Question 5

1 - Introduction

In this project, we have been asked to utilise some popular supervised classification methods to create a model that identifies whether a given review from Rotten Tomatoes is "negative" or "positive".

We have considered the following classifiers: Support Vector Machines with a Gaussian Radial Basis Function kernel (SVM), Random Forests (RF), Logistic Regression (Logistic) and Classification Trees (CT). Note that there are other classifiers which we could have considered, however for practical reasons we have decided to limit our analysis to these four classifiers. All future references to the "optimal classifier" refers to the optimal classifier among the four considered.

2 - Data

Our dataset contains 6874 movie reviews extracted from the review aggregator website Rotten Tomatoes. For each review, a number of numerical features have been constructed, which are related to the frequency of certain characters, frequencies of certain words, sentiment scores, and coordinates in a latent embedding representation of the text. Each review has an associated class label, denoting whether the review was negative or positive. Note that we have removed the first column of the dataset as it contains the raw review text, which is not useful for our analysis.

The numerical features are all of similar magnitude, so we did not scale the numerical data in advance of our analysis.

We have split our dataset into training/validation data and test data. We have chosen to allocate 60% of our data to model training/validation and the remainder to testing. Generally, it is good to have a large quantity of data in the training/validation phase so that the classifier's learned parameters have low associated variance. It is also advisable to have a large quantity of data in the testing phase in order to ensure that an accurate estimate of the generalisation accuracy can be obtained. We feel as though a 60/40 split facilitates both of these objectives.

We have used set.seed() in R so that our training/validation data is consistent across all classifiers. This ensures that we are comparing the classifiers fairly.

```
options(scipen = 999)

# Load in packages
library(caret)
library(glmnet)
library(doParallel)

# Set up data
dat0 = read.csv("data_rotten_tomatoes_review.csv", header = T)
dat0 = dat0[, -1]
dat0$class = factor(dat0$class)

# Split the data into training/validation and testing
train_val <- createDataPartition(dat0$class, p = 0.6, list = FALSE)
dat = dat0[train_val,]
dat_test = dat0[-train_val,]
```

3 - Method

We have used accuracy as the measure of predictive performance for each classifier. This measure represents the proportion of observations that were correctly classified. In Section 6, we also consider sensitivity and specificity, which represent the proportion of true positives and true negatives correctly classified, respectively. Validation accuracy is used to compare the classifiers (the classifier with the highest validation accuracy is chosen as the optimal classifier) and test accuracy is used to obtain an estimate of the generalisation accuracy after the optimal classifier has been selected (estimates how our classifier should perform on unseen data). Prior to the computation of the test accuracy, our optimal classifier is refitted using both the training and the validation data.

For each classifier, we have tuned the associated hyperparameters. Hyperparameter tuning is important, as it enables us to obtain a classifier that is flexible enough to capture patterns in our data without incurring overfitting. Hyperparameter tuning involves creating a discrete set of hyperparameters and creating classifiers with all possible combinations of hyperparameters in the specified set. The predictive performance of these classifiers is tested on the validation data, and the classifier with the best predictive performance is considered to have the optimal hyperparameters.

Note that the chosen hyperparameters were optimal among those considered, so it is likely that we have not selected the globally optimal parameters. This is a constraint of our chosen tuning procedure. It is not advisable to specify too many hyperparameters as this will increase the runtime of the algorithm. However, if we do not consider a sufficient number of hyperparameters, we run a greater risk of selecting sub-optimal hyperparameters. There is a balance to be struck when tuning, and we have constructed the hyperparameter sets with this balance in mind. It is generally considered good practice to re-tune with higher/lower hyperparameters in the event where the optimal hyperparameters are the highest/lowest of those considered. We have neglected to do this in instances where the predictive performance is not overly sensitive to changes in hyperparameters.

We have tuned the hyperparameters and selected the optimal classifier using a k-fold cross validation procedure with 2 folds. This procedure involves dividing the dataset into 2 folds, using 1 fold to fit the classifier and testing the predictive performance on the remaining fold. The procedure is then repeated with the folds switched. We feel as though this method is most appropriate as in this specific application it is not overly computationally intensive and estimates of predictive performance from k-fold cross validation tend to be more accurate than those derived from hold-out sampling. We would consider leave-one-out sampling to be inappropriate for this application due to the size of our dataset (the leave-one-out approach is not computationally efficient when our dataset is large). We have set the number of iterations to 5 for each classifier. This means that we will be running the k-fold cross validation procedure 5 times, and the final validation accuracy will be the average validation accuracy from the 5 iterations. We feel as though 5 iterations is sufficient to account for the sampling variability of the process.

We have used parallel computing to decrease the runtime of our algorithms.

4 - Analysis

4.1 - SVM

The first classifier we considered was an SVM classifier. This classifier took a long time to tune (approximately 6 minutes in R). However, the SVM approach has some advantages, namely, it is adept at performing non-linear classification.

SVM classifiers take a C parameter and a sigma parameter as inputs. The C parameter gives the number of observations which are allowed to be on the wrong side of the margin (the larger the C, the more tolerant we are for violations of the margin). The sigma parameter controls the flexibility of the classifier (having a high sigma implies that the SVM will be better able to capture local non-linear patterns in the data). We have chosen [1, 10, 100, 500] as our set of possible C values and [0.00001, 0.001, 0.1, 5] as our set of possible

sigma values. Adding more hyperparameters would increase the runtime of our algorithm to an unreasonable extent.

```
# Set up parallel computing

cl <- makeCluster(2)
registerDoParallel(cl)

# Set up k-fold cross validation (k = 2, iterations = 5)
train_ctrl <- trainControl(method = "repeatedcv", number = 2, repeats = 5)


################### 1 - SVM with GRBF kernel ##############################
#
# set grid
tune_grid <- expand.grid(C = c(1, 10, 100, 500),
                         sigma = c(0.00001, 0.001, 0.1, 5))
#
set.seed(888)

fit_svm_grbf <- train(class ~ ., data = dat,
                      method = "svmRadial",
                      trControl = train_ctrl,
                      tuneGrid = tune_grid)

fit_svm_grbf$results
```

```
##       C   sigma  Accuracy      Kappa  AccuracySD    KappaSD
## 1     1 0.00001 0.5239942 0.00000000 0.000000000 0.00000000
## 5    10 0.00001 0.5591857 0.07900011 0.008318312 0.01874471
## 9   100 0.00001 0.6818226 0.35913956 0.009380376 0.01842241
## 13 500 0.00001 0.6799321 0.35650331 0.011413902 0.02262012
## 2     1 0.00100 0.6805623 0.35644397 0.009382602 0.01843997
## 6    10 0.00100 0.6791566 0.35522939 0.010868860 0.02163844
## 10 100 0.00100 0.6632574 0.32394039 0.008999273 0.01812111
## 14 500 0.00100 0.6264178 0.25059333 0.006976123 0.01378764
## 3     1 0.10000 0.5384392 0.03339705 0.003012447 0.00639372
## 7    10 0.10000 0.5505574 0.06171758 0.005154281 0.01091265
## 11 100 0.10000 0.5505574 0.06171758 0.005154281 0.01091265
## 15 500 0.10000 0.5505574 0.06171758 0.005154281 0.01091265
## 4     1 5.00000 0.5239942 0.00000000 0.000000000 0.00000000
## 8    10 5.00000 0.5239942 0.00000000 0.000000000 0.00000000
## 12 100 5.00000 0.5239942 0.00000000 0.000000000 0.00000000
## 16 500 5.00000 0.5239942 0.00000000 0.000000000 0.00000000
```

```
fit_svm_grbf$bestTune
```

```
##     sigma   C
## 9 0.00001 100
```

Results The classifier with (sigma, C) = (0.00001, 100) had the highest accuracy. Hence, this is our optimal SVM classifier.

4.2 - RF

The second classifier we considered was a RF classifier. This classifier also took along time to tune (approximately 7 minutes in R). However, it has the advantage of being an ensemble method. Ensemble methods aggregate predictions made by multiple classifiers, which tends to improve accuracy. The use of only a subset of predictor variables is also an advantage that is specific to the RF method as the classifiers behave more independently than if we build them on all of the data.

RF classifiers take a mtry parameter as an input. This parameter controls the number of predictors used to construct the classifiers. We have chosen the set [1, 10, 20, 50, 79] as the set of possible parameters (79 is the maximum we can have as there are 79 predictors in our dataset). Similar to the SVM case, we consider the addition of more parameters to be unnecessary due to the resultant increase in model runtime.

```
################## 2 - Random Forest ########################################
#
# set grid
tune_grid <- expand.grid( mtry = c(1, 10, 20, 50, 79) )

#
set.seed(888)

fit_rf <- train(class ~ ., data = dat,
                method = "rf",
                trControl = train_ctrl,
                tuneGrid = tune_grid)

fit_rf$results
```

```
##   mtry  Accuracy     Kappa  AccuracySD    KappaSD
## 1    1 0.6762482 0.3457703 0.008287655 0.01721458
## 2   10 0.6879787 0.3738421 0.006693462 0.01277839
## 3   20 0.6848764 0.3677888 0.008833549 0.01694059
## 4   50 0.6825982 0.3636500 0.009125310 0.01770321
## 5   79 0.6804653 0.3594359 0.009340212 0.01805433
```

```
fit_rf$bestTune
```

```
##   mtry
## 2   10
```

Results The classifier with mtry = 10 had the highest accuracy. Hence, this is our optimal RF classifier.

4.3 - Logistic

The third classifier we considered was a logistic classifier. The advantage of this classifier is that it is quick to run and performs well when the data is linearly separable.

Logistic classifiers take a lambda parameter and an alpha parameter as inputs. Typical usage is for the program to construct the lambda and alpha values on its own, so we have not specified lambda and alpha values in advance.

```
################# 3 - Logistic ################################################
#
set.seed(888)

fit_logistic <- train(class ~ ., data = dat,
                      method = "glmnet",
                      trControl = train_ctrl)

fit_logistic$results
```

```
##   alpha       lambda  Accuracy     Kappa  AccuracySD     KappaSD
## 1  0.10 0.0003787958 0.6782356 0.3536682 0.011422476 0.022857795
## 2  0.10 0.0037879577 0.6785264 0.3542102 0.011472649 0.022958340
## 3  0.10 0.0378795774 0.6846340 0.3659028 0.011338515 0.022581780
## 4  0.55 0.0003787958 0.6781386 0.3534729 0.011377132 0.022783561
## 5  0.55 0.0037879577 0.6795928 0.3562736 0.011317312 0.022668419
## 6  0.55 0.0378795774 0.6846340 0.3648992 0.004611035 0.009130745
## 7  1.00 0.0003787958 0.6776539 0.3524866 0.011584067 0.023216840
## 8  1.00 0.0037879577 0.6823073 0.3616531 0.011952766 0.023850673
## 9  1.00 0.0378795774 0.6809016 0.3578022 0.006677842 0.014183220
```

```
fit_logistic$bestTune
```

```
##   alpha     lambda
## 3   0.1 0.03787958
```

Results The classifier with (alpha, lambda) = 0.1, 0.0379 had the highest accuracy. Hence, this is our optimal logistic classifier.

4.4 - CT

The final classifier we considered was a CT classifier. This classifier is very quick to run in R, and as a result we were able to specify a large number of hyperparameters during the tuning process. Another advantage is that classification trees are explainable and easy to interpret. A drawback is that the trees suffer from high variance, so we would not expect this approach to outperform RF.

CT classifiers take a cp parameter as an input. This parameter controls the complexity of the classification trees. A low cp value implies that the tree can have a large number of splits and vice versa. We have considered the following cp parameters: [0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5]. This is a large number of hyperparameters. Consequently, we do not consider it necessary to add more.

```
################# 4 - Classification Trees ###########################
#
# set grid
tune_grid <- expand.grid( cp = c(0.0001, 0.001, 0.01, 0.05, 0.1, 0.2, 0.3, 0.4,
                                 0.5) )

#
set.seed(888)

fit_ct <- train(class ~ ., data = dat,
```

```
                method = "rpart",
                trControl = train_ctrl,
                tuneGrid = tune_grid)

fit_ct$results
```

```
##       cp  Accuracy     Kappa  AccuracySD    KappaSD
## 1 0.0001 0.6114397 0.2202936 0.017142702 0.03400220
## 2 0.0010 0.6174988 0.2321868 0.014965641 0.02968924
## 3 0.0100 0.6714978 0.3410747 0.007547738 0.01504753
## 4 0.0500 0.6698497 0.3403472 0.006776417 0.01121170
## 5 0.1000 0.6698497 0.3403472 0.006776417 0.01121170
## 6 0.2000 0.6698497 0.3403472 0.006776417 0.01121170
## 7 0.3000 0.6698497 0.3403472 0.006776417 0.01121170
## 8 0.4000 0.5239942 0.0000000 0.000000000 0.00000000
## 9 0.5000 0.5239942 0.0000000 0.000000000 0.00000000
```

```
fit_ct$bestTune
```

```
##     cp
## 3 0.01
```

```
# stop the parallel computing procedure
stopCluster(cl)
registerDoSEQ()
```

Results The classifier with cp = 0.01 had the highest accuracy. Hence, this is our optimal CT classifier.

4.5 - Classifier Comparison

In this stage, we have compared each of the four optimal classifiers from sections 4.1 - 4.4. The classifier with the highest average validation accuracy out of the four considered was chosen as our optimal classifier.

```
# Compare the classifiers
candidates = list(SVM = fit_svm_grbf, RF = fit_rf,
                      Logistic = fit_logistic, CT = fit_ct)
comp <- resamples(candidates)
summary(comp)
```

```
##
## Call:
## summary.resamples(object = comp)
##
## Models: SVM, RF, Logistic, CT
## Number of resamples: 10
##
## Accuracy
##               Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## SVM      0.6684440 0.6776539 0.6825012 0.6818226 0.6862579 0.6994668    0
## RF       0.6761997 0.6850460 0.6875909 0.6879787 0.6926806 0.6984973    0
```

```
## Logistic 0.6698982 0.6759573 0.6839554 0.6846340 0.6932865 0.7033446    0
## CT       0.6635967 0.6660204 0.6689287 0.6714978 0.6774115 0.6844401    0
##
## Kappa
##               Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## SVM      0.3338665 0.3502130 0.3590838 0.3591396 0.3685726 0.3938719    0
## RF       0.3515094 0.3696449 0.3729059 0.3738421 0.3820439 0.3943602    0
## Logistic 0.3361366 0.3484743 0.3642713 0.3659028 0.3833658 0.4033723    0
## CT       0.3245696 0.3309199 0.3352511 0.3410747 0.3493804 0.3678657    0
```

5 - Analysis of Optimal Classifier

```
best_to_worst = sort(comp, decreasing = TRUE)
best = best_to_worst[1]

switch(best,
       Logistic = {
         class_hat_logistic <- predict(fit_logistic, newdata = dat_test)
         cm = confusionMatrix(class_hat_logistic, dat_test$class,
                              positive = "positive")
       },
       RF = {
         class_hat_rf <- predict(fit_rf, newdata = dat_test)
         cm = confusionMatrix(class_hat_rf, dat_test$class,
                              positive = "positive")
       },
       SVM = {
         class_hat_grbf <- predict(fit_svm_grbf, newdata = dat_test)
         cm = confusionMatrix(class_hat_grbf, dat_test$class,
                              positive = "positive")
       },
       CT = {
         class_hat_ct <- predict(fit_ct, newdata = dat_test)
         cm = confusionMatrix(class_hat_ct, dat_test$class,
                              positive = "positive")
       }
)

# Accuracy - Proportion of observations classified correctly
accuracy = round(cm$overall[1], 4) * 100

# Sensitivity - Proportion of true positives correctly identified as such
sens = round(cm$byClass[1], 4)

# Specificity - Proportion of true negatives correctly identified as such
spec = round(cm$byClass[2], 4)

sens_spec = c(sens, spec)
neg_pos = c("positive", "negative")
better = neg_pos[which.max(sens_spec)]
worse = neg_pos[which.min(sens_spec)]
```

Our analysis from section 4.5 suggests that the RF classifier had the highest average validation accuracy. Hence, it is the optimal classifier for predicting whether a given Rotten Tomatoes review is "negative" or "positive".

Our optimal model correctly predicted the sentiment of the review 72.71% of the time when it was tested on the test dataset, implying that predictive performance was adequate. There are numerous ways to improve this accuracy figure which, for practical purposes, we have neglected to implement (increase the number of hyperparameters considered, increase the quantity of data used to train the classifiers, consider alternative classification methods, consider alternative predictors etc.).

The sensitivity of our optimal classifier (the proportion of true positives correctly identified as such) is 0.7451.

The specificity of our optimal classifier (the proportion of true negatives correctly identified as such) is 0.7072.

These results suggest that our classifier can correctly identify positive reviews better than negative reviews.