COM2004/3004 Lab Week 5
Linear Classification

Objectives

- **To gain some practical experience with linear classification.**
- **To reduce Bayesian classifiers to linear classifiers by constraining the covariances of the classes to be equal.**
- **To compare the performance of a generative model and a discriminative model on a challenging classification task.**

1. Background

In today's lab we will be revisiting the Bayesian classification code that we used last week but we will constrain the system to have equal covariance matrices effectively turning it into a linear classifier. We will then compare this `generative' approach with the `discriminative' approach that we saw in the lecture on Friday: learning the linear classifier parameters directly by using the Perceptron learning algorithm.

2. Introduction

This week we will be using another data set from the UCI machine-learning repository: abalone data. An abalone is a type of see snail. The age of a specimen can be determined by cutting the shell through the cone and counting rings through a microscope (rather like trees), but this is a time consuming and expensive procedure. The task here is to predict the number of rings given simple external measurements of the weight and dimension of the animal. For the dataset we are using the real values are known (i.e. the rings were counted after the snails were measured). Results vary from 1 to 29 rings, so this would usually be treated as a 29-class classification problem. To simplify things a little I have regrouped the data into just two classes of roughly equal size: young (<10 rings) and old (>=10 rings). I have also only taken the female samples. There are 7 measurements[1] (which are all quite highly correlated) that are to be used to predict the class label.

Compared to the wine classification task this task is quite challenging. It will be impossible to get 100% correct because the classes are not linearly separable. Further, most of the specimens have either 8, 9, 10 or 11 rings and so lie close to the young/old borderline. However, you should be able to get percentage correct scores that are considerably higher than the 50% that would be expected by guessing alone.

3. Obtaining the data

Download the data from MOLE and save it in a file called `ablone.dat`. Load the data into MATLAB. The data will form a matrix with 1,306 rows and 8 columns. Each row is a separate

[1] For details visit http://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.names

sample (i.e. a different snail). The first column stores a class label (1 or 2). Columns 2 to 8 are the results of the 7 length and weight measurements. Note that, like last week, the features are stored as columns and the samples as rows. This makes things easier when using MATLAB but in our lecture notes and many textbooks the opposite convention is used.

4. Generative modeling: Bayesian classification with equi-covariant multivariate normal distributions.

Compared to last week, there are a lot more samples (1,306 compared with 178) so we will not worry about leave-one-out testing, instead, we will simply cut the data into equal sized testing and training sets like we did for the first half of the lab last week.

By adapting the code you wrote last week, evaluate the performance of a Bayesian classifier using multivariate normal distributions with full covariance matrices. When considering changes to the code note that the main difference is that this week there are only two classes rather than three. (If you wish you may trying wrapping the code into a function and seeing if you can design it so that it works for any number of classes.)

(If you did not complete the first part of last week's lab there is some code on MOLE in the Week4 lab folder – but you will still need to adapt it to make it work for this week's data.)

How well does your classifier perform? (The score will probably be in the 60-70% range for this task, so don't worry if performance seems a lot poorer than last week).

Using equal covariance matrices.

If you correctly followed the same procedures as last week you will have estimated a separate covariance matrix for each class. These matrices will not be equal and so the system will not have been a linear classifier. In order to reduce it down to a linear system we need to ensure that there is only one covariance matrix. There are different ways that you might imagine doing this:

First, you might imagine simply estimating a single covariance matrix from the complete training set before dividing it into classes. This produces a single matrix but it is not the correct thing to do. We want the matrix to represent the spread *within* the classes, if you simply train a model using the full training data set it will also be capturing the spread *between* the classes.

Second, you could imagine *averaging* the two class-dependent covariance matrices. This is closer to the correct thing but it doesn't take care of the fact that the classes might have had unequal numbers of examples. The best approach is to first move the centres of the two classes onto the same point and then treat them as a single class. To move the classes onto the same point you can simply subtract the class mean vector from each data sample.

E.g., for each sample in class one you need to compute,

```
abalone1_train (i,:)- mean1
```

and for each sample in class two you would compute,

```
abalone2_train(i,:) - mean2
```

You can do this without a loop by making a matrix where every row is a copy of the `mean1` vector and subtract the matrix from the `abalone1_train` matrix and likewise for `abalone2_train`. To construct the repeated mean matrix you can use the MATLAB `repmat` command. Type `help repmat`.

Go ahead and modify your code so that a single covariance matrix is computed called, for example, `cov_global`.

You can now redo the classification using this single matrix in both `mvnpdf` evaluations

```
p1 = mvnpdf(abalone1_test, mean1, cov_global);
p2 = mvnpdf(abalone2_test, mean2, cov_global);
etc
```

What is the new result? How does it compare with the result when using the more flexible non-linear classifer (i.e. when using two different covariance matrices)?

If we wished we could represent this linear Gaussian classifier in the standard linear classifier form, i.e. g(**x**) = **w**'**x**+w_0 and output w_1 if g(**x**)>0 else w_2. The parameters **w**' and w_0 would simply be a function of the covariance matrix and the class means that we have learnt from the training data. Actually, although this would involve an extra bit of calculation, once we had computed **w**' and w_0 once, all test points could be classified very quickly by evaluating g(**x**) directly (e.g., for an *N*-dimension problem this amounts to just *N+1* multiplications, *N* additions and a comparison against 0)

5. Disciminative modeling: the Perceptron learning algorithm

In the 2nd half of the lab we turn to look at *a discriminative* approach to classification in which the **w** parameters needed for the evaluation of g(**x**) are learnt directly. We will use the Perceptron learning algorithm that was discussed in the lecture last week.

The Perceptron learning algorithm is quite easy to implement, but to write it in an efficient vectorised form requires a little thinking. In order to save time I've provided an implementation for you. Visit MOLE and download the code `perceptron.m` from the lab class folder. The `perceptron` function takes labeled training data as input and outputs the learnt **w** parameters. The implementation is very similar to the code that we examined in the lecture except the inner loop has been vectorised, i.e. all the points are classified in a single line, and the gradients are accumulated in a single line. Take some time to examine the code. Don't worry if it is not immediately clear how each line is working, but just satisfy yourself that you understand what each line is meant to be doing.

Now use `perceptron.m` with the same training data that you used previously in order to learn the weights. You will need to read the documentation at the head of the function so that you understand what inputs are required. In particular, note that the class labels have to be in the form of a vector storing -1's and +1's. So you will have to manipulate your training data a little because it currently has the classes labeled represented as 1 and 2.

Experiment with different learning rates and different numbers of iterations. The function returns the number of errors that are made on the training set. You want this number to be as low as possible.

Evaluating the classifier: You now need to evaluate the **w** vector that the learning algorithm has produced. To do this you will need to evaluate **w'x**+w_0 for each element in the test set and generate a label by comparing the result against 0. Then compute the percentage of labels that match the correct test set labels. It is probably best to write a little function to do this.

How well does the new classifier perform? How does performance compare with the linear classifier that you built using Gaussians in the first half of the lab?

6. Notes

You should find that although the new classifier is again linear, its performance is a little better. You might be confused about this because in lectures we previously spoke about Bayesian classifiers being optimal – their decision boundaries are positioned so as to minimize the number of classification errors. However, this is only true in the theoretical sense that they are optimal if we use the **correct distribution** for p(**x**|w). In practice we don't know this distribution so we estimate it from the training data. This is typically done by assuming p(**x**|w) has some general shape (e.g. Gaussian) and then estimating specific parameters, e.g. means and covariances, that best fits the training data. This would be fine is our initial assumption was correct, but the fact is that the data is very unlikely to fit some nice well-behaved distribution like a Gaussian. For example, it might be approximately Gaussian but slightly *skewed*, or with slightly *elongated tails* etc etc. So even with the best possible parameter tuning there will still exist a discrepancy between the p(**x**|w) that we've estimated and the true distribution. If we aren't using the correct p(**x**|w) then all bets are off and we can't guarantee that the model we have chosen will be the one that minimizes errors.

The *discriminative* approach on the other hand is doing something that is closer to directly minimizing errors. It is doing this without regard to the distribution of p(**x**|w). This more pragmatic approach often works better particularly in cases where p(**x**|w) has a complicated distribution that is hard to parameterize. It is also perhaps closer to what we do as humans, e.g. we are often not aware of small differences within classes (e.g. in speech, all 'b' phonemes sound roughly similar) but we are very sensitive to small changes at the borders between classes ('b' and 'p' are perceptually quite different, but if you look at features extracted from their sound signals they can be extremely similar).

Programming Challenge

7. Programming exercise

In the lecture notes two versions of the Perceptron learning algorithm are discussed. The first works in *batch mode* like the code you have used here, i.e. all points are first classified and then the weights are updated. The second, works in an *online mode* and processes thee samples one at a time, i.e. the w parameters are adjusted directly after processing each sample.

Copy perceptron.m to perceptron2.m and then rewrite it so that it works in the online manner. Does it produces better results for the Abalone task? (I haven't tried this myself yet so I will be interested to hear if anyone gets this far.)