PRACTICAL 7

Aim:

The aim of this practical is to illustrate how kNN classifiers can be fit in R. We will also discuss why (and how) data should be split into training and testing sets before fitting a model.

Before you start

Load the `tidyverse`, `class` and `ISLR` packages. We have not used the `class` package before, so if working on your own computer you will have to install this package first.

Instructions:

This document contains discussions of some basic concepts, and instructions which you should execute in RStudio. **Text in bold** indicate actions that are to be taken within RStudio. `Text highlighted in grey` shows the code you should enter in RStudio.

Due date:

For Tutorial 7, it will be assumed that you have mastered the basic concepts discussed here.

About the data

In this practical, we will use another dataset from the `ISLR` package in R. Recall from Tutorial 6 that this package contains datasets accompanying the book *An Introduction to Statistical Learning with Applications in R*, by Gareth James, Daniela Witten, Trevor Hastie and Rob Tibshirani. We will be working with the `Caravan` dataset, where the aim is to predict whether someone will purchase caravan insurance. There are 5 822 observations in the dataset, and 85 attributes, measuring demographic characteristics of individuals. The target variable is `Purchase`, which indicates whether a given person purchased a caravan insurance policy or not.

Explore the data

1. **Examine the dimensions of the dataset.**
   ```
   dim(Caravan)
   ```

2. **Examine the internal structure of the dataset.**
   ```
   str(Caravan)
   ```

3. You will see from the output in Step 2 above, that the target variable `Purchase` has two levels: "Yes" and "No".
   Find the number of observations for each level of the target variable.
   ```
   summary(Caravan$Purchase)
   ```

   Only about 6% of the individuals in the dataset purchased caravan insurance. This means that the dataset is *imbalanced* with respect to the target variable. This is something which might cause problems with classification models: consider what would happen if, instead of trying to learn from the patterns in the data, we simply make

a prediction of "No" for every observation in this dataset. We would be correct about 94% of the time, but it would not be a very good classifier!

Pre-process the data

4. Recall that for kNN classifiers we typically standardise the attributes in the dataset.
Standardise the attributes and store this in a new data frame called `myattr` which consists only of the attributes, not the target variable.

```
myattr <- scale(Caravan[,-86])
```

5. **Check the variance of the first 2 attributes in the original `Caravans` dataset, and also of the first 2 attributes in the standardised `myattr` dataset.**

```
var(Caravan[,1])
var(Caravan[,2])
var(myattr[,1])
var(myattr[,2])
```

The variance for the standardised attributes should be 1.

6. The target variable should not be standardised.
Create a vector consisting of only the target variable `Purchase`, and call it `mylabs`.

```
mylabs <- Caravan[,86]
```

7. We have previously talked about the importance of choosing a model that will generalise well – in other words, we want a model that will perform well not only on the data that was used in training the model, but also perform well on new, unseen data. This can be accomplished by using a portion of the data to train the model, and another

portion of the data to evaluate the model performance afterward. This enables objective evaluation of a model's performance on "unseen" data – that is, on data not used in constructing the model. This is done to help prevent *overfitting*. We typically talk about "training data" as the portion of the data that was used to build the model and "test data" as the portion of the data not used in training the model but kept to evaluate model performance.

7.1 The first 1000 observations in this dataset will be kept as test data and the rest designated as training data.

Create 4 objects, named `train.attr`, `test.attr`, `train.lab` and `test.lab` respectively. `train.attr` and `test.attr` should contain the attributes for the training and test datasets respectively, while `train.lab` and `test.lab` should contain the target variable values.

```
test <- 1:1000
train.attr <- myattr[-test,]
test.attr <- myattr[test,]
train.lab <- mylabs[-test]
test.lab <- mylabs[test]
```

Note that in this case we just took the first 1000 observations as test data, since the data is not sorted in any particular order. This is actually not the best approach to take, since if there is any inherent order in the dataset, the test and training set will not be representative. It is therefore more typical to take a random sample of a specific proportion of the dataset; say 20% for test data and 80% for training data.

7.2 **Check the dimensions of the 4 objects you created in the previous step.**

```
dim(train.attr)
dim(test.attr)
length(train.lab)
length(test.lab)
```

Fitting kNN models

8. **Set a seed equal to 1.**

```
set.seed(1)
```

This is done before the model building process starts, to ensure reproducibility of results.

9. We will start by fitting a kNN model which only considers the single closest neighbour in each case.

9.1 **First check the help file for the `knn()` function from the `class` package, and check the arguments that are required.**

```
?knn
```

9.2 **Fit a 1NN (one nearest neighbour) classifier. Store this model in an object named `onenn`.**

```
onenn <- knn(train.attr, test.attr, train.lab, k = 1)
```

9.3 **Check how well the model constructed in the previous step performs on the (unseen) test data.** To do this we will use the `table()` function. This function creates a crosstabulation; in the specific case below it will cross tabulate the predicted values from the 1NN model and the true labels from the test set.

```
table(onenn, test.lab)
```

Note that the model gets 88.3% of predictions correct. This might seem like a good result, but remember that this dataset was very imbalanced. A simple majority classifier (i.e. predicting "No" in each case) could get 94 % accuracy. The model performance is therefore actually not very good.

10. We now fit a model which considers the three closest neighbours when a classification is made. Note that the voting scheme used by the `class::knn()` function is majority vote (you can verify this in the help documentation for this function).

10.1 **Fit a 3NN (three nearest neighbour) classifier. Store this model in an object named `threenn`.**

```
threenn <- knn(train.attr, test.attr, train.lab, k = 3)
```

10.2 **Check how well the model constructed in the previous step performs on the (unseen) test data.**

```
table(threenn, test.lab)
```

You should find that the model gets 92.6% of predictions on the test data correct.

11. We now fit a model which considers the five closest neighbours when a classification is made.

11.1 **Fit a 5NN (five nearest neighbour) classifier. Store this model in an object named `fivenn`.**

```
fivenn <- knn(train.attr, test.attr, train.lab, k = 5)
```

11.2 **Check how well the model constructed in the previous step performs on the (unseen) test data.**

```
table(fivenn, test.lab)
```

You should find that the model gets 93.4% of predictions on the test data correct.

Discussion of results

Although the performance of the 5NN model is an improvement over that of the 1NN model, recall that a majority classifier (i.e. predicting "No" in each case) would yield around 94% accuracy. (This is equivalent to fitting a kNN model with $k = n$, where n is the number of observations in the dataset.)

We should however take into account that there is likely to be a cost involved in trying to sell insurance to a given individual, for example the cost incurred by a salesperson who has to travel to a potential client (both actual cost and time cost). If a salesperson therefore just attempts to sell to a random selection of individuals, he / she will be successful in around 6% of cases, which might not be worthwhile given the costs involved. Obviously, a salesperson would prefer to sell insurance only to customers who are likely to buy it! Instead of focusing on the overall error rate, we can rather focus on the proportion of individuals that are correctly predicted to buy insurance. This means that we should consider the success rate if a salesperson visits each of the individuals for which the model predicts that he / she would buy insurance.

The 1NN model predicts 67 + 9 = 76 clients will purchase caravan insurance (refer to the output in Question 9.3); 9 of these are clients who actually purchased insurance, giving a success rate of 11.8% - much better than the 6%!

For the 3NN model, this success rate is 20% and for the 5NN model 26.7%.

Even though this is a difficult (imbalanced) dataset, the kNN models seem to be successful in finding patterns in the data.

Additional note

We evaluated the performance of the different models using the (unseen) test data. Take note of the fact that, if we were to evaluate the performance on the training data instead – that is, the data used to construct the model – the 1NN model could in theory obtain 100% accuracy! This is because each point is predicted to be its own nearest neighbour in such an instance. If you wanted to verify whether this is true, for this practical you will find that the model does not achieve exactly 100% accuracy! This is not an error, but happens because there are duplicates in this dataset… in other words, observations in the training dataset with exactly the same attribute values, but different values of the target variable `Purchase`. (For example, check rows 1023 and 1198 of the original `Caravan` dataset.) When using the `knn()` function, ties are broken at random, which means that one of these instances will randomly be chosen as the one nearest neighbour, and it could therefore happen that the "wrong" one is chosen.