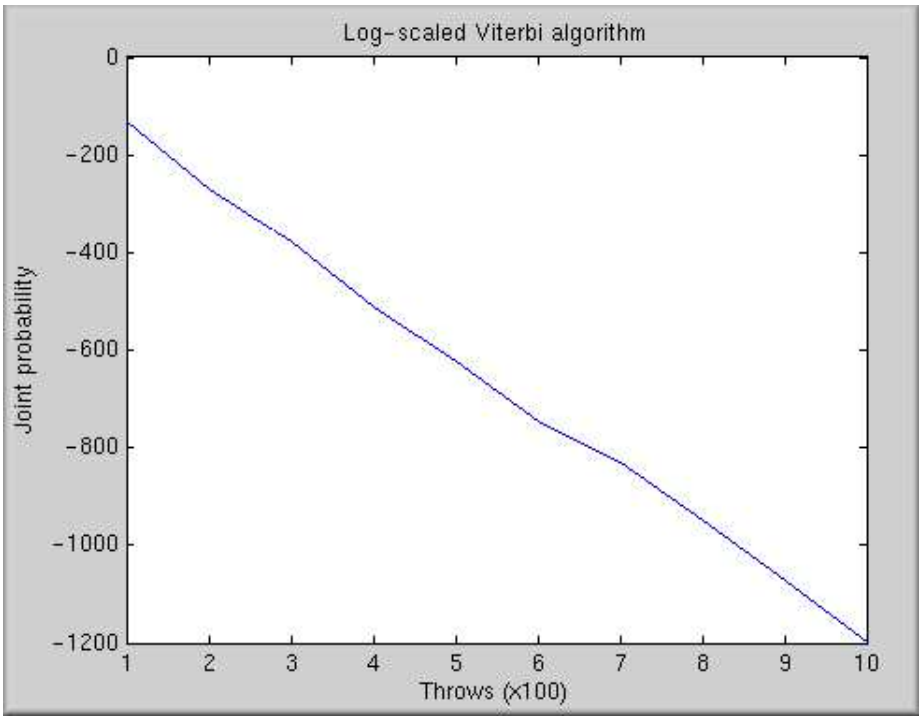

Figure 9: Log-scaled Viterbi algorithm

The figure clearly shows that the joint probability now linearly decreases. It may not correctly reflect the nature of the probability as function of the size of the data, but it at least guarantees a *traceable* problem. It also means that the probability returned remains *finite*.

Log-scaled Viterbi Algorithm: Application

The newly-created algorithm was applied to each of the 100 sequences. the algorithm returns us the perceived path which is estimated by the joint probability sought. It can then be compared to the real path given in **hpath** in the following way:

- For every sequence in the test data:
  - Initialise a counter for the current sequence by setting it to 0.
  - For every site in the sequence:
    - If the hidden state predicted is that which is given in **hpath**, increment the counter of the current sequence.
- Return the set of 100 counters.

We now have the count of correct predictions for each one of the data set sequences. We can plot this as an histogram. The figure below shows the required results, although quite suspiciously there are more incorrect predictions than correct ones.

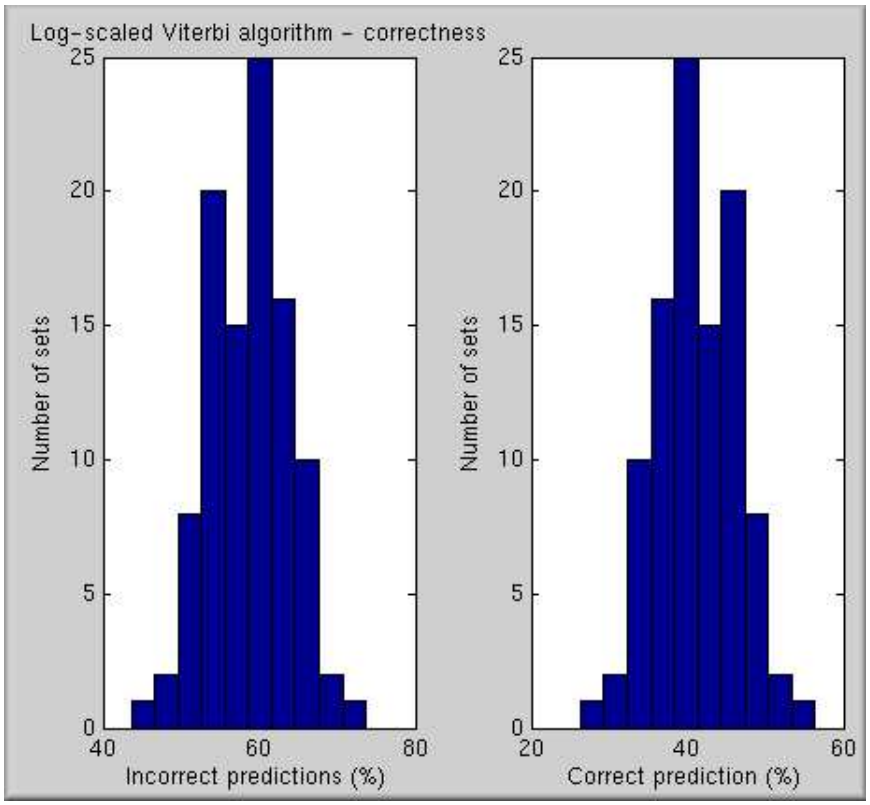

Figure 10: Viterbi prediction histogram

Parameter Estimation

Let us assume that the model parameters that I decided to stick to are unknown. If I have access to **hpath** and **data**, then I can estimate the values of these parameters from sample statistics.

I count all transitions T_{kl} and emissions $E_k(b)$ by scanning the data that I have. I then take advantage of the maximum likelihood estimators (whose equations I do not yet know how to code in LaTeX). In principle, these will give me good estimates of the transitions and emission. The clear drawback is that an immensely large amount of data is essential to get good estimates and that amount of data is not always available in hand.

For the whole data[8], the estimates I get for the transitions are:

k_{fl}	0.0504
k_{ff}	0.9496
k_{lf}	0.0998
k_{ll}	0.9002

Table 3: Transition values

The numbers above are quite accurate (see the beginning of the section where the real values are revised).

The algorithm utilised is as follows:

- Initialise counters for all transitions (there are 4 in our case).
- For all 100 sequences:
  - For all sites from 2..1000:
    - Check the type of transition by comparison with the predecessor in the sequence.
    - Increment relevant transition counter according to the type of transition detected above.
- For each of the 4 transitions, set $t_{kl}$to be T_{kl} / (T_{kl_1} + T_{kl_2}+ ... + $T_{kl_{n-1}}$+ T_{kl_n}) where *n* is the number of possible transitions from k.

The algorithm is very similar for the emissions and will not be overly discussed. It involves many more transitions though which causes a slight bit of clutter.

To demonstrate the effect a large number of sequences has on the accuracy of estimation, I modified **p3_3.m** to become a demo that runs the process above *k* times for an increasing *k* number of sequences. The results are shown below[9].

[8] 100 sequences of 1,000 throws have 99,900 transitions.

[9] I wanted to run this with k set to 100, but even the new strong machines in the engineering lab, e.g. eng038 and eng042, found it very hard to cope with.

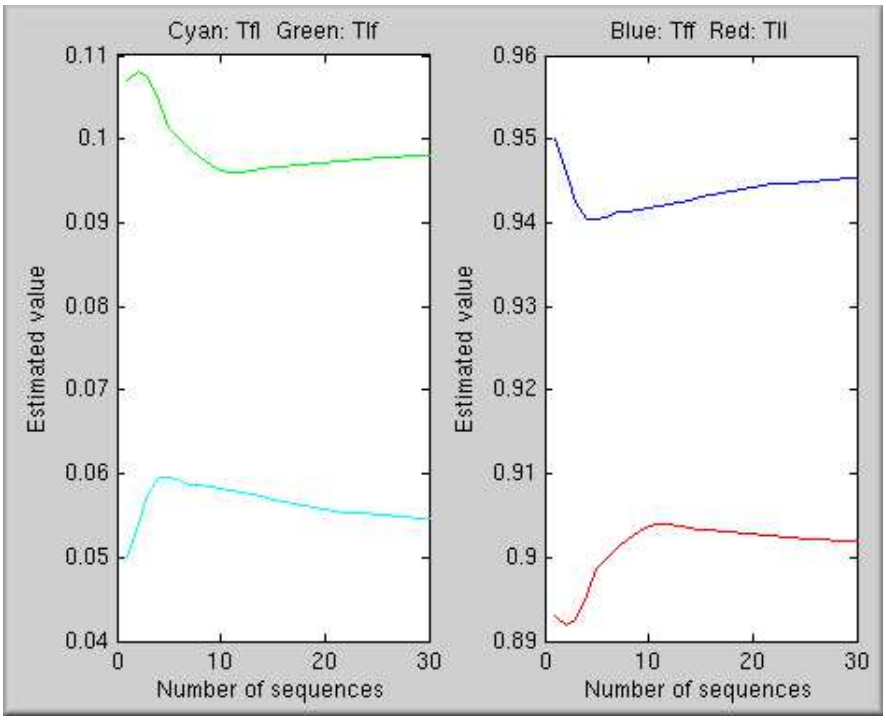

Figure 11: Estimating transition values

For completion, here are the estimations for the emission probabilities[10]:

$E_f(1)$	0.1657
$E_f(2)$	0.1612
$E_f(3)$	0.1637
$E_f(4)$	0.1614
$E_f(5)$	0.1634
$E_f(6)$	0.1845
$E_l(1)$	0.1055
$E_l(2)$	0.1078
$E_l(3)$	0.1042
$E_l(4)$	0.1073
$E_l(5)$	0.1047
$E_l(6)$	0.4705

Table 4: Emission values

[10]All sequences were taken into account to generate the emission values. Increasing number of sequences k is expected to exhibit the same behaviour as in the previous figure.

If **hpath** was unavailable and the model parameters needed to be estimated, the Baum-Welch training (EM-algorithm) would be a reasonably good approach of solving this problem.

Acknowledements

Many thanks to Mark O'Leary for allowing me to labour on this coursework in my office at work.