Lab

Aim

To learn about **k-Nearest Neighbour** – a lazy type of predictive algorithm (an instance-based learner).

To investigate the impact of several data transformation/preprocessing functions including:

- **Normalisation** of numeric values
- **Standarisation (centering and scaling)** of numeric values
- **Binarisation** of nominal values – **one-hot encoding**

Note: the actual preprocessing above is covered in detail in module CMM535.

Before you start

Complete any previous lab which you have not finished.

Load libraries:

- caret
- mlbench
- partykit

k-Nearest Neighbour

Caret offers a number of k-nearest neighbour implementations. We will mainly use `"knn"`.

The PimaIndiansDiabetes dataset (from the mlbench library), contains 768 instances, 8 numeric attributes (pregnant, glucose, pressure, triceps, insulin, mass, pedigree and age) and a nominal class (diabetes) with 2 values. To use it

```
data(PimaIndiansDiabetes)
```

Inspect this dataset.

Try the following code which applies kNN to the PimaIndiansDiabetes dataset The code is using a leave one out evaluation method.

```
# leave one out evaluation
ctrl1 <- trainControl(method="LOOCV")
# ensure reproducibility of results by setting the seed to a known value
set.seed(123)
#use knn on the PimaIndiansDiabetes dataset
mod11.knn<- train(diabetes~., data=PimaIndiansDiabetes,
            method="knn", trControl=ctrl1)
```

To check accuracy for values k values

```
print(mod11.knn)
```

or

```
print(mod11.knn$results)
```

To visualise the accuracy for various k values

```
plot(mod11.knn, type="p")
```

It has tried values of k of 5, 7 and 9. The best value is 9 based on accuracy.

A confusion matrix can be obtained as follows (you'll use this code more than once today):

```
# Get results for best k, which are stored in "pred".
results1 <- (mod11.knn$pred[mod11.knn$pred$k == mod11.knn$bestTune$k,])
#Remove unwanted columns (k and instance number)
results1$k <- NULL
results1$rowIndex <- NULL
# Produce the confusion matrix.
table(results1)
```

What if a different value of k is wanted?

If we wanted to test values of k from 1 to 20, we could set the `tuneGrid` as follows:

```
# using same train control as before, i.e. leave one out (defined above)
# # ensure reproducibility of results by setting the seed to a known value
set.seed(123)
mod21.knn<- train(diabetes~., data=PimaIndiansDiabetes,
            method="knn", tuneGrid=expand.grid(k=1:20), trControl=ctrl1)
print(mod21.knn)
plot(mod21.knn, type="p")
```

The best k value is 19 for accuracy. Note: some students got 20 instead with the same seed. This may depend on the version of R (or packages) used.

Exercise 1

Obtain a confusion matrix for the best k value and compare it with the confusion matrix obtained earlier.

Exercise 2

Repeat the experiments above but change the evaluation method. Use 3 repeats of 10-fold cross validation and bootstrap. Has the best k remained the same?

Using only selected values of k

To try only selected values of k use a vector in the `tuneGrid`

For example, if we only want to use k=21 and k=23

```
mod31.knn <- train(diabetes~., data=PimaIndiansDiabetes,
      method="knn", tuneGrid=expand.grid(k=c(21,23)), trControl=ctrl1)
print(mod31.knn)
plot(mod31.knn, type="p")
```

Exercise 3

Apply knn to some of the datasets which you have seen in the class. Try to find examples where a small k works best and examples where a larger k works best.

Try to find an example where kNN works really badly.

Pre-processing numeric values

To check the range of values for numeric attributes, use `summary()`. Try

```
summary(PimaIndiansDiabetes)
```

You will see that the attribute values have quite different ranges, so it is best to normalise or standardise (centre and scale). Although in caret numeric attributes are generally standardised, for kNN normalisation is sometimes considered as it uses the same range [0,1] for all attributes, the same rage used when calculating individual distances.

Normalisation

To preprocess values so that they are normalised to a value in the range `ranges` (a MATRIX of min and max values for each attribute, default [0,1]) set the `preProcess` parameter to `c("range")`. Try

```
set.seed(123)
mod51.knn<- train(diabetes~., data=PimaIndiansDiabetes,
                  method="knn", tuneGrid=expand.grid(k=1:20),
                  preProcess=c("range"), trControl=ctrl1)
print(mod51.knn)
plot(mod51.knn, type="p")
```

Standardisation: centering and scaling data

Normally, numeric attributes are centered (mean is 0) and scaled (standard deviation of 1). To do this use `preProcess()` set to `c("center", "scale")`. Try

```
set.seed(1)
mod41.knn<- train(diabetes~., data=PimaIndiansDiabetes,
            method="knn", preProcess=c("center", "scale"),
            trControl=ctrl1)
print(mod41.knn)
plot(mod41.knn, type="p")
```

Compare the best k with the one you obtained when normalising data.

Checking the pre-processed datasets

In the exercises above the preprocessing is done when the model is trained, so there is no option to see how the dataset looks like.

Checking normalised data

To see the normalised dataset try

```
preProcValuesN <- preProcess(PimaIndiansDiabetes, method = c("range"))
diabetesNormalised <- predict(preProcValuesN, PimaIndiansDiabetes)
# checking the results. Use head for knitting so that the results appear
# in the knitted document. View shows them in a separate window.
# View(diabetesNormalised)
head(diabetesNormalised, 15)
```

To check that all the values are on the scale [0, 1] use the summary function.

```
summary(diabetesNormalised)
```

Checking standardised data - centered and scaled data

To check the centered and scaled dataset use

```
preProcValuesCS <- preProcess(PimaIndiansDiabetes,
      method =c("center", "scale"))
diabetesCenteredScaled <- predict(preProcValuesCS, PimaIndiansDiabetes)
# View(diabetesCenteredScaled)
head(diabetesCenteredScaled, 15)
# Checking summary of each attribute.
summary(diabetesCenteredScaled)
```

Exercises

4. Normalise the WeatherPlay dataset (package partykit) and check the resulting instances are similar to those in obtained during class exercises.
5. Standardise (centre and scale) the WeatherPlay dataset (package partykit) and check the results.

From nominal to binary – one-hot encoding

We have already seen that in Caret attributes are numeric. Nominal attributes may be ignored in the computation of distances, depending on actual package used. Nominal attributes with p different values may be converted to p binary attributes (one-hot encoded).

```
#binarise nominal attributes – one-hot encoding

# make a copy
```

```
noClass <-WeatherPlay

# remove the class - it is not transformed
noClass$play <- NULL

set.seed(123)
#binarise nominal attributes - one-hot encoding
binaryVars <- dummyVars(~ ., data = noClass)
newWeather <- predict(binaryVars, newdata = noClass)

# add the class to the binarised dataset
binWeather <-cbind(newWeather, WeatherPlay[5])

# check the result.
# View(binWeather) could have been used instead but it does not appear in
the knitted document.

head(binWeather, 14)
```

You can see that each value for outlook is represented by a binary variable. The same applies to each value of attribute windy. Dataset binWeather can now be used (after normalising or centering and scaling).

Exercises

6. Transform nominal attributes into binary ones (i.e. one-hot encode) for the WeatherPlay dataset and then normalise all numeric attributes. Apply kNN and compare the results to the ones you got without binarisation.
7. Transform nominal attributes into binary ones for the WeatherPlay dataset and then centre and scale all numeric attributes. Apply kNN and compare the results to the ones you got without binarisation.
8. Compare the performance of kNN with the 2 datasets obtained in exercises 3 and 4.

kNN with weights (not in Practical Examination)

There are some kNN algorithms which implement weighted kNN. For example

`Kknn` and `ownn.`

kknn (in package kknn) has several parameters including

- `kmax` – the maximum k value.
- `d` – a parameter for the distance function. Use 1 for Manhattan distance and 2 for Euclidean distance.
- `kernel` – this is where the weights are implemented. There are various options which are outside the scope of this module. The choices include "rectangular" (this is the unweighted version), "triangular", "epanechnikov", "biweight", "triweight", "cos", "inv", "gaussian", "rank" and "optimal".

These can be used with the tuneGrid parameter in the train function (see example below).

Below is an example which uses 2 different methods for weights and 3 different values for k. **Note that this takes a really long time to run!**

```
tg <- expand.grid(kmax = 5:7,
                  distance = 2,
                  kernel = c('triangular', 'optimal'))
set.seed(123)
mod107.knn<- train(diabetes~., data=PimaIndiansDiabetes,
           method="kknn", preProcess=c("range"),
           trControl=ctrl1, tuneGrid =tg)
```

ownn (in package snn) implements optimal weights and only has one parameter `K` (note that this is uppercase!).

For example,

```
set.seed(123)
mod108.knn<- train(diabetes~., data=PimaIndiansDiabetes,
           method="ownn", preProcess=c("range"),
           trControl=ctrl1, tuneGrid =expand.grid(K=5:7))
```

Exercise 9

Use weighted kNN in a dataset of your choice.