Performance Assessment

Gordon Ross

Performance Assessment

We have discussed a few different approaches to the classification problem, however we skipped lightly over the issue of assessing how each classifier performed. We must now discuss this in more detail.

Previously, our approach was as follows: we take a data-set, and use this to estimate the model parameters. Then, for each observation we checked whether it was correctly classified. The performance was then defined as the proportion of items that it correctly classified. We can use this to compare the performance of different classifiers (e.g. discriminant analysis vs KNN) on a particular task and hence choose the best one.

There are (at least) two reasons why things are often more subtle in practice.

Performance Assessment

1. Testing the classifier on the same set of observations used to estimate the parameters leads to overfitting bias, and makes the classifier appear better than it is.
2. Simply counting the number of correct classifications ignores the probabilistic element of prediction.

We will discuss both of these in turn.

1. Cross-Validation

1. Testing the classifier on the same set of observations used to estimate the parameters leads to overfitting bias, and makes the classifier appear better than it is.
2. Simply counting the number of correct classifications ignores the probabilistic element of prediction.

1. Cross-Validation

We discussed this issue in the first lecture — we should not evaluate classifier performance on the same data-set we use to estimate parameters since this uses the data twice and leads to overfitting.

Instead we can divide the data into a training set and a test set. We estimate parameters using the training set, and evaluate the model using the test set.

However this is an inefficient use of data. If we make the test set small, then we will not accurately estimate the error rate. And if we make it too big, we dont have enough observations in the training set to accurately estimate model parameters

1. Cross-Validation

Cross validation is a standard approach to solve this issue. Suppose that rather than dividing the data into training and test sets, we instead randomly divide it into 3 equally sized subsets, so that each observation belongs to only a single subset. Call these A, B and C.
We then do the following, in order

1. Train the model using the combined data in sets A and B, and then compute the error rate on set C. Here, the combined set (A,B) is essentially the training set, and C is the test set.
2. Train the model using the combined data in sets B and C, and then compute the error rate on set A. Here, the combined set (B,C) is essentially the training set, and A is the test set.
3. Train the model using the combined data in sets A and C, and then compute the error rate on set B. Here, the combined set (A,C) is essentially the training set, and B is the test set.

The average of these 3 error rates is then used as the final error rate estimate.

1. Cross-Validation

This is known as 3-fold cross-validation. The clever thing is that we are never training and testing on the same data at the same time, but we do not 'waste' data like we do in the simple training/test set approach.
There is no reason why we have to use only 3 subsets. We could instead divide our data into 5 subsets A,B,C,D,E and do the following:

1. Train the model using the combined data in sets A,B,C,D and then compute the error rate on set E
2. Train the model using the combined data in sets A,B,C,E and then compute the error rate on set D
3. Train the model using the combined data in sets A,B,D,E and then compute the error rate on set C
4. Train the model using the combined data in sets A, C,D,E and then compute the error rate on set B
5. Train the model using the combined data in sets B,C,D,E and then compute the error rate on set A

This is known as 5-fold cross validation

1. Cross-Validation

In general, if we divide the data into K subsets, then it is known as K-fold cross validation.

An extreme case is known as Leave-One-Out-Cross-Validation (LOO-CV). In this case, if we have n observations in our data set, then we take $K = n$. In other words, for every observation (x_i, y_i) in the dataset, we train the model using all the other observations, and then compute the error when classifying this single observation (x_i, y_i)

The most commonly used forms of cross-validation are 5-fold, 10-fold, and LOO-CV

1. Cross-Validation

Some general points:

- A drawback of using cross-validation is that it can be computationally expensive. In K-fold cross-validation we have to train the model K times. This isnt a huge problem for the examples we have seen, since training the model has so far just involved estimating a few parameters using maximum likelihood. But as we look at more complex algorithms and larger data sets, this can start becoming an issue.
- While in theory it might feel like LOO-CV is best since it uses a training set size that is very close to the size of the actual data we have available (and hence it should be an unbiased estimate of the error rate). However, the variance of this estimate can be high, since all the folds are so highly correlated. As such, K-fold validation is often a better choice.

1. Cross-Validation

Cross-validation can be useful for choosing model parameters. Suppose we want to classify objects using the k-Nearest Neighbors algorithm. How do we choose which value of k to use? We saw previously that small values allow a more flexible decision boundary, but can overfit.

Solution: we try various values of the k parameter and choose the one which gives the best error rate.

(I realise that it is confusing that the letter 'k' is being used both for the number of neighbours in the KNN algorithm, and the number of folds in cross-validation. There is nothing I can do about this! For now, I will use lower case k when talking about KNN, and upper-case K when talking about K-fold cross validation)

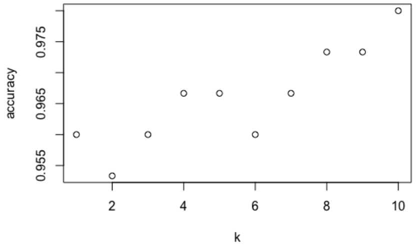
1. Cross-Validation

Lets use 10-fold cross-validation on the Iris data with all 3 classes and 4 features. The createFolds() function in the 'caret' package can be used to generate the split into 10 subsets, or we can do it ourselves using the sample() function. For a given k, we can compute the cross-validated error rate by averaging over all 10 folds:

```
library(caret)
k <- 3 #k value for k-nearest neighbours
numfolds <- 10
folds <- createFolds(iris$Species, k=numfolds)
accuracies <- numeric(numfolds)
for (i in 1:numfolds) {
    traindata <- iris[-folds[[i]],-5]
    trainlabels <- iris[-folds[[i]],5]
    testdata <- iris[folds[[i]],-5]
    testlabels <- iris[folds[[i]],5]
    preds <- knn(traindata,testdata,trainlabels,k=k)
    accuracies[i] <- mean(preds==testlabels)
  }
mean(accuracies)
```

1. Cross-validation

I performed this for each value of $k \in (1, 2, \ldots, 10)$. This shows the results:



The accuracy is maximised for $k = 10$, so this would be a good value to use for this Iris data set (trying higher values didnt improve it further).

Of course, a different value of k might be better for a different classification problem! Given data, we can always use cross-validation to decide which value to use.

2. Scoring Rules

1. Testing the classifier on the same set of observations used to estimate the parameters leads to overfitting bias, and makes the classifier appear better than it is.
2. Simply counting the number of correct classifications ignores the probabilistic element of prediction.

2: Scoring Rules

So far, we have defined the performance of a classifier as the percentage of correct classifications it produces:

$$\frac{1}{n}\sum_{i=1}^{n} I(\hat{y}_i = y_i)$$

But for the discriminant classifier (although not for KNN), we actually get a probabilistic prediction $p(y_i = k|x_i)$.

2: Scoring Rules

Suppose the true class of an observation is $y_i = 0$. If one classifier predicts:

$$p(y_i = 0|x_i) = 0.9$$

and another predicts

$$p(y_i = 0|x_i) = 0.51$$

Then both are correctly classifying the observation, but surely the first is doing better, since it is more confident. At the moment, we are not ‘rewarding’ classifiers for this, i.e. we are throwing information away.

2: Scoring Rules

Definition: Suppose Y is a random variable with a finite set $\mathcal{Y}$ of possible outcomes. Suppose there are K such outcomes. A probability forecast is a K-length vector $p = (p_1, \ldots, p_K)$ which assigns a probabilistic value to each possible outcome. These vectors live on the K-dimensional simplex Δ_K (i.e. they sum to 1). A **scoring rule** is a function $S : \Delta_K \times \mathcal{Y} \to \mathbb{R}$. We write this as $S(p, y)$.

(this is just a formal way of saying that if a classifier produces a probability distribution over the set of classes such as $p(y|x)$, then a scoring rule takes this distribution and the true class labels and reduces it to a single number that assesses the quality of the forecast)

2: Scoring Rules

Suppose we have a classification problem with C classes. For an observation x_i, we forecast $p = (p_{i,1}, \ldots, p_{i,C})$. In discriminant analysis, this would just be $p_{i,j}. = p(y_i = j|x_i)$.

Suppose the true class label $y_i = r$. Then the **logarithmic scoring rule** would assign a score of $-\log p_{i,r}$ for this observation, i.e the logarithm of the probability we assigned to the true class. With n observations, the score is:

$$\sum_{i=1}^{n} -\log p_{i,r} \quad \text{where } y_i = r$$

Low scores mean better performance.

2: Scoring Rules

Another commonly used scoring rule is the Brier score. This is most commonly used for binary classification problems (but it does have a multiclass generalisation). In the binary case where each outcome y_i is either 0 or 1, let p_i denote the probability we assign to $p(y_i = 1|x_i)$. Then the Brier score is:

$$\frac{1}{n}\sum_{i=1}^{n}(p_i - y_i)^2$$

Again, low scores mean better performance.

2: Scoring Rules

Example: consider a binary classification problem where a classifier has been learned and is evaluated on 5 test objects.. The first two columns show the probabilities of the classifier, while the latter column shows the true class label:

$p(y_i = 0 \mid x_i)$	$p(y_i = 1 \mid x_i)$	y_i
0.9	0.1	0
0.6	0.4	1
0.5	0.5	0
0.2	0.8	1
0.9	0.1	1

2: Scoring Rules

$p(y_i = 0 \mid x_i)$	$p(y_i = 1 \mid x_i)$	y_i
0.9	0.1	0
0.6	0.4	1
0.5	0.5	0
0.2	0.8	1
0.9	0.1	1

The logarithmic score is:

$$-\log 0.9 - \log 0.4 - \log 0.5 - \log 0.8 - \log 0.1 = 4.24$$

The Brier score is:

$$\frac{(0.1-0)^2 + (0.4-1)^2 + (0.5-0)^2 + (0.8-1)^2 + (0.1-1)^2}{5} = 0.294$$

2: Scoring Rules

In practice, we use scoring rules in the same way we previously used the classification error. We choose a scoring rule which we think is appropriate (e.g. logarithmic score). We then learn any model parameters by choosing them to minimise this score, usually by using cross-validation.

If we are trying to compare the performance of several algorithms then we can compare them by checking which one gives the lowest score, according to the rule we have chosen, and then use the one which performs best.

Note that standard KNN doesnt output probabilities (although there are extensions which do). However we can still compute its Brier score by treating it as assigning a probability of 1 to the class it chose for an observation, and a probability of 0 to all other classes.