Introduction to Machine Learning in R

Sebastian Palmas, Kevin Oluoch

2019/11/07

Introduction

This hands-on workshop is meant to introduce you to the basics of machine learning in R: more specifically, it will show you how to use R to work well-known machine learning algorithms, including unsupervised (k-means clustering) and supervised methods (such as k-nearest neighbours, SVM, random forest).

This introductory workshop on machine learning with R is aimed at participants who are not experts in machine learning (introductory material will be presented as part of the course), but have some familiarity with scripting in general and R in particular.

We will be using sample datasets available in R and from free online sources, just be sure that your internet is working to download some of the data.

Objectives

The course aims at providing an accessible introduction to various machine learning methods and applications in R. The core of the courses focuses on unsupervised and supervised methods.

The course contains exercises to provide opportunities to apply learned code.

At the end of the course, the participants are anticipated to be able to apply what they have learnt, as well as feel confident enough to explore and apply new methods.

The material has an important hands-on component and participants

Pre-requisites

- Participants are expected to be familiar with the R syntax and basic plotting functionality.
- R 3.5.1 or higher.
- The wine dataset needs to be downloaded from an online repository.

Overview of Machine Learning

Machine learning is a method of data analysis that automates analytical model building. It is a branch of artificial intelligence based on the idea that systems can learn from data, identify patterns and make decisions with minimal human intervention.

Machine learning algorithms are often categorized as supervised or unsupervised. In supervised learning, the learning algorithm is presented with labelled example inputs, where the labels indicate the desired output. Supervised algorithms are composed of classification, where the output is categorical, and regression, where the output is numerical. In unsupervised learning, no labels are provided, and the learning algorithm focuses solely on detecting structure in unlabelled input data.

Note that there are also semi-supervised learning approaches that use labelled data to inform unsupervised learning on the unlabelled data to identify and annotate new classes in the dataset (also called novelty detection).

Packages

R has multiple packages for machine learning. These are some of the most popular:

- `caret`: Classification And REgression Training
- `randomForest`: specific for random forest algorithm
- `nnet`: specific for neural networks
- `Rpart`: Recursive Partitioning and Regression Trees
- `e1071`: SVM training and testing models
- `gbm`: Generalized boosting models
- `kernlab`: also for SVM

I will use the `caret` package in R. `caret` can do implementation of validation, data partitioning, performance assessment, and prediction. However, `caret` is mostly using other R packages that have more information about the specific functions underlying the process, and those should be investigated for additional information. Check out the caret home page for more detail. We will also use `randomForest` for the Random Forest algorithm and `caretEnsemble` for an example of an ensemble method.

In addition to `caret`, it's a good idea to use your computer's resources as much as possible, or some of these procedures may take a notably long time, and more so with the more data you have. `caret` will do this behind the scenes, but you first need to set things up. Say, for example, you have an quad core processor, meaning your processor has four cores essentially acting as independent CPUs. This is done by allowing parallel processing using the `doSNOW` package.

The other packages that we will use are:

- `tidyverse`: for data manipulation

- `corrplot`: for a correlation plot

If you don't have them installed, please do:

```
install.packages("caret")
install.packages("caretEnsemble")
install.packages("tidyverse")
install.packages("corrplot")
install.packages("doSNOW")
install.packages("randomForest")
```

To start, let's load some packages:

```
library(caret)
library(corrplot)
library(tidyverse)
```

Data Set – Wine

We will use the wine data set from the UCI Machine Learning data repository. These are results of a chemical analysis of wines grown in the same region in Italy. The analysis determined the quantities of 13 constituents found in each of the three types of wines.

The goal is to predict wine quality, of which there are 7 values (integers 3-9). We will turn this into a binary classification task to predict whether a wine is 'good' or not, which is arbitrarily chosen as 6 or higher. After getting the hang of things one might redo the analysis as a multiclass problem or even toy with regression approaches, just note there are very few 3s or 9s so you really only have 5 values to work with. The original data along with detailed description can be found here, but aside from quality it contains predictors such as residual sugar, alcohol content, acidity and other characteristics of the wine.

The original data is separated into white and red data sets. I have combined them and created additional variables: color and good, indicating scores greater than or equal to 6 (denoted as 'Good' or 'Bad').

```
wine_red <- read.csv("https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality
                     sep=";")
wine_white <- read.csv("https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequali
                       sep=";")

wine_red <- wine_red %>%
  mutate(color="red")

wine_white <- wine_white %>%
  mutate(color="white")
```

```r
wine <- wine_red %>%
  rbind(wine_white) %>%
  mutate(white=1*(color=="white"),
         good=ifelse(quality>=6,"Good", "Bad") %>% as.factor())

write.csv(wine, file = "wine.csv")
```

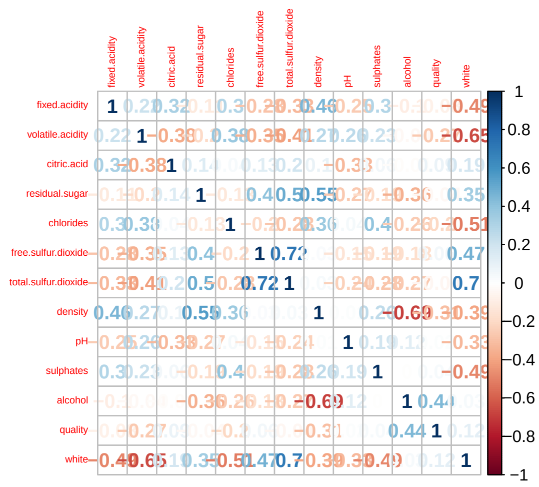
The following will show some basic numeric information about the data

```r
summary(wine)
```

```
##  fixed.acidity    volatile.acidity
##  Min.   : 3.800   Min.   :0.0800
##  1st Qu.: 6.400   1st Qu.:0.2300
##  Median : 7.000   Median :0.2900
##  Mean   : 7.215   Mean   :0.3397
##  3rd Qu.: 7.700   3rd Qu.:0.4000
##  Max.   :15.900   Max.   :1.5800
##   citric.acid     residual.sugar
##  Min.   :0.0000   Min.   : 0.600
##  1st Qu.:0.2500   1st Qu.: 1.800
##  Median :0.3100   Median : 3.000
##  Mean   :0.3186   Mean   : 5.443
##  3rd Qu.:0.3900   3rd Qu.: 8.100
##  Max.   :1.6600   Max.   :65.800
##    chlorides        free.sulfur.dioxide
##  Min.   :0.00900   Min.   :  1.00
##  1st Qu.:0.03800   1st Qu.: 17.00
##  Median :0.04700   Median : 29.00
##  Mean   :0.05603   Mean   : 30.53
##  3rd Qu.:0.06500   3rd Qu.: 41.00
##  Max.   :0.61100   Max.   :289.00
##  total.sulfur.dioxide    density
##  Min.   :  6.0        Min.   :0.9871
##  1st Qu.: 77.0        1st Qu.:0.9923
##  Median :118.0        Median :0.9949
##  Mean   :115.7        Mean   :0.9947
##  3rd Qu.:156.0        3rd Qu.:0.9970
##  Max.   :440.0        Max.   :1.0390
##        pH           sulphates
##  Min.   :2.720   Min.   :0.2200
##  1st Qu.:3.110   1st Qu.:0.4300
##  Median :3.210   Median :0.5100
##  Mean   :3.219   Mean   :0.5313
##  3rd Qu.:3.320   3rd Qu.:0.6000
##  Max.   :4.010   Max.   :2.0000
##     alcohol         quality
##  Min.   : 8.00   Min.   :3.000
##  1st Qu.: 9.50   1st Qu.:5.000
##  Median :10.30   Median :6.000
##  Mean   :10.49   Mean   :5.818
##  3rd Qu.:11.30   3rd Qu.:6.000
##  Max.   :14.90   Max.   :9.000
##     color               white
##  Length:6497        Min.   :0.0000
##  Class :character   1st Qu.:1.0000
##  Mode  :character   Median :1.0000
##                     Mean   :0.7539
##                     3rd Qu.:1.0000
##                     Max.   :1.0000
##    good
##  Bad :2384
##  Good:4113
##
##
##
##
```

We can visualize the correlations between all variables in the dataset with the `corrplot::corrplot` function.

```r
corrplot(cor(wine[, -c(13, 15)]),
         method = "number",
         tl.cex = 0.5)
```



Data partition

The function `createDataPartition` from the `caret` package will produce indices to use as the training set. In addition to this, we will normalize the continuous variables to the [0,1] range. For the training data set, this will be done as part of the training process, so that any subsets under consideration are scaled separately, but for the test set we will go ahead and do it now

```r
set.seed(1234) #so that the indices will be the same when re-run
trainIndices <- createDataPartition(wine$good,
                                    p = 0.8,
                                    list = F)
wine_train <- wine[trainIndices, -c(6, 8, 12:14)] #remove quality and color, as well as density and oth
wine_test <- wine[!1:nrow(wine) %in% trainIndices, -c(6, 8, 12:14)]
```

Random forest

Random Forests is a learning method for classification and regression. It is based on generating a large number of decision trees, each constructed using a different subset of your training set. These subsets are usually selected by sampling at random and with replacement from the original data set. In the case of classification, the decision trees are then used to identify a classification consensus by selecting the most common output. In the event, it is used for regression and it is presented with a new sample, the final prediction is made by taking the average of the predictions made by each individual decision tree in the forest.

The portion of samples that were left out during the construction of each decision tree in the forest are referred to as the Out-Of-Bag (OOB) dataset. As we'll see later, the model will automatically evaluate its own performance by running each of the samples in the OOB dataset through the forest.

Implementation

The R package `randomForest` is used to create random forests.

```r
library(randomForest)
```

```
## Warning: package 'randomForest' was built
## under R version 3.5.3
```

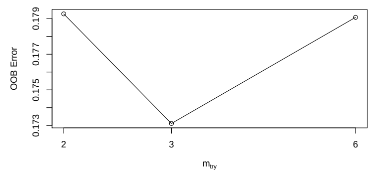
Tune The Forest

By "tune the forest" we mean the process of determining the optimal number of variables to consider at each split in a decision-tree. Too many prediction variables and the algorithm will over-fit; too few prediction variables and the algorithm will under-fit. so first, we use `tuneRF` function to get the possible optimal numbers of prediction variables. The `tuneRF` function takes two arguments: the prediction variables and the response variable.

This function also returns a plot on how the error varies depending on the number of prediction varialbles.

```
trf <- tuneRF(x=wine_train[,1:9], # Prediction variables
              y=wine_train$good) # Response variable
```

```
## mtry = 3  OOB error = 17.31%
## Searching left ...
## mtry = 2     OOB error = 17.93%
## -0.03555556 0.05
## Searching right ...
## mtry = 6     OOB error = 17.91%
## -0.03444444 0.05
```



`tuneRF` returns the several numbers of variables randomly sampled as candidates at each split (mtry). error.](https://en.wikipedia.org/wiki/Out-of-bag_error) prediction error.

To build the model, we pick the number with the lowest [Out-of-Bag (OOB)

```
(mintree <- trf[which.min(trf[,2]),1])
```

```
## [1] 3
```

Fit The Model

We create a model with the `randomForest` function which takes as arguments: the response variable the prediction variables and the optimal number of variables to consider at each split (estimated above). We also get the function to rank the prediction variables based on how much influence they have in the decision-trees' results.

```
rf_model <- randomForest(x=wine_train[,-10], # Prediction variables
                         y=wine_train$good, # Response variable
                         mtry=mintree, # Number of variables in subset at each split
                         importance = TRUE # Assess importance of predictors.
)
rf_model
```

```
##
## Call:
##  randomForest(x = wine_train[, -10], y = wine_train$good, mtry = mintree,      importance = TRUE)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##         OOB estimate of  error rate: 16.62%
## Confusion matrix:
##       Bad Good class.error
## Bad  1389  519   0.2720126
## Good  345 2946   0.1048314
```

We can have a look at the model in detail by plotting it to see a plot of the number of trees against OOB error: the error rate as the number of trees increase.

```
plot(rf_model, main="")
```

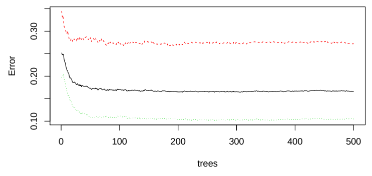
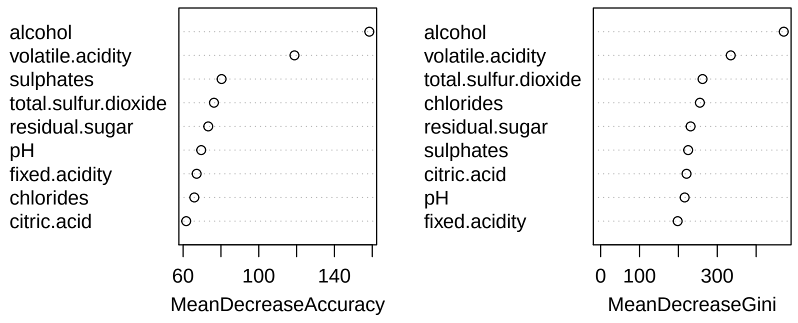


Figure 1: Error rates on random forest model

We can have a look at each variable's influence by plotting their importance based on different indices given by the importance function.

```
varImpPlot(rf_model, main="")
```



## Validation

We can check the model fitness against the test dataset

```
preds_rf <- predict(rf_model, wine_test[,-10])
confusionMatrix(preds_rf, wine_test[,10], positive='Good')
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Bad Good
##       Bad  337   98
##       Good 139  724
##
##                Accuracy : 0.8174
##                  95% CI : (0.7953, 0.8381)
##     No Information Rate : 0.6333
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5996
##
##  Mcnemar's Test P-Value : 0.009369
##
##             Sensitivity : 0.8808
##             Specificity : 0.7080
##          Pos Pred Value : 0.8389
##          Neg Pred Value : 0.7747
##              Prevalence : 0.6333
##          Detection Rate : 0.5578
##    Detection Prevalence : 0.6649
##       Balanced Accuracy : 0.7944
##
##        'Positive' Class : Good
##
```

More information on Random Forests

- https://uc-r.github.io/random_forests
- https://towardsdatascience.com/random-forest-in-r-f66adf80ec9

k-Nearest Neighbors (k-NN)

We will predict if a wine is good or not. We have the data on good, so this is a problem of supervised classification.

Consider the typical distance matrix that is often used for cluster analysis of observations. If we choose something like Euclidean distance as a metric, each point in the matrix gives the value of how far an observation is from some other, given their respective values on a set of variables.

k-NN approaches exploit this information for predictive purposes. Let us take a classification example, and $k = 5$ neighbors. For a given observation x_i, find the 5 closest k neighbors in terms of Euclidean distance based on the predictor variables. The class that is predicted is whatever class the majority of the neighbors are labeled as. For continuous outcomes we might take the mean of those neighbors as the prediction.

So how many neighbors would work best? This is an example of a tuning parameter, i.e. k, for which we have no knowledge about its value without doing some initial digging. As such we will select the tuning parameter as part of the validation process.

Implementation

The `caret` package provides several techniques for validation such as k-fold, bootstrap, leave-one-out and others. We will use 10-fold cross validation. We will also set up a set of values for k to try out.

`train` is the function used to fit the models. You can check all available methods here. This function is used for: * evaluate, using resampling, the effect of model tuning parameters on performance * choose the "optimal" model across these parameters * estimate model performance from a training set

You can control training parameters such as resampling method and iterations with the `cv_opts` function. In this case we will use a k-fold cross-validation (cv) with 5 resampling iterations.

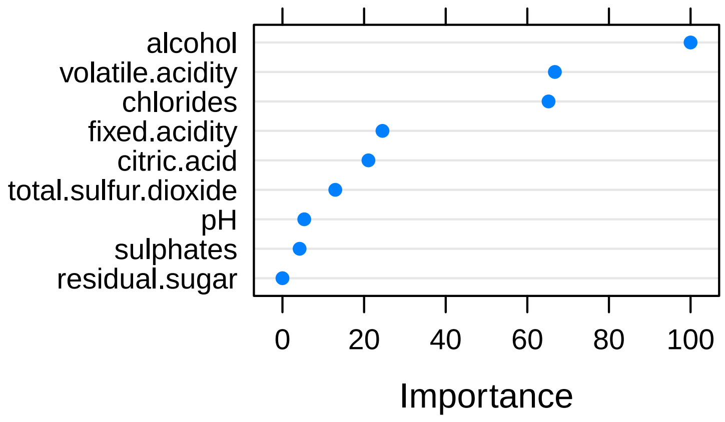
```
cv_opts = trainControl(method="cv", number=10)
```

```
knn_opts = data.frame(.k=c(seq(3, 11, 2), 25, 51, 101)) #odd to avoid ties
knn_model = train(good~., data=wine_train, method="knn",
                  preProcess="range", trControl=cv_opts,
                  tuneGrid = knn_opts)
knn_model
```

```
## k-Nearest Neighbors
##
## 5199 samples
##    9 predictor
##    2 classes: 'Bad', 'Good'
##
## Pre-processing: re-scaling to [0, 1] (9)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 4679, 4678, 4679, 4679, 4680, 4679, ...
## Resampling results across tuning parameters:
##
##   k    Accuracy   Kappa
##     3  0.7518692  0.4574756
##     5  0.7453311  0.4405051
##     7  0.7512978  0.4518213
##     9  0.7497564  0.4476991
##    11  0.7534099  0.4561257
##    25  0.7509106  0.4469177
##    51  0.7468703  0.4326833
##   101  0.7451462  0.4272734
##
## Accuracy was used to select the
##  optimal model using the largest value.
## The final value used for the model was k
##  = 11.
```

Additional information reflects the importance of predictors. For most methods accessed by `caret`, the default variable importance metric regards the area under the curve or AUC from a ROC curve analysis with regard to each predictor, and is model independent. This is then normalized so that the least important is 0 and most important is 100. Another thing one could do would require more work, as caret doesn't provide this, but a simple loop could still automate the process. For a given predictor x, re-run the model without x, and note the decrease (or increase for poor variables) in accuracy that results. One can then rank order those results. I did so with this problem and notice that only alcohol content and volatile acidity were even useful for this model. K-NN is susceptible to irrelevant information (you're essentially determining neighbors on variables that don't matter), and one can see this in that, if only those two predictors are retained, test accuracy is the same (actually a slight increase).

```
dotPlot(varImp(knn_model))
```



## Validation Now lets see how it works on the test set

```
preds_knn = predict(knn_model, wine_test[,-10])
confusionMatrix(preds_knn, wine_test[,10], positive='Good')
```

```
## Confusion Matrix and Statistics
```

```
##
##           Reference
## Prediction Bad Good
##       Bad  276  130
##       Good 200  692
##
##                Accuracy : 0.7458
##                  95% CI : (0.7211, 0.7693)
##     No Information Rate : 0.6333
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.4351
##
##  Mcnemar's Test P-Value : 0.0001457
##
##             Sensitivity : 0.8418
##             Specificity : 0.5798
##          Pos Pred Value : 0.7758
##          Neg Pred Value : 0.6798
##              Prevalence : 0.6333
##          Detection Rate : 0.5331
##    Detection Prevalence : 0.6872
##       Balanced Accuracy : 0.7108
##
##        'Positive' Class : Good
##
```

We get a lot of information here, but to focus on accuracy, we get around 75.04%. The lower bound (and p-value) suggests we are statistically predicting better than the no information rate (randomly guessing).

Neural networks

Neural nets have been around for a long while as a general concept in artificial intelligence and even as a machine learning algorithm, and often work quite well. In some sense they can be thought of as nonlinear regression. Visually however, we can see them in as layers of inputs and outputs. Weighted combinations of the inputs are created and put through some function (e.g. the sigmoid function) to produce the next layer of inputs. This next layer goes through the same process to produce either another layer or to predict the output, which is the final layer. All the layers between the input and output are usually referred to as 'hidden' layers. If there were no hidden layers then it becomes the standard regression problem.

One of the issues with neural nets is determining how many hidden layers and how many hidden units in a layer. Overly complex neural nets will suffer from a variance problem and be less generalizable, particularly if there is less relevant information in the training data. Along with the complexity is the notion of weight decay, however this is the same as the regularization function we discussed in a previous section, where a penalty term would be applied to a norm of the weights.

Parallel processing

In general, machine learning algorithms are computationally intensive, requiring a lot of computing power. We can use parallel processing to significantly reduce computing time of some of these algorithms[1]. If you are not set up for utilizing multiple processors the following might be relatively slow. You can replace the method with `nnet` and shorten the `tuneLength` to 3 which will be faster without much loss of accuracy. Also, the function we are using has only one hidden layer, but the other neural net methods accessible via the `caret` package may allow for more, though the gains in prediction with additional layers are likely to be modest relative to complexity and computational cost. In addition, if the underlying function has additional arguments, you may pass those on in the train function itself.

We will use parallel processing with type SOCK[2].

[1] It is also more efficient to computing NN using GPU instead of the more general CPUS.

[2] If you are using a Linux/GNU or macOS you can use the FORK type. In this case, the environment is linked in all processors.

```
library(doSNOW)
cl <- makeCluster(3, type = "SOCK")
registerDoSNOW(makeCluster(3, type = "SOCK"))
```

Implementation

In here I reduce the nummber of maximum iterations `maxit` to save time.

```
nnet_model = train(good~., data=wine_train,
                   method="avNNet",
                   trControl=cv_opts,
                   preProcess="range",
                   tuneLength=5,
                   trace=F,
                   maxit=10)
nnet_model
```

```
## Model Averaged Neural Network
##
## 5199 samples
##    9 predictor
##    2 classes: 'Bad', 'Good'
```

```
##
## Pre-processing: re-scaling to [0, 1] (9)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 4680, 4679, 4678, 4680, 4679, 4679, ...
## Resampling results across tuning parameters:
##
##   size  decay  Accuracy   Kappa
##   1     0e+00  0.7284013  0.3965111
##   1     1e-04  0.7287841  0.3885902
##   1     1e-03  0.7253277  0.3791323
##   1     1e-02  0.7291661  0.4007401
##   1     1e-01  0.7314834  0.3944406
##   3     0e+00  0.7089882  0.3167041
##   3     1e-04  0.7212815  0.3746202
##   3     1e-03  0.7216783  0.3634109
##   3     1e-02  0.7289812  0.3925674
##   3     1e-01  0.7243551  0.3574783
##   5     0e+00  0.7091598  0.3136761
##   5     1e-04  0.7184094  0.3554943
##   5     1e-03  0.7243629  0.3685528
##   5     1e-02  0.7280134  0.3808471
##   5     1e-01  0.7226288  0.3688245
##   7     0e+00  0.7255075  0.3693651
##   7     1e-04  0.7243606  0.3796744
##   7     1e-03  0.7199446  0.3509379
##   7     1e-02  0.7145766  0.3321129
##   7     1e-01  0.7251306  0.3804193
##   9     0e+00  0.7258972  0.3796424
##   9     1e-04  0.7307127  0.3972374
##   9     1e-03  0.7289775  0.3849989
##   9     1e-02  0.7276339  0.3877238
##   9     1e-01  0.7305152  0.3867421
##
## Tuning parameter 'bag' was held constant
##  at a value of FALSE
## Accuracy was used to select the
##  optimal model using the largest value.
## The final values used for the model
##  were size = 1, decay = 0.1 and bag = FALSE.
```

Once you've finished working with your cluster, it's good to clean up and stop the cluster child processes (quitting R will also stop all of the child processes).

```
stopCluster(cl)
```

Validation

```
preds_nnet = predict(nnet_model, wine_test[,-10])
confusionMatrix(preds_nnet, wine_test[,10], positive='Good')
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Bad Good
##       Bad  289  163
##       Good 187  659
##
##                Accuracy : 0.7304
##                  95% CI : (0.7053, 0.7543)
##     No Information Rate : 0.6333
##     P-Value [Acc > NIR] : 7.158e-14
##
##                   Kappa : 0.4132
##
##  Mcnemar's Test P-Value : 0.2189
##
##             Sensitivity : 0.8017
##             Specificity : 0.6071
##          Pos Pred Value : 0.7790
##          Neg Pred Value : 0.6394
##              Prevalence : 0.6333
##          Detection Rate : 0.5077
##    Detection Prevalence : 0.6518
##       Balanced Accuracy : 0.7044
##
##        'Positive' Class : Good
##
```

More information on NNs

- https://www.datacamp.com/community/tutorials/neural-network-models-r
- https://www.analyticsvidhya.com/blog/2017/09/creating-visualizing-neural-network-in-r/
- https://datascienceplus.com/neuralnet-train-and-test-neural-networks-using-r/

Ensemble method

You can combine the predictions of multiple caret models using the `caretEnsemble` package.

```
library(caretEnsemble)
```

```
## Warning: package 'caretEnsemble' was built
## under R version 3.5.3
```

Given a list of caret models, the `caretStack` function can be used to specify a higher-order model to learn how to best combine the predictions of sub-models together.

Let's first look at creating 4 sub-models for the ionosphere dataset, specifically:

- Linear Discriminate Analysis (LDA)
- Logistic Regression (via Generalized Linear Model or GLM)
- k-Nearest Neighbors (kNN)
- Random forest(rf)

Below is an example that creates these 4 sub-models. This is a slow process.

```
# Example of Stacking algorithms
# create submodels
control <- trainControl(method="repeatedcv", number=10, repeats=3, savePredictions=TRUE, classProbs=TRUI
algorithmList <- c('lda', 'glm', 'knn', 'rf')
set.seed(1234)
models <- caretList(good~., data=wine_train, trControl=control, methodList=algorithmList)
```

```
## Warning in trControlCheck(x = trControl,
## y = target): x$savePredictions == TRUE is
## depreciated. Setting to 'final' instead.
```

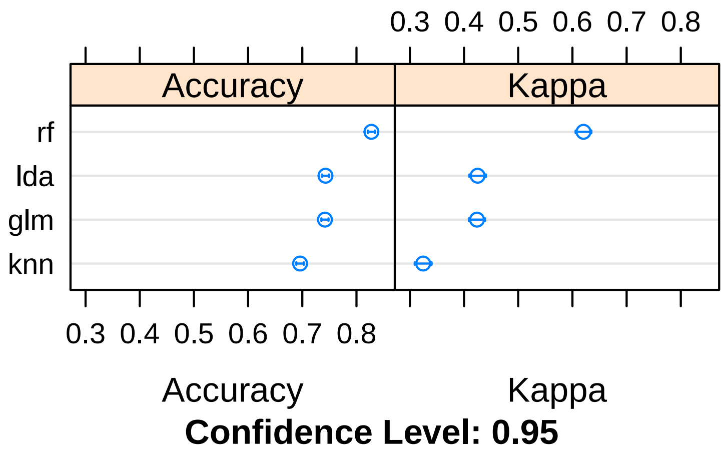
```
## Warning in trControlCheck(x = trControl, y
## = target): indexes not defined in trControl.
## Attempting to set them ourselves, so each
## model in the ensemble will have the same
## resampling indexes.
```

```
results <- resamples(models)
summary(results)
```

```
##
## Call:
```

```
## summary.resamples(object = results)
##
## Models: lda, glm, knn, rf
## Number of resamples: 30
##
## Accuracy
##          Min.   1st Qu.    Median      Mean
## lda 0.7013487 0.7325991 0.7439844 0.7428332
## glm 0.7038462 0.7258766 0.7461538 0.7418079
## knn 0.6653846 0.6826923 0.6971154 0.6958994
## rf  0.7861272 0.8177885 0.8291747 0.8275347
##       3rd Qu.      Max. NA's
## lda 0.7548077 0.7750000    0
## glm 0.7525289 0.7677543    0
## knn 0.7081131 0.7480769    0
## rf  0.8360577 0.8557692    0
##
## Kappa
##          Min.   1st Qu.    Median      Mean
## lda 0.3258011 0.4081795 0.4271401 0.4250502
## glm 0.3289198 0.3908777 0.4299979 0.4238487
## knn 0.2476846 0.2963154 0.3305481 0.3244994
## rf  0.5292575 0.5996030 0.6225940 0.6204321
##       3rd Qu.      Max. NA's
## lda 0.4531663 0.5011152    0
## glm 0.4489842 0.4822137    0
## knn 0.3492237 0.4388797    0
## rf  0.6432326 0.6858892    0
```

```
dotplot(results)
```



We can see that the Random Forests creates the most accurate model with an accuracy of 82.75%.

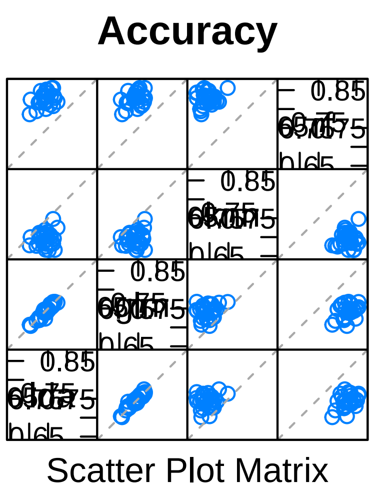
When we combine the predictions of different models using stacking, it is desirable that the predictions made by the sub-models have low correlation. This would suggest that the models are skillful but in different ways, allowing a new classifier to figure out how to get the best from each model for an improved score.

If the predictions for the sub-models were highly corrected (>0.75) then they would be making the same or very similar predictions most of the time reducing the benefit of combining the predictions.

```
# correlation between results
modelCor(results)
```

```
##            lda       glm       knn        rf
## lda 1.0000000 0.9439576 0.1620105 0.4171421
## glm 0.9439576 1.0000000 0.1901886 0.4099201
## knn 0.1620105 0.1901886 1.0000000 0.1751962
## rf  0.4171421 0.4099201 0.1751962 1.0000000
```

```
splom(results)
```



We can see the LDA and GLM have high correlation and all other pairs of predictions have generally low correlation. Let's eliminate the glm method because it has the lowest accuracy.

```
algorithmList <- c('lda', 'knn', 'rf')
set.seed(1234)
models <- caretList(good~.,
                    data=wine_train,
                    trControl=control,
                    methodList=algorithmList)
```

```
## Warning in trControlCheck(x = trControl,
## y = target): x$savePredictions == TRUE is
## depreciated. Setting to 'final' instead.

## Warning in trControlCheck(x = trControl, y
## = target): indexes not defined in trControl.
## Attempting to set them ourselves, so each
## model in the ensemble will have the same
## resampling indexes.
```

```
results <- resamples(models)
```

Let's combine the predictions of the classifiers using a simple linear model. `caretStack` finds a a good linear combination of chosen classification models. It can use linear regression, elastic net regression, or greedy optimization.

```
# stack using glm
stackControl <- trainControl(method="repeatedcv",
                             number=10,    #number of resampling iterations
                             repeats=3,    #the number of complete sets of folds to compute
```

```
                              savePredictions=TRUE,
                              classProbs=TRUE)
set.seed(1234)
stack.glm <- caretStack(models,
                        method="glm",
                        metric="Accuracy",
                        trControl=stackControl)
print(stack.glm)
```

```
## A glm ensemble of 2 base models: lda, knn, rf
##
## Ensemble results:
## Generalized Linear Model
##
## 15597 samples
##     3 predictor
##     2 classes: 'Bad', 'Good'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 14038, 14038, 14038, 14036, 14038, 14037, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.8280646  0.6237795
```

We can see that we have lifted the accuracy to 75.34% which is a small improvement over using SVM alone. This is also an improvement over using random forest alone on the dataset, as observed above.

We can also use more sophisticated algorithms to combine predictions in an effort to tease out when best to use the different methods. In this case, we can use the random forest algorithm to combine the predictions. This method is slower than using `glm`.

```
# stack using random forest
set.seed(1234)
stack.rf <- caretStack(models,
                       method="rf",
                       metric="Accuracy",
                       trControl=stackControl)
```

```
## note: only 2 unique complexity parameters in default grid. Truncating the grid to 2 .
```

```r
print(stack.rf)
```

```
## A rf ensemble of 2 base models: lda, knn, rf
##
## Ensemble results:
## Random Forest
##
## 15597 samples
##     3 predictor
##     2 classes: 'Bad', 'Good'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 14038, 14038, 14038, 14036, 14038, 14037, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##   2     0.8296253  0.6288798
##   3     0.8281290  0.6258433
##
## Accuracy was used to select the
##  optimal model using the largest value.
## The final value used for the model was
##  mtry = 2.
```

We can see that this has lifted the accuracy to 96.26% an impressive improvement on SVM alone.

k-means clustering

K-means clustering is one of e simplest and popular unsupervised machine learning algorithms. The objective of K-means is simple: group similar data points into clusters and discover underlying patterns. A cluster refers to a collection of data points aggregated together because of certain similarities.

You'll define a target number k, which refers to the number of centroids you need in the dataset. A centroid is the imaginary or real location representing the center of the cluster.

Every data point is allocated to each of the clusters through reducing the in-cluster sum of squares.

In other words, the K-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible.

The 'means' in the K-means refers to averaging of the data; that is, finding the centroid.

To process the learning data, the K-means algorithm in data mining starts with a first group of randomly selected centroids, which are used as the beginning points for every cluster, and then performs iterative (repetitive) calculations to optimize the positions of the centroids

It halts creating and optimizing clusters when either:

- The centroids have stabilized - there is no change in their values because the clustering has been successful.
- The defined number of iterations has been achieved.

Implementation

In R, we use

```
stats::kmeans(x, centers = 3, nstart = 10)
```

where

- x is a numeric data matrix
- centers is the pre-defined number of clusters
- the k-means algorithm has a random component and can be repeated nstart times to improve the returned model

To learn about k-means, let's use the iris dataset with the sepal and petal length variables only (to facilitate visualisation). Create such a data matrix and name it `iris_subset`.

```
iris_subset <- iris %>% select(Sepal.Length, Petal.Length)
```

```
cl <- kmeans(iris_subset, 3, nstart = 10)
cl
```

```
## K-means clustering with 3 clusters of sizes 58, 51, 41
##
## Cluster means:
##   Sepal.Length Petal.Length
## 1     5.874138     4.393103
## 2     5.007843     1.492157
## 3     6.839024     5.678049
##
## Clustering vector:
##   [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [21] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [41] 2 2 2 2 2 2 2 2 2 2 3 1 3 1 1 1 1 1 1 1
##  [61] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 3 1 1
##  [81] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1
## [101] 3 1 3 3 3 3 1 3 3 3 3 3 3 1 1 3 3 3 3 1
## [121] 3 1 3 1 3 3 1 1 3 3 3 3 3 3 3 3 3 3 1 3
## [141] 3 3 1 3 3 3 1 3 3 1
##
## Within cluster sum of squares by cluster:
## [1] 23.508448  9.893725 20.407805
##  (between_SS / total_SS =  90.5 %)
##
## Available components:
##
## [1] "cluster"      "centers"
## [3] "totss"        "withinss"
## [5] "tot.withinss" "betweenss"
## [7] "size"         "iter"
## [9] "ifault"
```

```r
plot(iris_subset, col = cl$cluster)
```

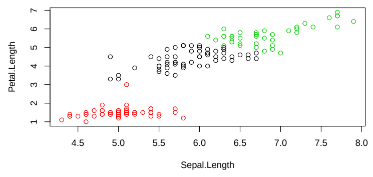


Figure 2: k-means algorithm on sepal and petal lengths

Due to the random initialization, one can obtain different clustering results. When k-means is run multiple times, the best outcome, i.e. the one that generates the smallest total within cluster sum of squares (SS), is selected. The total within SS is calculated as:

For each cluster results:

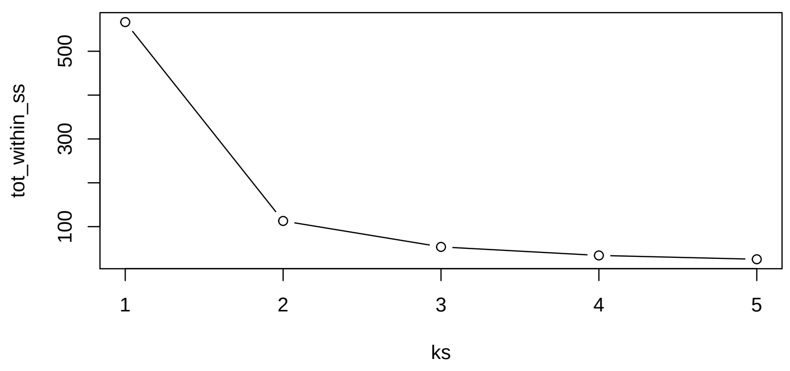
- for each observation, determine the squared euclidean distance from observation to centre of cluster
- sum all distances

Note that this is a local minimum; there is no guarantee to obtain a global minimum.

How to determine the number of clusters

We can check how the squared distances changes with different values of k (centers).

```r
ks <- 1:5
tot_within_ss <- sapply(ks, function(k) {
    cl <- kmeans(iris_subset, k, nstart = 10)
    cl$tot.withinss
})
plot(ks, tot_within_ss, type = "b")
```



More information on k-means

- `https://www.datanovia.com/en/lessons/k-means-clustering-in-r-algorith-and-practical-examples/`
- `https://towardsdatascience.com/clustering-analysis-in-r-using-k-means-73eca4fb7967`

Literature

- `http://topepo.github.io/caret/index.html`
- `https://lgatto.github.io/IntroMachineLearningWithR/`
- An introduction to Machine Learning with Applications in R. `http://web.ipac.caltech.edu/staff/fmasci/home/astro_refs/ML_inR.pdf`