An introduction to Clustering and Classification using R and the caret package

-R group QIMR Berghofer, Brisbane

Jack Galbraith, Translational Research Institute, UQ Diamantina,
Faculty of Medicine, University of Queensland

We will be using the iris dataset in later parts so we will import the dataset now.

```
library(knitr)
library(scatterplot3d)
library(MASS)
library(class)
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

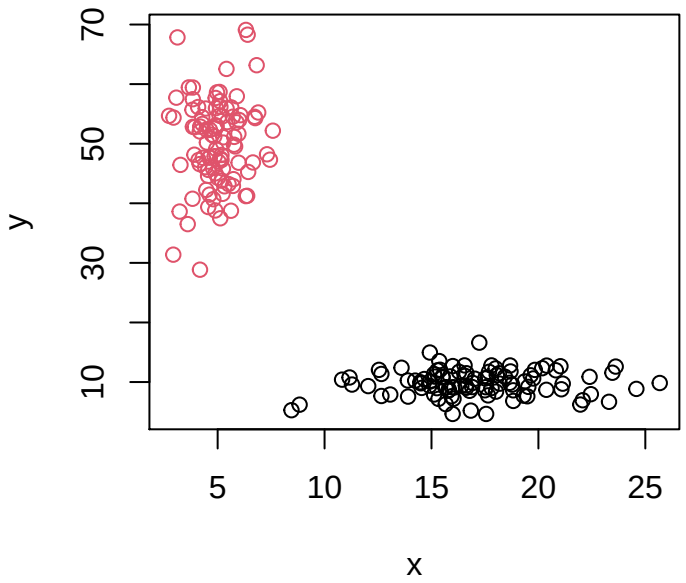
```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

K-means clustering

For a clear example of kmeans we'll create a dataset with two normally distributed random clusters across two variables x and y. The kmeans algorithm can either use random start locations across the range of the data set or choose datapoints at random as initial locations. R's base stats package includes kmeans() which makes use of the latter start locations. Below we have plotted the random dataset coloring points based on their clusters from kmeans(xydata, k=2).

```
set.seed(100)
x<-c(rnorm(100, mean = 5, sd=1), rnorm(100, mean = 17, sd = 4))
y<-c(rnorm(100, mean = 50, sd = 7), rnorm(100, mean = 10, sd = 2))
xydata<-data.frame(x, y, stringsAsFactors = F)
ktest<-kmeans(xydata, 2, nstart=5)
plot(x,y, col=ktest$cluster)
```

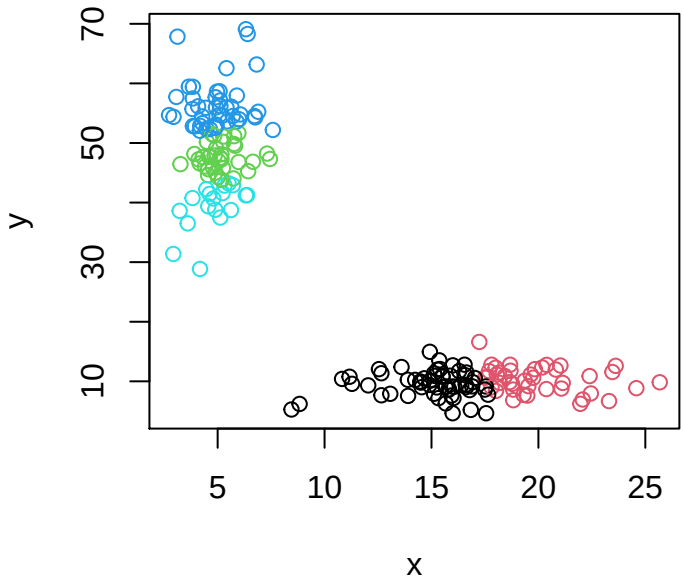


Clusters are user defined, as such we could try more than 2 clusters on this data set. Since the starts are random it is possible to get stuck in a local optimum instead of global, by assigning nstart values in kmeans() we can increase the number of random starts that are tried before the algorithm chooses the best start. For each start all datapoints are allocated to their closest cluster (starting point), and the means of all distances to the starting points for each cluster is calculated. These means then form the start locations for the next iteration and so on until no more changes occur each iteration or the iter.max is reached. The below plot uses 5 clusters with iter.max=10 and nstart=5, therefore the algorithm will:

1. assign clusters membership based on initial start locations
2. calculate mean of clusters
3. repeat using the mean of clusters as new start locations up to 10 times

4. repeat all of the above with new start locations for a total of 5 different initial start locations
5. return clustering based on best result of 5 different starts (hopefully a global optimum)

```
ktest<-kmeans(xydata, 5, iter.max=10, nstart=5)
plot(x,y, col=ktest$cluster)
```

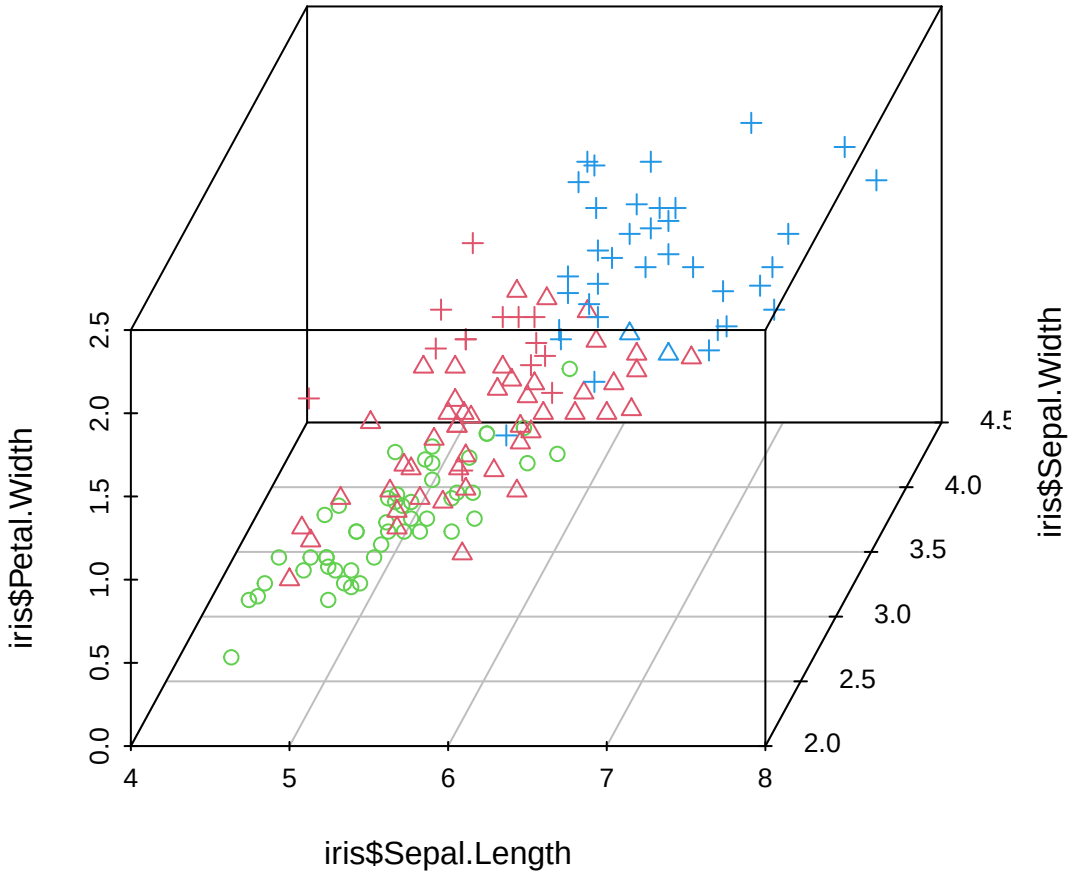


In fact we can keep increasing the number of clusters until k=n where n is the number of samples in our dataset and we will continue to see improvements to the 'within sum of squared error'. Generally speaking it is better to keep the number of clusters as low as possible while still achieving decent 'within sum of squared error', this helps avoid finding arbitrarily large numbers of clusters in a dataset (akin to overfitting).

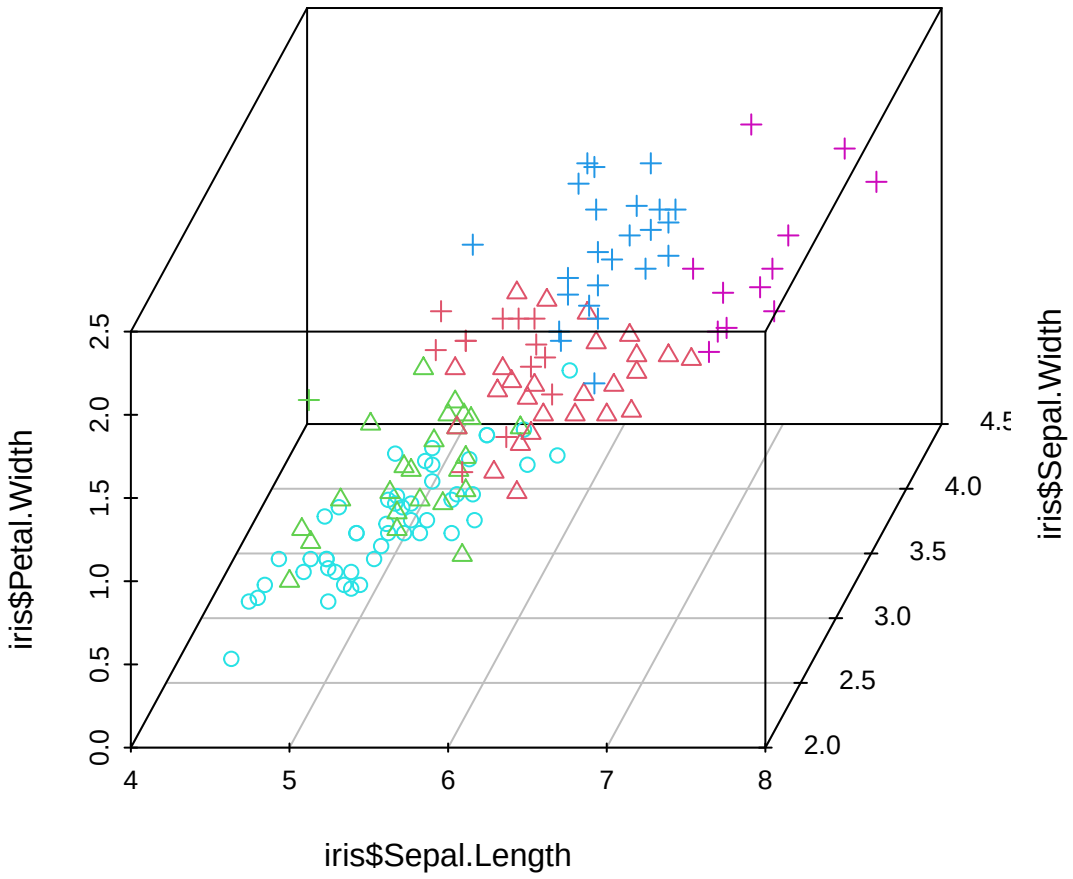
```
## $cluster
##   [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [38] 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [75] 1 1 1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 3 3 3 3 1 3 3 3 3
## [112] 3 3 1 1 3 3 3 3 1 3 1 3 1 3 3 1 1 3 3 3 3 3 1 3 3 3 3 1 3 3 3 1 3 3 3 1 3
## [149] 3 1
##
## $centers
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     5.901613    2.748387     4.393548    1.433871
## 2     5.006000    3.428000     1.462000    0.246000
## 3     6.850000    3.073684     5.742105    2.071053
##
## $totss
## [1] 681.3706
##
```

```
## $withinss
## [1] 39.82097 15.15100 23.87947
##
## $tot.withinss
## [1] 78.85144
##
## $betweenss
## [1] 602.5192
```

Now we will apply kmeans to a real dataset,iris, using k=3. We can see that the stats function kmeans() returns an object of class 'kmeans' with the above listed attributes. Coloring the datapoints based on their cluster membership and shape based on flower species, we can see that with three clusters we can clearly separate the circles (setosa) from the other species but not separate virginica and versicolor. This could be of use in a pipeline using both clustering and classification.

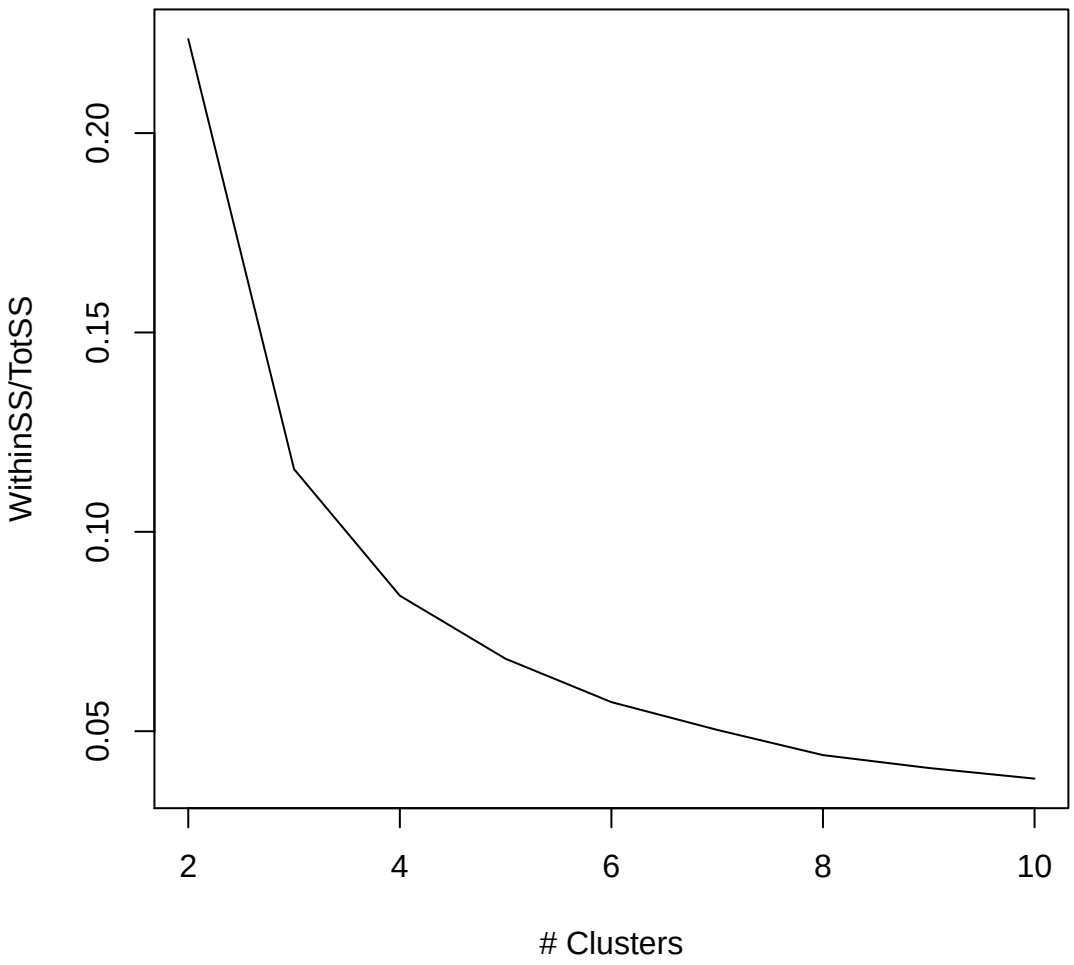


Trying 5 clusters yields improved results with only 1 of the clusters showing much mixing of versicol and verginca. However, the question of how many clusters to use still remains.



How many clusters should you use?

By plotting the 'within cluster sum of squared error' divided by the 'total sum of squared error' we can get an idea of how quickly improvements from additional clusters reduces. The below plot is known as a scree plot and can give an indication as to a good number of clusters to use.



From this example, it might be a good idea to try k=3,4,5 for any subsequent model if including kmeans as a first step. Other useful methods for choosing the number of clusters to use include the silhouette method and the gap-statistic (cluster package).

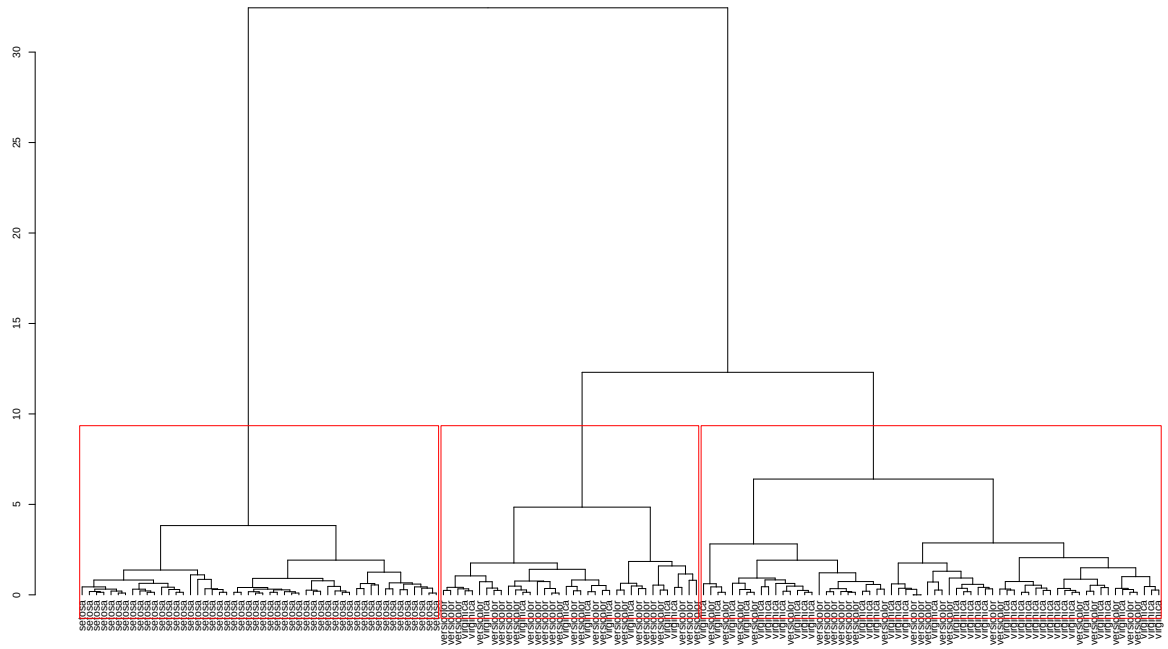
Hiearchical Clustering

The R base stats package contains the hclust() and dist() functions which will both be used for hierarchical clustering. dist() defaults to euclidean distance but can take other arguments, or vegdist() from the "vegan" package can be used for alternatives such as Bray-Curtis.

1. The dist() function allows us to create a distance matrix - that is a matrix containing all distances between all possible pairs of data points. It is these distances that then are used for calculating which points are closest to one another during the hclust() clustering algorithm.
2. The hclust() function uses agglomerative clustering, all samples start separated from each other and are then joined together in successive rounds based on the pair-wise distances. ward.D2 method calculates the minimum variance for each successive merger but other common methods are included such as complete or average.

Below we have cluster based on the first 4 variables of the iris dataset and set the labels to the species of flower.

```
D.<-dist(iris[,-5])
H.<-hclust(D., method = "ward.D2")
H.$labels<-iris[H.$order,5]
plot(as.dendrogram(H.))
rect.hclust(H., 3, border = "red")
```




```
HClusts<-cutree(H., k=3)
#identify(H., function(x) print(x))
```

The identify() function is an interesting interactive way of selecting clusters. While rect.hclust() or cutree() draw around/return the user-specified number of clusters, identify() allows you to manually click on the

plotted graph to return specific clusters as well as specify a function to affect the returned data-point indices (using print(x) as above which would simply print the indices of data-points from the iris dataset to the console when clicking on the dendrogram). This may be useful for very large datasets where only 1 or 2 clusters show interesting behavior.

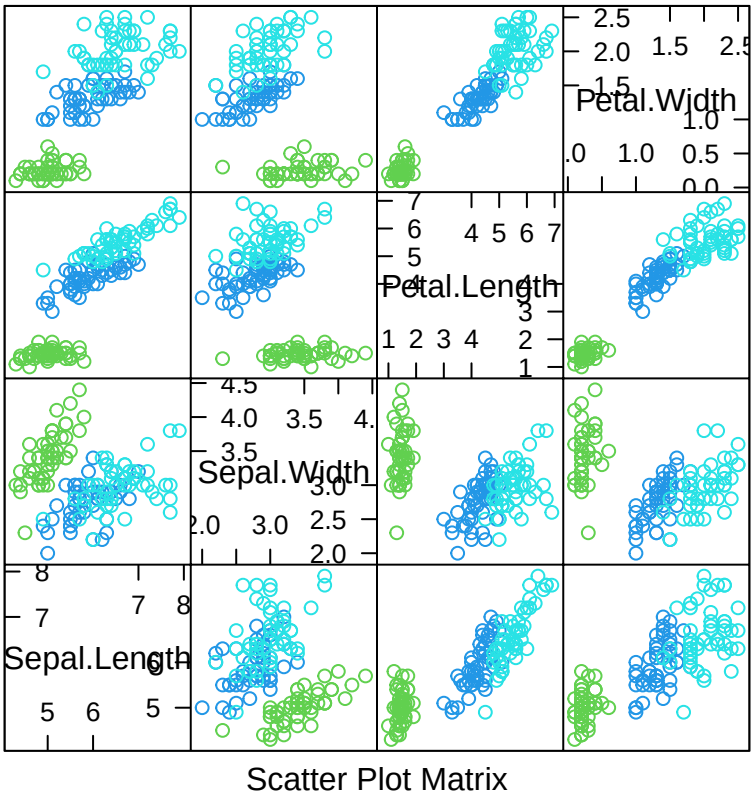
Caret Package

The Caret package acts as a wrapper for most other modelling functions and has a very informative vignette https://topepo.github.io/caret/. We will do a quick walkthrough to see the basics. The three main aspects we'll cover are:

- Data partitioning
- Data pre-processing
- Training a model

But first we can use the caret package to get a quick overview of the data via the featurePlot() function which acts as a wrapper for trellis plots. This can be useful with small numbers of variables but cumbersome with larger numbers. In this case we used plot="pairs" which gives a scatterplot.

```
featurePlot(iris[,-5], iris$Species, plot="pairs", col=c(3,4,5))
```



From a cursory glance we can see that Petal.Width and Petal.Length appear better discriminants of flower species than Sepal.Width or Sepal.Length ie. better separation of colours/species.

Data-partitioning

Splitting data into training and testing portions is at the core of machine learning as it allows estimates of variance for the model's predictions and makes comparing the predictive capacity of different models much easier. For example, if we used the whole dataset to train the model then we would need to go gather more data before we could test the model's predictions on new datapoints, only to find that the model may not be that good anyway. Below are three of the most common methods of datapartioning, holdout, k-fold cross validation, and bootstrapping.

1. **createDataPartition -** Holdout is simply witholding a portion of the data for a test set (i.e. 80% for training, 20% for testing).

```
irisPart.<-createDataPartition(iris$Species, p=0.8, list=F)
head(irisPart.)
```

```
##      Resample1
## [1,]         1
## [2,]         3
## [3,]         4
## [4,]         5
## [5,]         7
## [6,]         9
```

2. **createFolds -** K-fold cross validation splits the data into k number of equally sized partions and then trains with all but 1 of these partions using the left-out partition for testing. It then repeats the process of training and testing with different combinations of folds untill each partition has been tested once (i.e. for k=4 folds we would use 3 folds for training and 1 for testing repeating 3 more times until each fold had been tested once).

```
irisPart.<-createFolds(iris$Species, k=5)
kable(head(data.frame(irisPart.)),caption = "Example of k-fold Cross-validation")
```

Table 1: Example of k-fold Cross-validation

Fold1	Fold2	Fold3	Fold4	Fold5
4	11	2	1	6
8	12	3	5	9
13	14	7	19	10
18	17	23	20	15
28	21	31	26	16
30	22	34	27	25

3. **createResample -** Bootstrapping is random sampling but with replacement, meaning the same datapoint could be sampled multiple times while others are missed completely. In the case of the iris dataset (with 150 total datapoints) a bootstrap sample would be a n=150 sample taken from the iris dataset with replacement.

```
irisPart.<-createResample(iris$Species, times=10)
names(irisPart.)<-paste("Btsmp", 1:10)
kable(head(data.frame(irisPart.)), caption = "Example of Bootstrapping Sampling")
```

Table 2: Example of Bootstrapping Sampling

Btsmp.1	Btsmp.2	Btsmp.3	Btsmp.4	Btsmp.5	Btsmp.6	Btsmp.7	Btsmp.8	Btsmp.9	Btsmp.10
2	4	3	2	2	2	2	2	1	2
3	4	5	3	3	3	2	3	2	2
4	4	6	3	3	3	2	3	2	2
4	4	8	4	6	3	6	6	3	4
4	5	8	5	6	3	8	8	4	5
4	6	8	6	7	6	10	8	4	6

Training a Model

The train() function is the workhorse of the caret package that wraps other modelling functions (selected by the method= argument). By setting the arguments of trainControl() and saving as a variable, we can define our datapartitioning method within the modelling function train() itself using the trControl argument. Often this is easier than using the above partitioning methods separately from the model, though these methods may still be useful for creating your own models. The below example uses a k-nearest-neighbours classification model with 10 bootstrapped samples.

```
#control data partitions within future models
fitControl.boot<-trainControl(method="boot", number=10)

#Train a KNN classifier using the above bootstrapping controls
KNN.caret.boot<-train(Species~., data=iris, trControl=fitControl.boot, method="knn")
KNN.caret.boot
```

```
## k-Nearest Neighbors
##
## 150 samples
##   4 predictor
##   3 classes: 'setosa', 'versicolor', 'virginica'
##
## No pre-processing
## Resampling: Bootstrapped (10 reps)
## Summary of sample sizes: 150, 150, 150, 150, 150, 150, ...
## Resampling results across tuning parameters:
##
##   k  Accuracy   Kappa
##   5  0.9558162  0.9331921
##   7  0.9577030  0.9360492
##   9  0.9541347  0.9306772
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 7.
```

```
table(predict(KNN.caret.boot, iris), iris$Species, dnn = c("Predicted", "Actual"))
```

```
##             Actual
## Predicted    setosa versicolor virginica
##   setosa         50          0         0
##   versicolor      0         47         1
##   virginica       0          3        49
```

We can see that a simple summary of the model is provided with accuracy and model parameters, in this case k. Using the table() function we can build a simple confusion matrix, although the Caret package also has methods for this as well.

One of the most useful aspects of the Caret package is the ability to use completely different predictive models with just a minor tweak to the syntax. Below we have trained a random forest model using the same bootstrapped resamples as before for, seen in the knn example. The only difference here is the change of method from "knn" to "rf" for random forest.

NB: Since Caret acts as a wrapper to functions from other packages, you may have to install the underlying packages using install.packages() before you can use a particular model.

```
#Train a random forest using Cross Validation controls
RF.caret.boot<-train(Species~., data=iris, trControl=fitControl.boot, method="rf")
RF.caret.boot
```

```
## Random Forest
##
## 150 samples
##   4 predictor
##   3 classes: 'setosa', 'versicolor', 'virginica'
##
## No pre-processing
## Resampling: Bootstrapped (10 reps)
## Summary of sample sizes: 150, 150, 150, 150, 150, 150, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##   2     0.9473544  0.9204651
##   3     0.9509011  0.9258425
##   4     0.9489011  0.9227694
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 3.
```

```
table(predict(RF.caret.boot, iris), iris$Species, dnn = c("Predicted", "Actual"))
```

```
##             Actual
## Predicted    setosa versicolor virginica
##   setosa         50          0         0
##   versicolor      0         50         0
##   virginica       0          0        50
```

Pre-processing

By using the preProcess function we can remove highly correlated variables or reduce large multidimensional datasets to just the first couple of principle components. preProcess() takes a dataset and method, such as iris and pca, and then returns an object that can be used with the predict function.

```
#Preprocess predictors to just PC1 and PC2 using PCA
#then run a KNN classifier on the principle components

fitControl.CV<-trainControl(method="LOOCV", p=0.9)
```

```
pcaIRIS<-predict(preProcess(iris, method="pca"), iris)
kable(head(pcaIRIS), caption="Example of PCA preprocessing")
```

Table 3: Example of PCA preprocessing

Species	PC1	PC2
setosa	-2.257141	-0.4784238
setosa	-2.074013	0.6718827
setosa	-2.356335	0.3407664
setosa	-2.291707	0.5953999
setosa	-2.381863	-0.6446757
setosa	-2.068701	-1.4842053

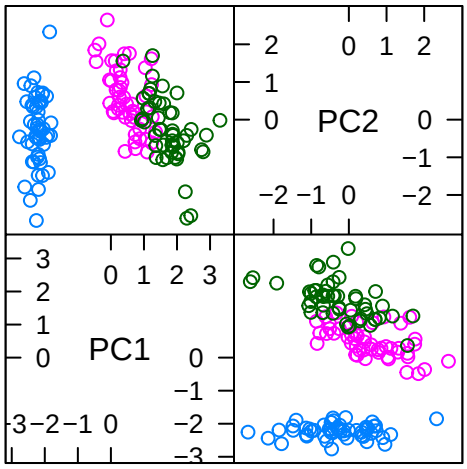
```
KNN.caret.PCAboot<-train(Species~., data=pcaIRIS, trControl=fitControl.boot, method="knn")
KNN.caret.PCAboot
```

```
## k-Nearest Neighbors
##
## 150 samples
##   2 predictor
##   3 classes: 'setosa', 'versicolor', 'virginica'
##
## No pre-processing
## Resampling: Bootstrapped (10 reps)
## Summary of sample sizes: 150, 150, 150, 150, 150, 150, ...
## Resampling results across tuning parameters:
##
##   k  Accuracy   Kappa
##   5  0.8987256  0.8467159
##   7  0.9073444  0.8596615
##   9  0.9131606  0.8683503
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 9.
```

We can see that the accuracy has reduced slightly when using the pca preprocessed data since we have only trained on 2 components instead of 4. With very large datasets this can help reduce the effects of overfitting, though in such cases using models that include regularisation methods, such as lasso, would also assist.

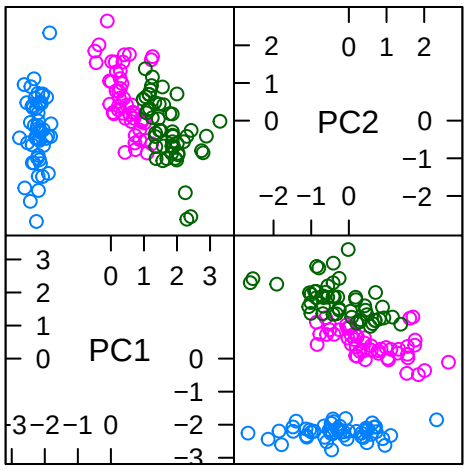
Lastly featurePlot() can be used to compare model predictions and identify issues with classifiers. By changing the plot argument we can create scatterplots or density plots etc.

```
featurePlot(pcaIRIS[,-1], y=pcaIRIS$Species, plot="pairs")
```



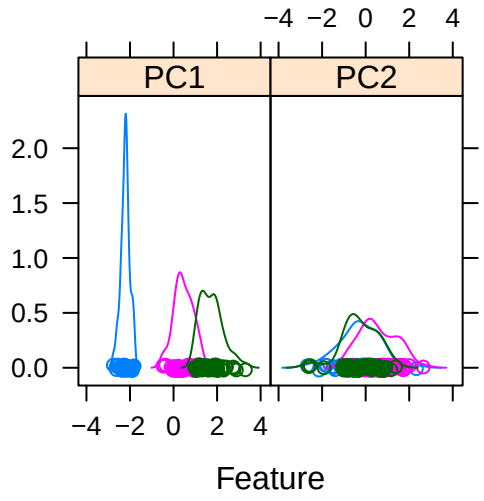
Scatter Plot Matrix

```
featurePlot(pcaIRIS[,-1], y=predict(KNN.caret.PCAboot, pcaIRIS), plot="pairs")
```



Scatter Plot Matrix

```
featurePlot(pcaIRIS[,-1], y=predict(KNN.caret.PCAboot, pcaIRIS), plot="density")
```



Density plots can be particularly useful. We can see from the above density plot, using the pca preprocessed data, that only pc1 allows for decent separation of the flower species, and therefore we would probably be able to create a model with similar accuracy using just the PC1 values as a predictor variable.

References/Resources

1. Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2014. An Introduction to Statistical Learning: with Applications in R. Springer Publishing Company, Incorporated.

2. Caret vignette https://topepo.github.io/caret/

3. Use help() to bring up R documentations for any functions