STAT40750 – Statistical Machine Learning (Online) - Assignment 3

Daniel Williams 21203054

18/04/2022

Hate Speech Detection

This markdown doccument addresses all sections of the task, 1,2 & 3.
The structure is as follows:.

Setting up the data, splitting into training data & test data.
Logistic regression, validated with k fold cross validation.
Logistic regression, with tuning via glmnet.
Both of these models are tested on the test data set.
.
Classification tree, validated with k fold cross validation.
Classification tree, tuned via CP variable.
Again, both of these are tested on the test data set.
.
Random forest, validation with k fold cross validation.
Random forest, tuned via mtry variable.
Both tested on test data set.
.
I will add commentary both within markdown, and inbedded into code chunks. .
Firstly loading the data, and ensuring that the output variable is coded as a factor rather than a string variable.

```
library(readr)
data_hate_speech <- read_csv(file = "data_hate_speech.csv")
data_hate_speech$class <- as.factor(data_hate_speech$class)
```

The next chunk is all about setting up ready for analysis, commentary added into the chunk.

```
set.seed(123) # repeatable results
hate <- data_hate_speech[,-c(1,15)]
# removing number of punctuation marks, this was causing issues with the fitting
# removing the text string itself, as we are unable to fit using this column, it is not numer
ic, nor a categorical variable.

# Below is a small chunk splitting the data, 80% for use in model training, and 20% reserved
 for testing the models predictive performance

N <- nrow(data_hate_speech)
test <- sample(1:N, N*0.2)
train <- setdiff(1:N, test)
data <- hate[train,]
data_test <- hate[test,]
N_train <- nrow(data)

# defining a custom function to be used throughout the analysis to quickly assess the accurac
y of a classifier

class_acc <- function(y, yhat) {
  tab <- table(y, yhat)
  return( sum(diag(tab))/sum(tab) )
}

K <- 4  # number of folds within the data
R <- 20 # number of repeats used in the logistic regression & classification trees
M <- 10 # number of repeats used in the random forests (lower due to computing speed)
out <- vector("list", R)   # three empty vectors for use in k fold cross validation for loops
out_ct <- vector("list", R )
out_rf <- vector("list", M)
```

Logistic Regression

For loop, performing a k fold cross validation of a logistic regression model fitted with the aim of predicting the presence of hate speech. K fold cross validation splits the data into “folds”, where one fold is dropped, the model is fitted on the remaining data, and then the model is tested on the “dropped fold”, within this loop that test result is then stored as an accuracy rating for the classifier. We effectively split the training data into training data and validation data, and randomly resample these to increase our knowledge of the models performance and generalisation error. The resampling allows us to account for the variable nature of a single sample of data. This approach will give us a validation error, which will act as our surrogate for the test error, and thus we will make our decision of which classifier is performing the best based on this.

```
for ( r in 1:R ) { # 20 repeats of the cross validation process
  acc <- matrix(NA, K, 1)  # accuracy vector to be used later in the loop
  folds <- rep( 1:K, ceiling(N/K) ) # three lines setting the fold proceedure
  folds <- sample(folds)
  folds <- folds[1:N_train]
  for ( k in 1:K ) { # looping the fitting and testing process through the folds
    train_fold <- which(folds != k)  # dropping a fold, fitting, validating, repeat
    validation <- setdiff(1:N_train, train_fold)

    # inbuilt logistic regression function
    fit_log_1 <- glm(class ~ ., data = data, family = "binomial", subset = train_fold)

    pred_log_1 <- predict(fit_log_1, type = "response", newdata = data[validation,])
    # testing the fitted model on the dropped fold
    pred_log_1 <- ifelse(pred_log_1 > 0.5, 1, 0)
    # tau set to 0.5, threshold value for predicting a 1, in this case meaning a hate result.
    acc[k,1] <- class_acc(pred_log_1, data$class[validation])
    #storing the accuracy result for this fold, then repeating for each k (4 times total)
  }
  # looping through this process 20 times and storing in a 20 x 4 output matrix
  out[[r]] <- acc
}


avg_per_rep <- sapply(out, colMeans)  # column means, averaging out the 4 fold results
mean_acc_lr_1 <- mean(avg_per_rep) # overall mean giving me overall average accuracy
mean_acc_lr_1
```

```
## [1] 0.9996671
```

```
stdev_lr_1 <- sd(avg_per_rep)
stdev_lr_1
```

```
## [1] 0.0001158936
```

Accuracy of 0.999, a very good classifier has been produced.
Very low standard deviation, 0.0001, so the classifier is also robust.

We see warning messages that the glm algorithm did not always converge, and that probabilities of 0 or 1 have occurred. This is a complete separation problem. Effectively we have very good predictor variables, which are able to perfectly predict whether a text string has hate speech within it. I have used all of the numerical variables, except number of punctuation marks, within the fitting process. I could have dropped a few of these columns, which would have resulted in perhaps a simpler model to understand & explain verbally, but may also reduce accuracy. This would be an interesting balance, however for the purpose of this assignment I will continue with the use of the great majority of the indicator columns. This complete separation problem does cause an issue in terms of infering information about which coefficients are most important, but doesn't really affect our ability to accuratly make classification decisions as we see with the average accuracy.

.

.

```
lr_non_tuned <- glm(class~., data = data, family = "binomial")
# fitting the logisitc regression to all of the training data
tau <- 0.5  # setting the threshold for p value to predict a "1" = Hate
p <- fitted(lr_non_tuned)
pred <- ifelse(p > tau, 1, 0) # setting a 1 if the p value exceeds threshold tau
y <- ifelse(data$class == "hate", 1, 0)
table(y, pred)
```

```
##    pred
## y      0    1
##   0    0 7608
##   1  955    0
```

Result is aligned to the k fold cross validation. The logistic regression classifier is perfect at predicting the class of the training data.

.

Below looking into the coefficients of each indicator variable. This gives us an indication of quite how complex the model is, and will be compared to the tuned version later on.

.

```
coef(lr_non_tuned)
```

```
##    (Intercept)         n_urls        n_chars     n_uq_chars       n_commas
##   -49.92215873     1.33415432     0.12037740    -1.08406974     0.37691238
##       n_digits     n_exclaims  n_extraspaces       n_lowers      n_periods
##     0.55556451    -0.57775005     1.65806102     0.02299789    -0.13179154
##        n_words     n_uq_words         n_caps n_charsperword n_prepositions
##    -0.16935504    -0.42482836     0.03037689     0.72371098    -0.79162358
##             w1             w2             w3             w4             w5
##    14.20201958    58.52671534    60.98850977    44.46098611    51.50183971
##             w6             w7             w8             w9            w10
##    48.75069564    51.40957103    18.62165516    52.81026509    35.73927374
##            w11            w12            w13            w14            w15
##    41.15510265    30.89698432     9.79119569    41.51059094    72.38383678
##            w16            w17            w18            w19            w20
##    41.56946774    26.23036859    43.25890344    40.49848961    23.95976797
##   sent_syuzhet      sent_bing     sent_afinn       sent_nrc
##    -1.59320439     0.29702750     0.32010462     0.49695446
```

Below, testing the logistic regression model on the test data.

.

```
pred_lr_non_tuned <- predict(lr_non_tuned, type = "response", newdata = data_test)
pred_lr_non_tuned <- ifelse(pred_lr_non_tuned > 0.5, 1, 0)
test_performance_lr_non_tuned <- class_acc(pred_lr_non_tuned, data_test$class)
test_performance_lr_non_tuned
```

```
## [1] 0.9995327
```

The logistic regression classifier performs excellently on the test data.
The test data was not used in the fitting process, so this is a encouraging result.
We could further look to improve the model by varying tau from 0.5, but this is not really required here.

Additionally we could look at optimising the area under the ROC curve, but again with such a good result, we will simply perform a lasso regression, to simplify the model instead.

.

.

Realistically, with this level of performance, we do not need to tune the logistic regression model to improve its performance. But we can look to remove some variables which perhaps do not add enough value to the model (in terms of performance), but do significantly add to its complexity. We will assess this by looking at the lasso method, which provides a penalisation for coefficients of small magnitude. It will help to find the optimum position between model complexity, and predictive performance.

.

```
library(caret)
library(glmnet)
library(tidyverse)

x <- model.matrix(class~., data)[,-1]
# dropping the class column and setting the predictor columns
y <- ifelse(data$class == "hate", 1, 0)
# coding hate as a 1, and no_hate as a zero for the purpose of fitting

cv.lasso <- cv.glmnet(x, y, alpha = 1, family = "binomial")
# alpha set to 1 for lasso regression
# cv.glmnet calculating our lambda value for use in fitting in the next line
model <- glmnet(x, y, alpha = 1, family = "binomial", lambda = cv.lasso$lambda.min)
coef(model)
```

```
## 39 x 1 sparse Matrix of class "dgCMatrix"
##                          s0
## (Intercept)     12.41995619
## n_urls           .
## n_chars          .
## n_uq_chars       0.15424214
## n_commas         .
## n_digits        -0.00668808
## n_exclaims       .
## n_extraspaces    .
## n_lowers         .
## n_periods        .
## n_words          .
## n_uq_words       .
## n_caps           .
## n_charsperword   .
## n_prepositions   .
## w1               -4.27486816
## w2              -10.57366291
## w3              -14.12211105
## w4              -11.75515486
## w5               -7.87748937
## w6              -14.35744416
## w7              -11.74345261
## w8               -5.38638877
## w9               -7.51455778
## w10              -7.31243171
## w11             -10.24245921
## w12              -9.58197312
## w13              -7.15366577
## w14              -9.02716387
## w15             -12.31870281
## w16             -11.48511978
## w17              -6.24982899
## w18              -1.93654572
## w19             -12.15681121
## w20              -6.50853629
## sent_syuzhet     .
## sent_bing        .
## sent_afinn       .
## sent_nrc         .
```

We see the result that many of the variables have had their coefficient removed from the lasso regression process. The result is a simpler model, with less predictor variables.

.

.

Below testing the tuned logistic regression model on the test data.

```
x.test <- model.matrix(class~., data_test)[,-1]  # dropping class
probabilities <- model %>% predict(newx = x.test)
# p values calculated for the likelihood of a 1 = Hate for the test data
predicted.classes <- ifelse(probabilities > 0.5, "hate", "no_hate")
# again retaining tau as 0.5 as the threshold level
observed.classes <- data_test$class # observed data from the original split
head(cbind(predicted.classes, observed.classes)) # reviewing the first results
```

```
##   s0        observed.classes
## 1 "no_hate" "2"
## 2 "no_hate" "2"
## 3 "no_hate" "2"
## 4 "no_hate" "2"
## 5 "no_hate" "2"
## 6 "no_hate" "2"
```

```
mean(predicted.classes == observed.classes) # average level of agreement
```

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

Perfect prediction achieved from the tuned logistic regression model.
This is a positive result, and it has the added benefit of having less factors therefore a simpler model to explain & compute.
.

Classification Tree

.
Now moving on to the second supervised learned model, classification tree.
Employing a very similar for loop strategy, therefore I will omit any comments which were clarified above.

```
library(rpart)
library(partykit)

for ( r in 1:R ) {
  acc <- matrix(NA, K, 1)
  folds <- rep( 1:K, ceiling(N/K) )
  folds <- sample(folds)
  folds <- folds[1:N_train]
  for ( k in 1:K ) {
    train_fold <- which(folds != k)
    validation <- setdiff(1:N_train, train_fold)
    # rpart fitting of a classification tree ilo the glm function used for logistic regressio
n.
    fit_ct <- rpart(class~ ., data = data, subset = train_fold, control = list(cp = 0.05))

    pred_ct_1 <- predict(fit_ct, type = "class", newdata = data[validation,])
    acc[k,1] <- class_acc(pred_ct_1, data$class[validation])
  }

  out_ct[[r]] <- acc
}
avg_per_rep_ct <- sapply(out_ct, colMeans)
mean_acc_ct_1 <- mean(avg_per_rep_ct)
mean_acc_ct_1
```

```
## [1] 0.9773159
```

```
stdev_ct_1 <- sd(avg_per_rep_ct)
stdev_ct_1
```

```
## [1] 0.002001638
```

.

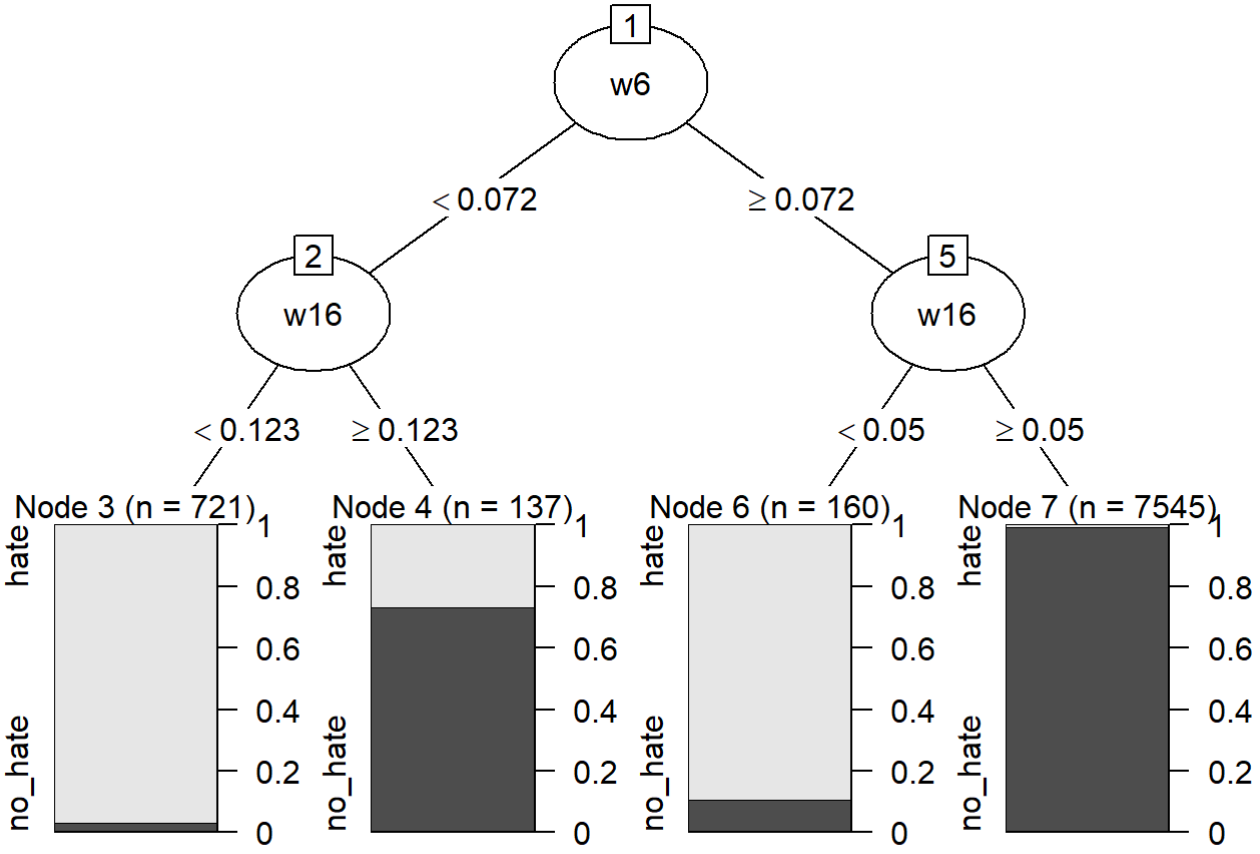
High mean, 0.9777, but not quite to the level which was achieved by both the un-tuned, and tuned logistic regression models.

.

Similarly for the standard deviation, realistically in most scenarios we would be delighted by this result, but it remains not as good as the result achieved by logistic regression.

.

```
classif_tree_all <- rpart(class~., data = data, control = list(cp=0.05))
# fitting a tree to all of the training data
plot( as.party(classif_tree_all), cex = 0.5)
```




```
# graphical view of the tree
phat <- predict(classif_tree_all, type = "class")
# phat a vector of p values for a 1 = Hate, result, type class gives 1,0 results
table(phat, data$class)
```

```
##
## phat      hate no_hate
##   hate     842      39
##   no_hate  113    7569
```

We see both graphically, and in table form the performance of the fitted classification trees on the training data in its entirety. Again we see that the classification tree is a very well performing classifier, but does not reach the standard set by the logistic regression. CP controls the complexity of the structure of the tree. I have set it at 0.05 here, but below I vary this, looking to tune the tree performance and improve.

.

.

```
# testing on actual test data
pred_ct <- predict(classif_tree_all, type = "class", newdata = data_test)
class_acc(pred_ct, data_test$class)
```

```
## [1] 0.978972
```

```
# altering the CP, and retesting
classif_tree_all2 <- rpart(class~., data = data, control = list(cp=0.01))
pred_ct2 <- predict(classif_tree_all2, type = "class", newdata = data_test)
class_acc(pred_ct2, data_test$class)
```

```
## [1] 0.9859813
```

```
# altering the CP and retesting
classif_tree_all3 <- rpart(class~., data = data, control = list(cp=0.005))
pred_ct3 <- predict(classif_tree_all3, type = "class", newdata = data_test)
class_acc(pred_ct3, data_test$class)
```

```
## [1] 0.9897196
```

We see the effect of varying the CP, resulting with each change, with a more complex tree. We see that the performance does increase, but less than 1% for each move, and considering we already have a high performing classifier, it does not seem prudent to keep increasing the complexity of the tree.

.

.

Random Forest

Deploying a random forest approach, with k fold cross validation, again omitting duplicated comments. .

```
library(randomForest)
M <- 10

for ( r in 1:M ) {
  acc <- matrix(NA, K, 1)
  folds <- rep( 1:K, ceiling(N/K) )
  folds <- sample(folds)
  folds <- folds[1:N_train]
  for ( k in 1:K ) {
    train_fold <- which(folds != k)
    validation <- setdiff(1:N_train, train_fold)

    fit_rf <- randomForest(class ~ ., data = data, subset = train_fold)
    # random forest fitting function utilised within same for loop structure
    pred_rf <- predict(fit_rf, newdata = data[validation,], type = "class")
    acc[k,1] <- class_acc(data$class[validation], pred_rf)
  }

  out_rf[[r]] <- acc
}

avg_per_rep_rf <- sapply(out_rf, colMeans)
mean_acc_rf_1 <- mean(avg_per_rep_rf)
mean_acc_rf_1
```

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

```
stdev_rf_1 <- sd(avg_per_rep_rf)
stdev_rf_1
```

```
## [1] 0
```

Perfect alignment between random tree model, and the training data dropped fold.

.

Zero standard deviation, again showing we have a robust repeatable model.

.

```
rf_all <- randomForest(class~., data = data)
# fitting a random forest to all of the training data
# below predicting hate / no_hate for the training data
pred_rf_all_train <- predict(rf_all, type = "class")
class_acc(data$class, pred_rf_all_train)
```

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

```
# and also for the test data, both using pre-defined class_acc function
pred_rf_all <- predict(rf_all, newdata = data_test, type = "class")
class_acc(data_test$class, pred_rf_all)
```

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

For both the training data, and the test data, the random forest provides a perfect level of prediction.

.

Realistically again, with this level of performance we do not need to tune this model to increase the performance, but nonetheless I will show the process of doing so, if we were to have an underperforming model, we could undertake this process to optimise our predictions.

.

```
mtry <- tuneRF(data[-c(1:18)],data$class, mtryStart = 7,
               #starting this iterative process looking for the optimum mtry at 7
               ntreeTry=200,
               stepFactor = 1.5,
               improve = 0.001,
               # level of improvement required to change mtry
               trace=TRUE,
               plot = FALSE,
               doBest = TRUE,
               nodesize = 5,
               importance=TRUE)
```

```
## mtry = 7  OOB error = 0.02%
## Searching left ...
## mtry = 5     OOB error = 0.02%
## 0 0.001
## Searching right ...
## mtry = 10    OOB error = 0.07%
## -2 0.001
```

```
best.m <- mtry$mtry  # indexing the best mtry value for use in fitting process
best.m <- 5

rf_all2 <- randomForest(class~., data = data, mtry = best.m)
# using optimum mtry from above, in the fitting process
pred_rf_all2 <- predict(rf_all2, newdata = data_test, type = "class")
class_acc(data_test$class, pred_rf_all2)
```

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

```
#prediction, and evaluation on the test data as before
```

mtry is the number of variables randomly sampled as candidates at each split. This is one of the main variables used to optimise the random forest process. The default mtry is the square root, of the number of predictor values available. In this case it is 6.3 rounded down to 6, but we can see that using the tuneRF function, we find that the optimum mtry is in fact 5. We then use that value within the random forest fitting algorithm, which again perfectly predicts the hate / no_hate status of the test data.

Closing Remarks

Many of my comments, evaluations and discussion are above, with their associated code chunks however some closing remarks here.

.

The best performing classifier on the test data was the random forest, therefore this would effectively be my answer to part 2.

.

The random forest was also able to achieve 100% perfect prediction of the test data set, therefore is a perfect tool for detecting hate sentences within a dataset.

I did have to reduce the number of loops within the K fold cross validation of the random forest as it was computationally intensive.

.

There is something to be said for interprability of the models, especially on a topic like this, therefore consideration was given to the classification trees as they performed well, and are the easiest of the 3 models to explain, and the only one which allows graphical explanation of the decision making process. However, we note that the data would not necessarily stay consistent over a long period of time, classification trees can tend to be over-fitted, so I would worry about the longevity of a classification tree model.