STAT30270 Statistical Machine Learning End Of Term Assignment

Fanahan McSweeney - 20203868

April 29, 2021

Question 5. Data Analysis Task

This task requires the classification of data relating to movie reviews in order to predict whether the reviews are *negative* or *positive*, so we are required to build a suitable binary classifier. The dataset includes 80 numerical features which can be used as predictor variables in our model, and a class column indicating whether the review is *negative* or *positive* which will be the target variable for this task.

(a) Use at least 3 of the supervised classification methods described in this course to predict the sentiment class label of a review on the basis of the numerical features.

The 3 supervised classification methods I have chosen to use for this task are as follows:

- Random Forests
- Boosting
- Support Vector Machines

Random forests are comprised of an ensemble of classification trees, where only a random subset of the available predictor variables are used to form each tree. For this task, I plan to use tuning methods to pick suitable values for the following parameters:

- *mtry* - the number of predictor variables randomly sampled for each tree split.
- *ntree* - the total number of trees to use in the model.

Boosting is another ensemble method which fits each new classification tree to a modified version of the original data, with the aim of strengthening the models ability at classifying the most difficult observations to classify. I will use the Breiman method for updating the observation weights in the algorithm, and I plan to use tuning methods to pick suitable values for the following parameters:

- *maxdepth* - the maximum depth of any node of the final tree.
- *mfinal* - The number of iterations (or trees) used in the model.

Support Vector Machines (SVMs) are another form of supervised model which aim to to separate data using hyperplanes. Kernel functions can be employed with these models to perform non-linear classification in a very efficient manner. There are many forms of kernel functions that we can employ with SVMs, but for this task I will only consider the Gaussian radial basis function (RBF) kernel. This is one the most commonly used kernels and can separate data using very complex boundaries. I plan to use tuning methods to pick suitable values for the following parameters:

- C - cost.
- *sigma* - Gaussian RBF kernel hyperparameter.

(b) Employ an appropriate framework to compare and tune the different methods considered, evaluating and discussing their relative merits.

For this task, I plan to use the *caret* package to perform k-fold cross-validation to tune each model for the classification methods mentioned above. Firstly, the data must be loaded from the *data_rotten_tomatoes_review.csv* file. The first column in the dataset (containing the text used in each review) is not of use for this task and can be dropped from the data. I will also convert the *class* column containing the *positive* or *negative* classification for each review to a factor, so that it can be used with the various classification algorithm functions that I will be using.

```r
# Load relevant libraries
library(caret)
library(randomForest)
library(adabag)
library(kernlab)

# read in reviews data
reviews_original <- read.csv("Data/data_rotten_tomatoes_review.csv")
# drop the phrase column
reviews <- reviews_original[,-1]
# convert class column to factor
reviews$class <- as.factor(reviews$class)
```

Next, I will split the data into 2 sets, one to be used for both training and validation, and the other to be set aside for testing later. I have chosen to split the data 80:20, so that 80% of the total available data can be used for training and validation, and the remaining 20% will be reserved for testing the final model that I choose.

I will also standardise the data so that each feature has a mean value of 0 and standard deviation of 1. Rather than scaling all of the data together, I have performed the standardisation on the training/validation data set first, as in reality we would have no prior knowledge of the data in the test set. Instead, I will save the mean and standard deviation values of the training/validation set, and I will use these values to standardise the test set as well (rather than using the mean and standard deviation of the test data).

```r
# split data into training/validation and test sets (80:20 split)
train_val <- createDataPartition(reviews$class, p=0.80, list=FALSE)
reviews_trainval <- reviews[train_val,]
reviews_test <- reviews[-train_val,]

# get mean and standard deviation of reviews training/validation data
dat0 <- reviews_trainval[-ncol(reviews_trainval)] # get dataframe with all numeric columns only
trainval_mean <- apply(dat0, 2, mean)
trainval_sd <- apply(dat0, 2, sd)
# for each column, subtract mean and divide by standard deviation
dat0 <- sweep(sweep(dat0, 2, trainval_mean), 2, trainval_sd, "/")
# replace original data with standardised data
reviews_trainval[-ncol(reviews_trainval)] <- dat0

# standardise the test data using same mean and standard deviation as training/validation set
dat0 <- reviews_test[-ncol(reviews_test)] # get dataframe with all numeric columns only
reviews_test[-ncol(reviews_test)] <- sweep(sweep(dat0, 2, trainval_mean), 2, trainval_sd, "/")
```

The *caret* package has cross-validation functionality inbuilt, which can be used for tuning each of the models. Below, I have set the cross-validation configuration to perform 4-fold cross-validation with 5 replications which will be used to estimate the predictive performance of each model.

```
# perform 4-fold cross-validation with 5 replications
train_ctrl <- trainControl(method="repeatedcv", number=4, repeats=5, verboseIter=T)
```

Random Forest Classifier Tuning

The first classifier I will tune is a random forest. As mentioned above, I plan to test a range of values for the *mtry* and *ntree* parameters to determine suitable values for this particular task. The *caret* package doesn't have an inbuilt random forest model that allows for the tuning of both of these parameters, so I have decided to define a new random forest function for the package using the *randomForest* function (from the *randomForest* package) by defining the appropriate elements required by the *caret* function.

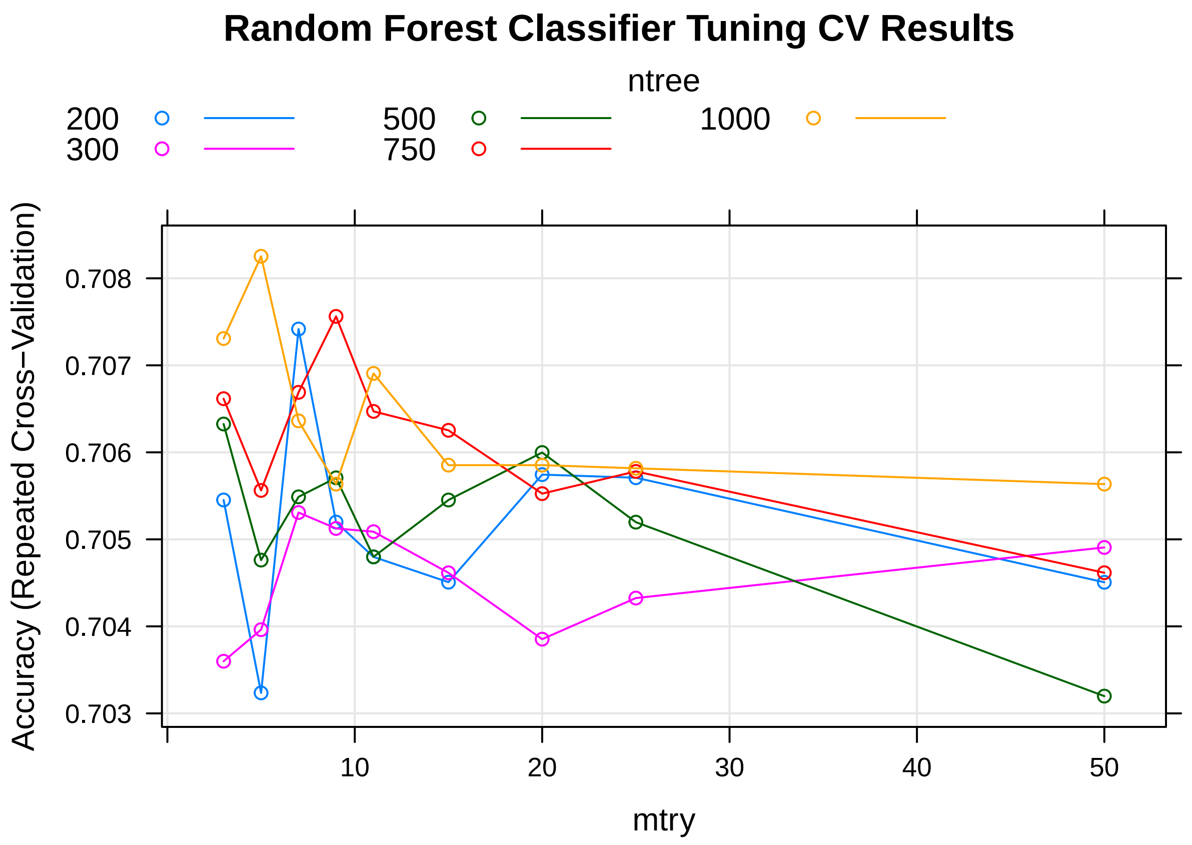
```
# Define custom random forest function algorithm to use with caret (tune mtry, ntree parameters) -
# using the "randomForest" function from the "randomForest "library
# (Define all neccesaary elements/functions that caret requires)
myRF <- list(type="Classification", library="randomForest", loop=NULL)
myRF$parameters <- data.frame(parameter=c("mtry", "ntree"), class=rep("numeric", 2),
                              label=c("mtry", "ntree"))
myRF$grid <- function(x, y, len=NULL, search="grid") {}
myRF$fit <- function(x, y, wts, param, lev, last, weights, classProbs, ...) {
  randomForest(x, y, mtry=param$mtry, ntree=param$ntree, ...)
}
myRF$predict <- function(modelFit, newdata, preProc=NULL, submodels=NULL)
  predict(modelFit, newdata)
myRF$prob <- function(modelFit, newdata, preProc=NULL, submodels=NULL)
  predict(modelFit, newdata, type="prob")
myRF$sort <- function(x) x[order(x[,1]),]
myRF$levels <- function(x) x$classes
```

Now that we have a suitable random forest function that can be used within the *caret* package, a tuning grid can be set up to describe each combination of the parameters we want to trial. The *train* function is then used to perform cross-validation on all possible model configurations based on the tuning grid, using the training/validation data set defined earlier.

```
# Random Forest
# ---------------------------------------
# define tuning grid of parameters to use
tune_grid1 <- expand.grid(.mtry=c(3, 5, 7, 9, 11, 15, 20, 25, 50),
                          .ntree=c(200, 300, 500, 750, 1000))
# perform cross validation on all variations of model
fit_rf_cv <- train(class~., data=reviews_trainval,
                   method=myRF,
                   trControl=train_ctrl,
                   tuneGrid=tune_grid1)
```

We can visualise the results of the random forest classifier cross-validation using an appropriate plot as shown below. The optimum number of predictors (*mtry*) and number of trees (*ntree*) based on the cross-validation results are also printed below.

```
plot(fit_rf_cv, main="Random Forest Classifier Tuning CV Results")
```



```
cat("\nOptimum number of predictors:", fit_rf_cv$bestTune$mtry,
    "\nOptimum number of trees:", fit_rf_cv$bestTune$ntree,"\n ")
```

```
##
## Optimum number of predictors: 5
## Optimum number of trees: 1000
##
```

Parallel Processing

Another very useful feature of the *caret* package is that it can be used with parallel computing. Below, I have set up parallel processing to allow the use of 3 cores on my machine (as I have 4 cores in total).

```
# Set up parallel computing
library(doParallel)
cl <- makeCluster(3) # I have 4 cores on my machine (keep 1 free)
registerDoParallel(cl) # start parallel computing backend
```

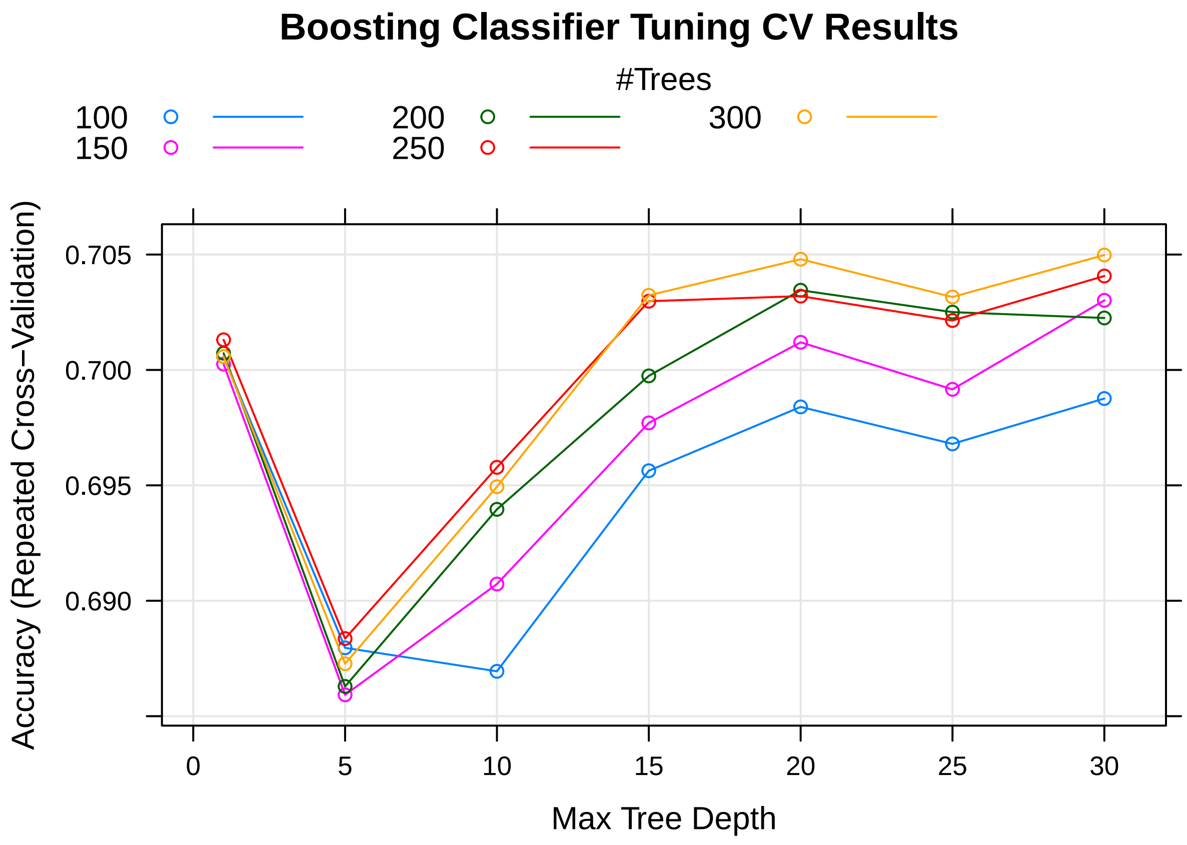
Boosting Classifier Tuning

Next, I will look at tuning a boosting classifier. Again, as mentioned above I plan to test a range of values for the *maxdepth* and *mfinal* parameters to determine suitable values for this particular task. The *AdaBoost.M1* model can be used to employ a boosting classifier in the *caret* package. Again, I have set up a tuning grid of all combinations of the parameter values that I would like to trial. Cross-validation can then be used to estimate the performance of each boosting model configuration.

```r
# Boosting
# ----------------------------------------
# define tuning grid of parameters to use
tune_grid_boost <- expand.grid(coeflearn=c("Breiman"),
                               maxdepth=c(1, seq(5, 30, 5)),
                               mfinal=seq(100, 300, 50))
# perform cross validation on all variations of model
fit_boost_cv <- train(class~., data=reviews_trainval,
                      method="AdaBoost.M1",
                      trControl=train_ctrl,
                      tuneGrid=tune_grid_boost)
```

We can visualise the results of the boosting classifier cross-validation using an appropriate plot as shown below. The optimum maximum depth (*maxdepth*) and number of trees (*mfinal*) based on the cross-validation results are also printed below.

```r
plot(fit_boost_cv, main="Boosting Classifier Tuning CV Results")
```



```r
cat("\nOptimum max. depth:", fit_boost_cv$bestTune$maxdepth,
    "\nOptimum number of trees:", fit_boost_cv$bestTune$mfinal, "\n ")
```

```
##
## Optimum max. depth: 30
## Optimum number of trees: 300
##
```

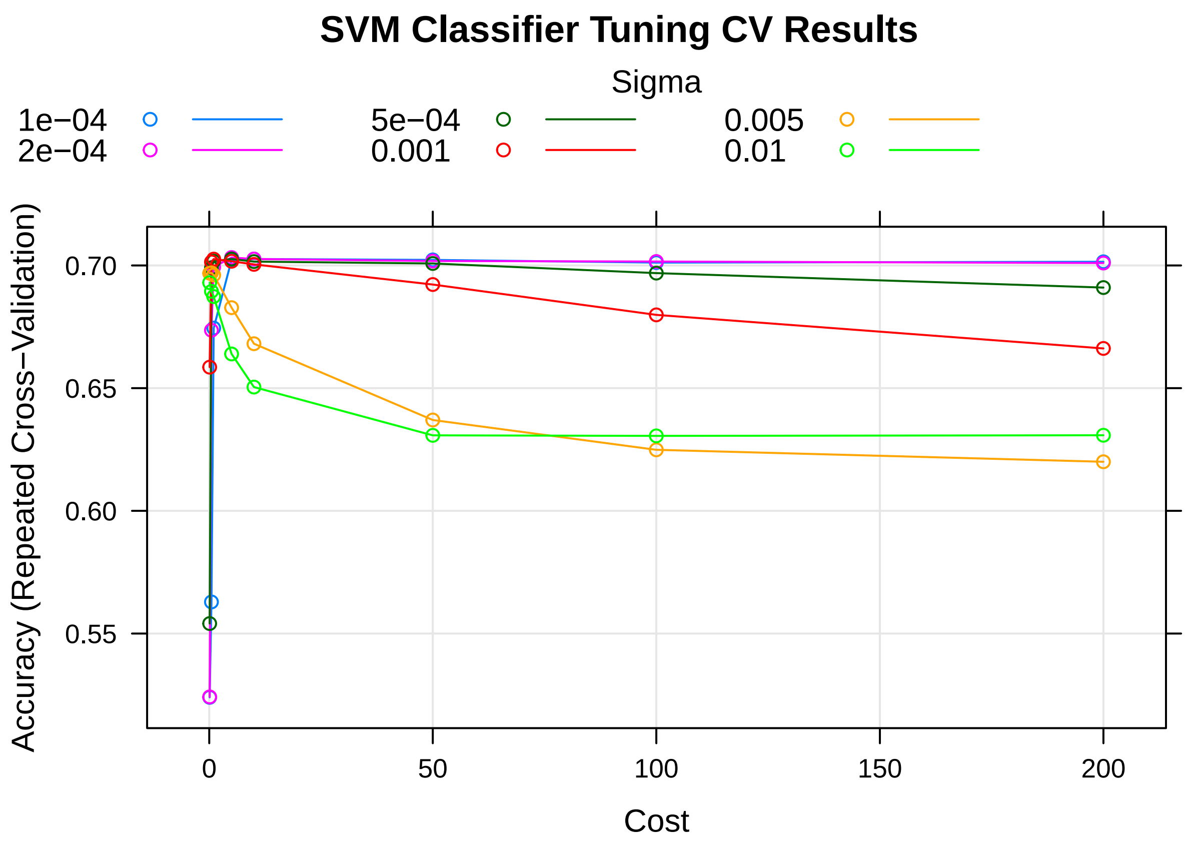
Support Vector Machine Classifier Tuning

The final classifier I will tune is a Support Vector Machine classifier. Again, as mentioned above I will use the Gaussian RBF kernel, and I plan to test a range of values for the C and *sigma* parameters to determine suitable values for this particular task. The *svmRadial* model can be used to employ a SVM Gaussian RBF classifier in the *caret* package. Again, I have set up a tuning grid of all combinations of the parameter values that I would like to trial. Cross-validation can then be used to estimate the performance of each SVM model configuration.

```r
# Support Vector Machines
# --------------------------------------
# define tuning grid of parameters to use
tune_grid_svm <- expand.grid(C=c(0.1, 0.5, 1, 5, 10, 50, 100, 200),
                             sigma=c(0.0001, 0.0002, 0.0005, 0.001, 0.005, 0.01))
# define tuning grid of parameters to use
fit_svm_grbf_cv <- train(class~., data=reviews_trainval,
                      method="svmRadial",
                      trControl=train_ctrl,
                      tuneGrid=tune_grid_svm)
```

We can visualise the results of the SVM Gaussian RBF classifier cross-validation using an appropriate plot as shown below. The optimum cost (C) and sigma hyperparameter value (*sigma*) based on the cross-validation results are also printed below.

```r
plot(fit_svm_grbf_cv, main="SVM Classifier Tuning CV Results")
```



```r
cat("\nOptimum cost:", fit_svm_grbf_cv$bestTune$C,
    "\nOptimum sigma:", fit_svm_grbf_cv$bestTune$sigma)
```

```
##
## Optimum cost: 5
## Optimum sigma: 2e-04
```

After completing the cross-validation to tune each model using the *caret* package, the parallel computing can be stopped.

```r
# Stop parallel computing cluster
stopCluster(cl)
```

While we could choose the best overall model from the 3 classification algorithms based directly off the results obtained above, it would be preferable to compare each model using the same sets of training and validation folds to give a more fair comparison, particularly as a relatively small number of folds and replications was used (due to excessive computation time!).

Therefore I have decided to perform further cross-validation, using the optimum parameters for each classifier as determined from the tuning cross-validation performed earlier. This time I will employ my own cross-validation algorithm rather than using the inbuilt functionality in the *caret* package as I will be able to train and test each classifier using the same training and validation folds, and I will also use a larger number of replications (20 rather than 5 used previously) as we now only need to consider the 3 tuned models in our cross-validation thus reducing computation time, which will help to improve our estimate of each model's performance.

The function below has been created in order to calculate the classification accuracy obtained by each model.

```r
# function to compute classification accuracy
class_acc <- function(y, yhat) {
  tab <- table(y, yhat)
  return(sum(diag(tab))/sum(tab) )}
```

The following code performs cross-validation on each of the optimum classifiers using 2 *for* loops to loop through each fold and replicate, in each case training each model on the current set of training data and evaluating their performance on the current validation set.

```r
# total number of rows in training/validation set
N_trainval <- nrow(reviews_trainval)
K <- 4 # set number of folds
R <- 20 # set number of replicates

# create vector of lists to store cross-validation accuracy outputs for the 3 models
out_svm <- vector("list", R) # store accuracy output (SVM)
out_rf <- vector("list", R) # store accuracy output (random forest)
out_boost <- vector("list", R) # store accuracy output (boosting)

# repeat the following steps R times (number of replicates)
for(r in 1:R) {
  # matrices to save accuracy of the classifiers in the K folds (for the current rep)
  acc_svm <- matrix(NA, K)
  acc_rf <- matrix(NA, K)
  acc_boost <- matrix(NA, K)

  # split training/validation data into K folds
  folds <- rep(1:K, ceiling(N_trainval/K))
  folds <- sample(folds) # random permute
  folds <- folds[1:N_trainval] # ensure we got N_trainval data points

  # repeat the following steps K times (for each of the K folds)
  for(k in 1:K) {
    # set current train data to all data in training/validation set minus the current fold 'k'
    train_fold <- which(folds!=k)
    # set current validation data to the current fold 'k'
    validation <- setdiff(1:N_trainval, train_fold)

    ## SVM model
    # fit SVM model to current training data, with optimum C and sigma value selection
    fit_svm <- ksvm(class~., data=reviews_trainval[train_fold,], type="C-svc", kernel="rbfdot",
                    C=fit_svm_grbf_cv$bestTune$C,
                    kpar=list(sigma=fit_svm_grbf_cv$bestTune$sigma))
    # predict classes of current validation set
    pred_svm <- predict(fit_svm, newdata=reviews_trainval[validation,])
    # calculate classification accuracy of the validation data
    acc_svm[k] <- class_acc(pred_svm, reviews_trainval[validation,ncol(reviews_trainval)])

    ## Random Forest Model
    # fit random forest model to current training data, with optimum
    fit_rf <- randomForest(class ~ ., data=reviews_trainval[train_fold,],
                           mtry=fit_rf_cv$bestTune$mtry,
                           ntree=fit_rf_cv$bestTune$ntree)
    # predict classes of current validation set
    pred_rf <- predict(fit_rf, newdata=reviews_trainval[validation,], type="class")
    # calculate classification accuracy of the validation data
    acc_rf[k] <- class_acc(pred_rf, reviews_trainval[validation,ncol(reviews_trainval)])

    ## Boosting Model
    # fit boosting model to current training data
    fit_boost <- boosting(class ~ ., data=reviews_trainval[train_fold,], boos = FALSE,
                          coeflearn = "Breiman", mfinal=fit_boost_cv$bestTune$mfinal,
                          control=rpart.control(maxdepth=fit_boost_cv$bestTune$maxdepth))
    # predict classes of current validation set
    pred_boost <- predict(fit_boost, newdata=reviews_trainval[validation,])
    # calculate classification accuracy of the validation data
    acc_boost[k] <- class_acc(pred_boost$class, reviews_trainval[validation,ncol(reviews_trainval)])
  }

  # add classification accuracy values for all models for current replicate to output lists
  out_svm[[r]] <- acc_svm
  out_rf[[r]] <- acc_rf
  out_boost[[r]] <- acc_boost
}
```

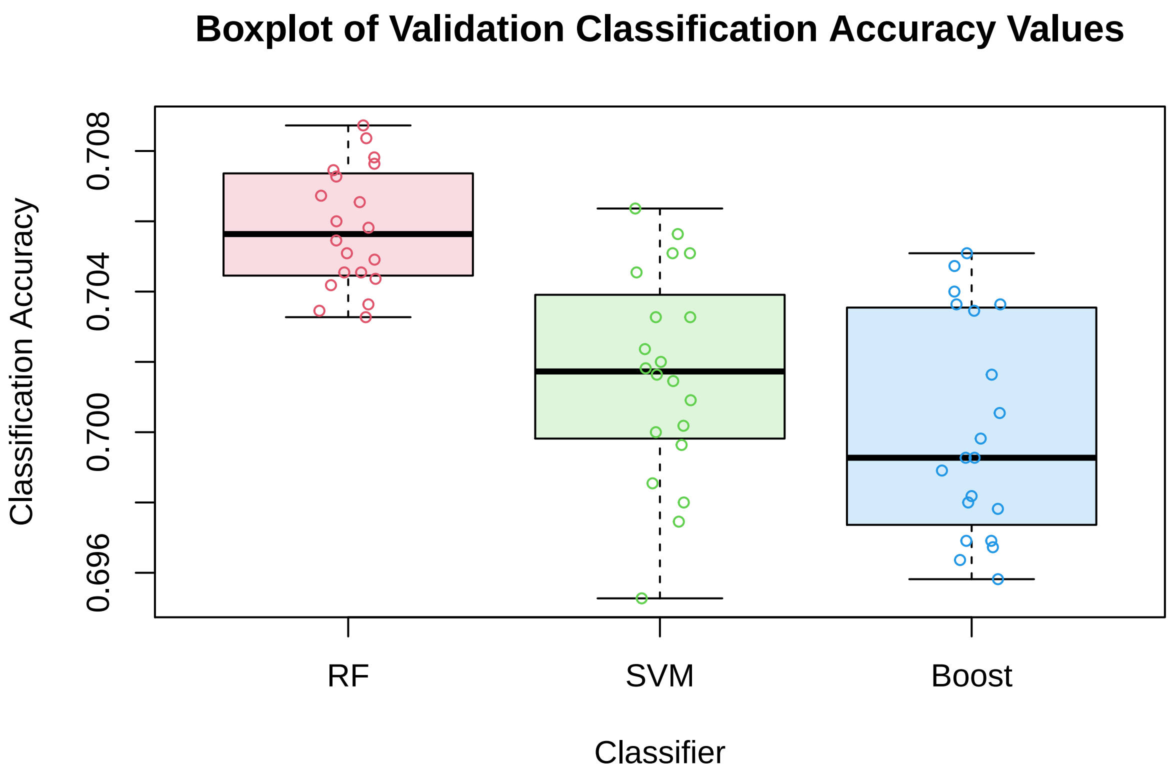
After completing the cross-validation, the average accuracy across the folds for each replicate can be calculated for each of the classifiers by applying the *colMeans* function to the output vector list objects returned by the cross-validation code, and the overall mean validation accuracy obtained by each model can also be calculated by taking the mean of all of the accuracy values over all replicates.

```
# get mean accuracy across all folds for all replicates (for all SVM models)
avg_fold_acc_svm <- t(sapply(out_svm, colMeans) )
# get mean overall accuracy across all replicates (for all SVM models)
avg_acc_svm <- mean(avg_fold_acc_svm)

# get mean accuracy across all folds for all replicates (for random forest model)
avg_fold_acc_rf <- t(sapply(out_rf, colMeans) )
# get mean overall accuracy across all replicates (for random forest model)
avg_acc_rf <- mean(avg_fold_acc_rf)

# get mean accuracy across all folds for all replicates (for boosting model)
avg_fold_acc_boost <- t(sapply(out_boost, colMeans) )
# get mean overall accuracy across all replicates (for boosting model)
```

```
avg_acc_boost <- mean(avg_fold_acc_boost)
```

To visualise the validation performance obtained by each of the classifiers, I have produced the boxplot below which shows the distribution of accuracy values obtained by each model over each of the cross-validation replicates. The overall average classification accuracy obtained by each classifier is also printed below.

```
# boxplot of average cv accuracy for each replicate, for all 3 models
boxplot(t(rbind(avg_fold_acc_rf, avg_fold_acc_svm, avg_fold_acc_boost)),
        col=adjustcolor(c(2:4), 0.2), names=c("RF", "SVM", "Boost"), ylab="Classification Accuracy", xla
stripchart(data.frame(cbind(t(avg_fold_acc_rf), t(avg_fold_acc_svm), t(avg_fold_acc_boost))),
           add=T, vertical = T, method = 'jitter', pch=21, cex=0.7, col=2:4)
```



```
# print overall mean cv accuracy for each model
cat("Average Cross-Validation Accuracy (Random Forest):", avg_acc_rf,
    "\nAverage Cross-Validation Accuracy (SVM):", avg_acc_svm,
    "\nAverage Cross-Validation Accuracy (Boosting):", avg_acc_boost)
```

```
## Average Cross-Validation Accuracy (Random Forest): 0.7057909
## Average Cross-Validation Accuracy (SVM): 0.7016273
## Average Cross-Validation Accuracy (Boosting): 0.7000364
```

While none of the classifiers performs significantly better than the other, the random forest classifier appears to perform the best overall. It achieved the best overall average validation accuracy, and its performance was relatively consistent across each of the replicates based off the distribution of accuracy values shown on the boxplot, so I will choose this random forest classifier as my best model.

(c) Evaluate the predictive classification performance of the best model you find. Provide a discussion about the ability of the selected model at detecting correctly negative and positive reviews.

Having chosen a random forest classifier with the number of randomly selected predictors used in each tree equal to 5 and number of trees equal to 1000 as the best model, we can now evaluate its predictive performance using the test data set aside earlier.

Firstly, we can refit the random forest classifier using the full set of training and validation data. After fitting this final model, we can use it to predict the classes of the reviews in test data set, and finally calculate the obtained classification accuracy. The classification accuracy of the test data is printed below.

```r
# fit random forest classifier using the full set of training/validation data
fit_rf_final <- randomForest(class ~ ., data=reviews_trainval,
                       mtry=fit_rf_cv$bestTune$mtry, ntree=fit_rf_cv$bestTune$ntree)

# predict classes of current validation set
pred_test <- predict(fit_rf_final, newdata=reviews_test, type="class")

# calculate classification accuracy of the validation data
acc_final <- class_acc(pred_test, reviews_test[,ncol(reviews_test)])
```

```r
cat("Classification Accuracy of Test Data (Random Forest):", acc_final)
```

```
## Classification Accuracy of Test Data (Random Forest): 0.7016012
```

The model achieves a similar classification performance on the test data as it achieved during validation, correctly classifying just over 70% of the reviews. This is reasonably satisfactory considering none of the models could achieve classification accuracy values significantly greater than 70% during cross-validation.

We can also plot a confusion matrix comparing the predicted and actual classes for the test data, to show the number of reviews that were correctly and incorrectly classified for each class.

```r
# load libraries for kable tables
library("knitr"); library("kableExtra")

# create confusion matrix for the predicted test data
tab <- table(reviews_test[,ncol(reviews_test)], pred_test)

# add column to table showing class accuracy for each individual class
tab_classacc <- cbind(tab, class.accuracy=round(diag(tab)/rowSums(tab),3))

# print table of actual vs predicted classes and class accuracy for each class
add_header_above(
  kable_styling(
    kable(tab_classacc, align = 'c')),
  c("Actual"=1, "Predicted"=2, " "=1))
```

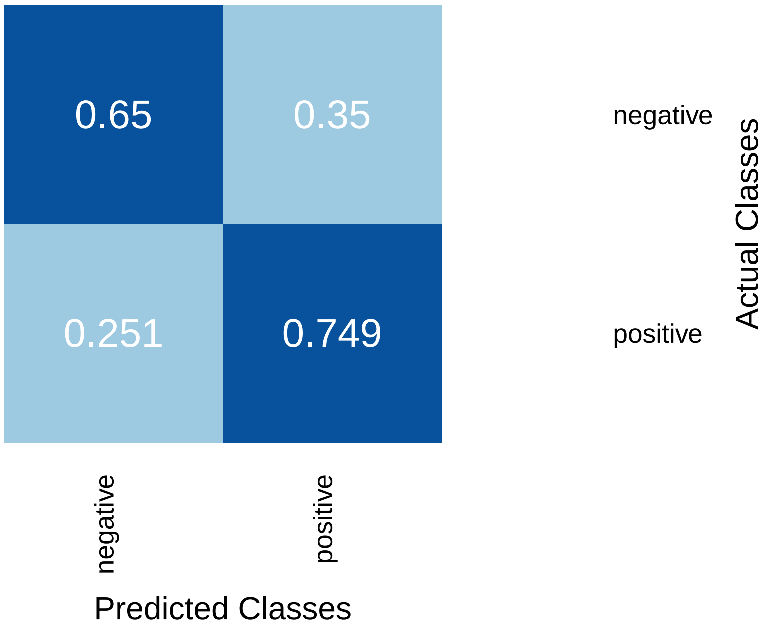
Actual	Predicted		
	negative	positive	class.accuracy
negative	425	229	0.650
positive	181	539	0.749

The *class.accuracy* column added to the table shows the classification accuracy achieved for both the negative and positive reviews (i.e. true negative rate or specificity, and true positive rate or sensitivity) and there is a noticeable difference between these. Although the classes in our test data set are imbalanced (720 positive reviews vs 654 negative reviews), the classification accuracy for each class is an indicator that the model has a greater sensitivity that specificity, meaning that it is performs better at classifying reviews that are actually positive than it does at classifying reviews that are actually negative.

A heatmap can also be a useful way of visualising the proportion of correctly and incorrectly predicted classes for the test data. This effectively shows (from top-left to bottom-right) the True Negative rate, False Positive rate, False Negative rate and True Positive rate obtained for the test data, with colour added to easily identify the different magnitudes of the rate values. As explained above, we can see from this plot that the model achieved different True Positive rate and True Negative rate values, and as a result the False Positive rate and False Negative rate values are also different.

Overall the model still performs relatively well, but the imbalance in classification performance of negative and positive reviews is of slight concern.

```
# load library
library(gplots)

# add heatmap to visualise proportion of correct/incorrect predictions for each class
heatmap.2(tab, Rowv = NA, Colv = NA, cexRow = 1, cexCol = 1, dendrogram = "none",
          scale = "row", trace = "none", asp=1, key=FALSE,
        xlab = "Predicted Classes", ylab = "Actual Classes",
        cellnote = round(tab/rowSums(tab),3), notecex = 1.5, notecol = "white",
        margins = c(6,6), col = blues9[4:8])
```


negative
positive
Actual Classes
negative
positive
Predicted Classes