COM2004/3004 Lab Week 4
Bayesian Classification

Objectives

- **To gain practical experience of Bayesian classification.**
- **To see how a classifier can be evaluated using a separate training and test set.**
- **To compare the performance of `diagonal covariance' and `full covariance' multivariate Gaussian models.**
- **To learn how to perform `leave-one-out' testing.**

1. Background

In the lectures we have studied the theory of Bayesian classification and in the tutorial we have looked at trivial 1-dimensional and 2-dimensional examples. In this lab class we are going to use the same techniques but apply them to a genuine 13-dimensional classification problem – wine classification!

The lab sheet this week is a little less `hand-holdy' than previous ones. For each section, first read the instructions carefully and make sure that you understand what is being asked before typing any MATLAB code. If you are confused or get stuck put up your hand or ask a friend for help.

2. Introduction

The data you will be using are the genuine results of chemical analyses of wines grown in the same region of Italy but produced from three different varieties of grape. The analyses have determined the quantities of 13 chemical constituents. The task is to use the results of the chemical analyses to identify which of the three grapes was used to produce an unlabeled wine.

3. Obtaining the data

Download the data from my webpage or from MOLE and save it in a file called `wines.dat`. Load the data into MATLAB. The data will form a matrix with 178 rows and 14 columns. Each row is a separate sample (i.e. a different wine). The first column stores a class label (1, 2 or 3). Columns 2 to 14 are the results of the 13 chemical analyses. Note that 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 convention is the opposite.

4. Notes: Training and evaluating a classifier.

As discussed in lectures using a classifier involves two stages. First the classifier is trained using samples for which we know the correct class labels. *Training* is the process of estimating the classifier's parameters, e.g. the means and variances of $p(x|w)$, etc. Once the classifier is trained we can use it to label unknown data. Generally however we first want to evaluate the classifier. This is called *testing*. Like training, testing requires data for which we know the correct labels. We

present a sample to the classifier and compare the classifier's output to the known correct label. We then measure the percentage of the data that has been classified correctly and hope to get as close to 100% as possible.

For reasons that will become clearer later in the course, it is very important that the data used for testing the classifier are not the same as the data used for training it. So we usually start by partitioning our data into a separate *training set* and *test set*.

5. Preparing the wine data

We are going to prepare the data by separating out the samples for each of the three classes, and then dividing the data for each class equally between training and testing sets.

To find samples belonging to class 1 we need to select the rows of the matrix for which the first column contains a 1. If the data was stored in a matrix X we could select these rows as follows

```
X(X(:,1)==1,:)
```

Use this idea to separate the data into three matrices that we will call `wines1`, `wines2` and `wines3`.

We want to split `wines1`, `wines2` and `wines3` into equal training and testing partitions. The easiest way is to put the odd rows in the training set and the even rows in the test set, e.g. the odd rows of a matrix X could be selected using

```
X(1:2:end, :)
```

Use this idea to make matrices called `wines1_train`, `wines1_test`, `wines2_train` etc.

Finally combine the test data for each class back into a single test data matrix.

```
wines_test=[wines1_test; wines2_test; wines3_test];
```

This will make things easier later.

(Store all your commands in a script so that you can reuse them again later.)

6. Training the classifier

We are going to use a multivariate Gaussian distribution to represent $p(x|\omega)$ for each class. So we need to estimate the mean vector and covariance matrix for each class. In the first instance we are going to assume that the features are uncorrelated and use a covariance matrix with 0 for all elements off the diagonal. This is no doubt a poor assumption but in means that our 13x13 covariance matrix only contains 13 variances that need estimating rather than 91 parameters (why 91?). Diagonal covariance matrices make the probability evaluations very quick to compute and so they are popularly used in classification systems.

We can estimate the 13-element mean vector *for each class* using `` `mean' ``. Note though that you should only use columns 2 to 14 – column 1 stores the label. Store the results in variables called `mean1`, `mean2` and `mean3`.

We can estimate the variances (i.e. the elements along the diagonal of the covariance matrix) using the `var` function. Store the results in vectors called `var1`, `var2`, and `var3`

7. Evaluating the classifier

Once you have estimated means and variances for each class the training stage is complete. You now need to use your classifier to process the test data and to compare the classifier outputs with the known test data labels.

To perform an actual classification we need to compute $p(x|\omega\_1).P(\omega\_1)$, $p(x|\omega\_2).P(\omega\_2)$ and $p(x|\omega\_3).P(\omega\_3)$ and see which gives the highest score. We will assume that the prior probabilities are equal, so we can simply compare $p(x|\omega\_1)$, $p(x|w\_2)$ and $p(x|\omega\_3)$.

To evaluate the multivariate Gassian pdf $p(x|\omega)$ for some known x we can use the MATLAB function `mvnpdf` (multivariate normal probability density function). This is part of the MATLAB statistics toolbox. Type `help mvnpdf` to see the documentation.

```
mvnpdf(x, m, C)
```

x is a vector containing a data sample, or a matrix containing many data samples stored as rows. m and C are the mean and covariance parameters. C is generally a covariance matrix, but if you pass a vector to C the function constructs a diagonal covariance matrix with the elements of the vector C along the diagonal, i.e. so you can simply pass your variance vectors, `var1`, `var2` or `var3` as the final parameter.

So now, for each of the three classes, we are going to evaluate $p(x|\omega)$ for all of the x's in our test set

```
p1 = mvnpdf(wines_test(:,2:end), mean1, var1);
```

Note again that the label column has been excluded.

Do this for each class to arrive at three vectors `p1`, `p2` and `p3` each with as many elements as there are samples in the test data, say n. We will now form these outputs, `p1`, `p2` and `p3` into a single matrix, `p`, with n rows and 3 columns, i.e. `p1` in the first column, `p2` in the second and `p3` in the third.

To compute the class label we now look at each of the n rows of `p` and find the index of the column containing the highest score. Store the n indexes in a vector called `labels`. We can do this with a single line of MATLAB using the `max` function.

We now need to compare the class labels that the classifier has output (`labels`) with the true labels that are stored in the first column of `wines_test`. Count how many are the same and divide by the total number of samples in the test set. Finally multiply by 100 to get the percentage correct.

What percentage would you expect to classify correctly if you had just guessed the label? Is your classifier doing better than this?

8. Improving the statistical model

Our diagonal covariance model was quick to compute and to evaluate but it is a poor model of the true distribution of the data. Some of the features are quite highly correlated. We will now repeat the process but this time using a full covariance model. If you have saved all of your commands this is very easy.

Rather than estimate a vector of variances using `var`, you can now estimate the 13x13 covariance matrix using the MATLAB function `cov`.

Estimate the covariance matrix for each of `wines1_train`, `wines2_train` and `wines3_train`, storing the results in `cov1`, `cov2` and `cov3`.

Testing is just the same as before but now you simply pass `cov1`, `cov2` and `cov3` as the final parameter of `mvnpdf` instead of `var1`, `var2` and `var3`.

Repeat the `mvnpdf` evaluations and compute a new `p` matrix. Use `max` again to get the output labels. Score the new labels by matching against the correct test set labels.

Has performance improved?

9. More accurate evaluation

The problem with this task is that after splitting the data into training and test sets there is barely sufficient data to accurately estimate the classifier parameters (remember that sample means and variances can be quite inaccurate when *N* is small). Also there is not really enough data for accurately evaluating the classifier: just 1 extra error changes the performance evaluation by more than 1%.

Ideally we want to be able to use all the data for training and all the data for testing – but it was said earlier that we need to keep training and testing data separate, i.e. testing on data that has been used during training does not produce valid results. So what is the solution?

The solution is to do something called 'leave-one-out' testing. In this approach we use just the first sample for testing and train using the remaining *N-1* samples. But then we repeat the exercise using the 2nd sample for testing and the other *N-1* for training, and then again using the 3rd sample for testing and so on until we have tested all *N* samples. We can then report the combined result of all *N* tests.

The obvious downside to this is that we now have to train *N* different classifiers, i.e. each using the full set of data but with a different test sample omitted. However computers are going at doing repetitive things, so for small problems leave-one-out testing is a practical approach.

By using a loop around the training and testing code that you have written, and by making the necessary modifications to the way in which the data is split into training and testing sets, implement a leave-one-out evaluation scheme for the wine data.

Use your code to re-evaluate the classifier using both the diagonal-covariance and full-covariance schemes. What are your new results and how do they compare to those you had earlier?

10. Selecting features

Say that you were told that the current wine classification system was too expensive to run. Performing 13 separate chemical analyses is taking too long and the chemists want to reduce the number of tests down to 6. Which 6 features would you choose to ensure best performance?

Notice that the code you have written is equally valid for tasks of any dimensionality, ie. the constant 13 should not have to appear anywhere in the code. So you could evaluate a 6-D version of the classifier simply by selecting the label column and 6 additional columns as the very first step of the program. e.g.

```
wines6D=wines([1,2,5,7,9,10,13],:)
```

Can you find 6 features that perform almost as well as the full 13?

In the 2nd half of the course we will be looking at some techniques for solving this problem but for now either use trial and error or try looking at the histograms of individual features to find dimensions along which the classes appear well separated.

Email me your 6 best features and the score they achieve.

If there are sufficient entries results will be announced in the lecture session.

Programming Challenge

11. Programming exercise

The MATLAB function randn(N) produces an NxN matrix containing random numbers drawn from the standard *normal* distribution (i.e. mean=0 and variance=1). The function rand generates numbers drawn from a *uniform* distribution between 0 and 1.

Write your own implementation of randn called my_randn. Your function should produce random numbers that are approximately normally distributed using only rand and other MATLAB functions (but not using randn!).

Hint: http://mathworld.wolfram.com/CentralLimitTheorem.html

12. What next – Friday Lecture

Classification: the Perceptron classifier.