[IS] Domača naloga 2

Martin Preradović, Niki Bizjak

9 12 2019

Cleaning

Before doing anything with our text, we have to clean it. The text that we receive consists of unicode characters, escape symbols, HTML tags, etc. Reading the data using `tsvread` does not work, because the text is incorrectly escaped, so we must read it using the `fread` function. After reading tables from disk, we convert training and test labels to factors. Finally, we load custom stopwords list.

```
library(tm)
require(data.table)

# Read training and test data from disk
train_data = fread('./insults/train.tsv', sep = '\t', header = TRUE, quote = '')
test_data = fread('./insults/test.tsv', sep = '\t', header = TRUE, quote = '')

# The label is either 0 or 1, so we can convert it to factor
train_data$label <- as.factor(train_data$label)
test_data$label <- as.factor(test_data$label)

# Read list of english stopwords
stopwords_file = file("./data/stopwords.txt", open="r")
custom_stopwords = readLines(stopwords_file)
close(stopwords_file)
```

After all the data is loaded, we create a corpus source and apply corpus operations on it. We first remove all unicode characters and escape sequences, remove stopwords, remove everything that is not a leter and strip whitespace. We are left with ascii text, but it contains multiple different versions of each word (for example `going`, `goes` and `go`). To remove this we could lemmatise our corpus, but for the sake of speed, we will simply stem our corpus - so remove the suffixes of words.

```
train_corpus <- Corpus(VectorSource(train_data$text_a))
test_corpus <- Corpus(VectorSource(test_data$text_a))

# Remove everything that is not a letter or space (so not [:alpha:])
train_corpus <- tm_map(train_corpus, content_transformer(gsub),
                       pattern = '[^[:alnum:][[:blank:]]]', replacement = ' ')
test_corpus <- tm_map(test_corpus, content_transformer(gsub),
                       pattern = '[^[:alnum:][[:blank:]]]', replacement = ' ')

# Remove unicode characters and "escape" characters
patterns = c("\\\\x[0-9|a-f]{2}", "\\\\u[0-9|a-f]{4}")
for (pattern in patterns) {
  train_corpus <- tm_map(train_corpus, content_transformer(gsub),
                         pattern = pattern, replacement = ' ')
  test_corpus <- tm_map(test_corpus, content_transformer(gsub),
                        pattern = pattern, replacement = ' ')
}

# Remove everything that is not a letter (so not [:alpha:])
```

```
train_corpus <- tm_map(train_corpus, removePunctuation)
train_corpus <- tm_map(train_corpus, removeNumbers)

test_corpus <- tm_map(test_corpus, removePunctuation)
test_corpus <- tm_map(test_corpus, removeNumbers)

# Remove english stopwords and stem them
train_corpus <- tm_map(train_corpus, removeWords, stopwords('english'))
train_corpus <- tm_map(train_corpus, removeWords, custom_stopwords)
# train_corpus <- tm_map(train_corpus, stemDocument)

test_corpus <- tm_map(test_corpus, removeWords, stopwords('english'))
test_corpus <- tm_map(test_corpus, removeWords, custom_stopwords)
# test_corpus <- tm_map(test_corpus, stemDocument)

# Remove whitespace and convert to lowercase
train_corpus <- tm_map(train_corpus, stripWhitespace)
train_corpus <- tm_map(train_corpus, content_transformer(tolower))

test_corpus <- tm_map(test_corpus, stripWhitespace)
test_corpus <- tm_map(test_corpus, content_transformer(tolower))
```

Removal of proper nouns

To remove proper nouns, we first use the `NLP` library to annotate sentences, words and parts-of-speech and then we remove only words that have a POS tag `"NNP"`, which is the tag for proper nouns.

```
library(NLP)
library(openNLP)

sentence_annotator <- Maxent_Sent_Token_Annotator()
word_annotator <- Maxent_Word_Token_Annotator()
pos_annotator <- Maxent_POS_Tag_Annotator()

# Vector of proper nouns that we have found in our text
found_proper_nouns = vector()

for (i in 1:length(train_corpus)) {
  document <- as.String(train_corpus[[i]]$content)

  annotated_sentences <- annotate(document, sentence_annotator)
  annotated_words <- annotate(document, word_annotator, annotated_sentences)
  annotated_pos <- annotate(document, pos_annotator, annotated_words)

  # Get only words from annoteted words (there are sentences in there too)
  words = annotated_pos[annotated_words$type == "word"]

  tags <- vector()
  for (i in 1:length(words$features))
    tags <- c(tags, words$features[[i]]$POS)

  # Get only proper nouns
  proper_nouns = words[tags=="NNP"]
  if (length(proper_nouns) > 0) {
```

```
    found_proper_nouns <- c(found_proper_nouns, document[proper_nouns])
  }
}

# Now that we have found proper nouns, we can remove them from our train and test corpus
train_corpus <- tm_map(train_corpus, removeWords, found_proper_nouns)
test_corpus <- tm_map(test_corpus, removeWords, found_proper_nouns)

# Remove english stopwords and stem them
train_corpus <- tm_map(train_corpus, removeWords, stopwords('english'))
train_corpus <- tm_map(train_corpus, removeWords, custom_stopwords)
# train_corpus <- tm_map(train_corpus, stemDocument)

test_corpus <- tm_map(test_corpus, removeWords, stopwords('english'))
test_corpus <- tm_map(test_corpus, removeWords, custom_stopwords)

train_corpus <- tm_map(train_corpus, stemDocument)
test_corpus <- tm_map(test_corpus, stemDocument)
```

Exploration

Word frequencies

Now that we have cleaned our text, we can plot the frequency of words in our corpus.

```
tdm <- TermDocumentMatrix(train_corpus)

# Calculate word frequency for each word in term-document matrix
term_frequency <- rowSums(as.matrix(tdm))

# Only plot words that appear at least 150 times in all documents
most_frequent <- subset(term_frequency, term_frequency >= 150)
barplot(most_frequent)
```

As we can see from our plot, the most common words are `like`, `you` `get`, `the`, `fuck`, `one` `know`, `make`, `think`, `peopl`, which are all commonly used English words. The word that stands out a bit is `fuck`, but this is

understandable as we are working with insults dataset.

Clustering

For clustering we will use the k-means clustering. The document-term matrix is very sparse, which means that there are a lot of zeros in our matrix. We could remove the most sparse terms in our matrix, but we have decided to only use the most common words to do the clustering.

```
# Get only the 50 most used words from our term_frequency matrix
most_used_words = names(sort(term_frequency, decreasing = TRUE)[1:50])

# First, we construct a document - term matrix
dtm <- DocumentTermMatrix(train_corpus, control = list(weighting=weightTfIdf))
mat <- as.matrix(dtm)

# The matrix mat is very sparse - meaning that there is a lot of empty space
# Because clustering is slow, we will only use the rows from the 50 most used words
mat <- mat[,most_used_words]

# Perform clustering for k = {2, 4, 8, 16}
for (number_of_clusters in c(2, 4, 8, 16)) {
  cat(sprintf("number_of clusters = %d\n", number_of_clusters))

  # Use the k-means clustering with number of clusters
  kmeans_result <- kmeans(mat, number_of_clusters)

  # Find the most popular words in all clusters
  for (i in 1:number_of_clusters) {
      kmeans_result_sorted <- sort(kmeans_result$centers[i,], decreasing=T)
      cat(names(kmeans_result_sorted)[1:10], "\n")
  }

  cat("\n");
}
```

```
## number_of clusters = 2
## fuck like idiot get dont know one right look time
## stupid shit work cant that peopl like dont get fuck
##
## number_of clusters = 4
## post still comment like come never time that want see
## know dont realli man like want never idiot work well
## stupid shit work cant that peopl like dont get fuck
## fuck like idiot get dont one right look time think
##
## number_of clusters = 8
## fuck that want need ass man one take right day
## still comment work year shit fuck tri think like dont
## bitch ass fuck gay get littl look like dont want
## stupid work cant that peopl like dont get fuck one
## get time come see need post dont back like got
## idiot fuck like dont get one peopl know think make
## like dont know one realli look make think say fuck
## right shit like dont get fuck one peopl know think
##
```

```
## number_of clusters = 16
## see get post dont back bitch ass man comment tri
## good look see need way get year know make one
## fuck bitch ass one that want get need like give
## right like dont get fuck one peopl know think make
## idiot fuck like dont get one peopl know think make
## time back take come get post tri need say got
## gay thing right one say peopl get realli shit call
## shit like dont get fuck one peopl know think make
## fuck day like dont get one peopl know think make
## work still know day right better make use one give
## know man never dont like well idiot cant that back
## get one think make like say peopl idiot shit post
## stupid work cant that peopl like dont get fuck one
## like look need man call tri come make life one
## realli know one want like way dont need idiot shit
## dont comment think know get shit need well say post
```

Projection to two dimensions

To project our documents to two dimensions we used both the PCA and t-SNE algorithm. The colors of the plot represent the training label of a document. The t-SNE visualisation is better, as we can more clearly see the individual document clusters in our plot - as we can see, we have one big cluster and many smaller ones.

```
# library(ggplot2)
#
# # First, project our documents in two dimensions using PCA
# pca <- prcomp(mat)
# qplot(pca$x[,1], pca$x[,2], color = train_data$label)
#
# # Then use the t-SNE to perform projection
# library(Rtsne)
# tsne <- Rtsne(mat, perplexity=20, theta=0.2, dims=2, check_duplicates = F)
# qplot(tsne$Y[,1], tsne$Y[,2], color = train_data$label)
```

POS tagging

We can get part-of-speech tags from our documents using `NLP` and `OpenNLP` package. First we use sentence annotator to find sentences, then word annotator to find words and finally we use POS annotator to get list of POS tags.

We will use this tags in the future for training our model, because we think it will improve our machine learning algorithm performance.

```
library(NLP)
library(openNLP)

sentence_annotator <- Maxent_Sent_Token_Annotator()
word_annotator <- Maxent_Word_Token_Annotator()
pos_annotator <- Maxent_POS_Tag_Annotator()

documents_tags <- matrix()

for (document in train_corpus) {
  document <- as.String(document)
```

```
  if (nchar(document) <= 0)
    cbind(documents_tags, vector())
    next

  annotated_sentences <- annotate(document, sentence_annotator)
  annotated_words <- annotate(document, word_annotator, annotated_sentences)
  annotated_pos <- annotate(document, pos_annotator, annotated_words)

  # Get only words from annoteted words (there are sentences in there too)
  words = annotated_pos[annotated_words$type == "word"]

  tags <- vector()
  for (i in 1:length(words$features))
    tags <- c(tags, words$features[[i]]$POS)

  # This is slow, but do we care?
  cbind(documents_tags, tags)
}
```

Modeling

Class label distribution

```
summary(train_data$label)
```

```
##    0    1
## 1630  590
```

The label distribution in our training (and for that matter testing) data is very unbalanced. There are only a quarter texts, that are labeled as insults.

Classification

For the classification, we have selected the *k-nearest neighbors*, *naive Bayes* and *boosted logistic regression* algorithms, as they are easy to understand, very fast and work pretty well with natural language processing problems.

We have also used the *Linear Support Vector Machines* model, because it is believed that it works extremly well with natural language processing problems. The downside of this model is that it is slower than the others.

For the performance metric we chose the *accuracy*, *precision*, *recall*, *f1 score* and *specificity*, because they are easy to calculate and give a good metric of the performance of our models.

We will only use the 60 most frequent terms in our term document matrix, because there is almost no difference in model accuracy if we select more terms. Higher number of features also means more training time.

```
train_dtm = DocumentTermMatrix(train_corpus, control = list(weighting=weightTfIdf))
```

```
## Warning in weighting(x): empty document(s): 731 801 815 1344 1953
```

```
test_dtm <- DocumentTermMatrix(test_corpus, control = list(weighting=weightTfIdf))
```

```
## Warning in weighting(x): empty document(s): 257 677 902
```

```
# Get only words that appear in both train and test data
common_words = intersect(train_dtm$dimnames$Terms, test_dtm$dimnames$Terms);
```

```r
# Order the words in train data by number of occurences in documents
train_tdm = TermDocumentMatrix(train_corpus)
train_word_frequency <- rowSums(as.matrix(train_tdm))
train_most_frequent = names(sort(train_word_frequency, decreasing = TRUE))
most_frequent = intersect(train_most_frequent, common_words)

# Get only the 50 most frequent words that are in both train and test data
most_frequent = most_frequent[1:60]

mat <- as.matrix(train_dtm)
test_mat <- as.matrix(test_dtm)

test_mat <- test_mat[,most_frequent]
mat <- mat[,most_frequent]
```

After we have found the intersection, we can start training our models.

```r
library(caret)
```

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

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

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:NLP':
##
##     annotate
```

```r
library(MLmetrics)
```

```
##
## Attaching package: 'MLmetrics'
```

```
## The following objects are masked from 'package:caret':
##
##     MAE, RMSE
```

```
## The following object is masked from 'package:base':
##
##     Recall
```

```r
# Set a random generator seed, so we get the same result every time
set.seed(1024);

# Create a table from document-term matrix and labels in the end
model_train_data <- data.frame(mat, train_data$label)

possible_methods = c("knn", "naive_bayes", "LogitBoost", "kernelpls", "pls", "svmLinear", "svmLinearWei

# Create a control for our model training - we will use repeated cross validation
control <- trainControl(method = 'repeatedcv', number = 10, repeats = 5)

cat('| Classification model |   Accuracy |  Precision |     Recall |   F1 Score | Specificity|\n');
```

```
## | Classification model |   Accuracy |  Precision |     Recall |   F1 Score | Specificity|
```

```r
for (method in possible_methods) {
  # Train the model using the training data and repeated cv training control
  model <- train(
    train_data.label ~ .,
    data = model_train_data,
    method=method,
    trControl=control
  )

  # To predict many results at once, use the extractPrediction function from carret
  model_predictions = extractPrediction(
    list(mode_name=model),
    testX = test_mat,
    testY = as.factor(test_data$label),
  )

  # The returned matrix has the following columns: obs, pred, model, dataType, object
  # The obs is the observed data type, pred the predicted
  # We are only interested in test data, so we only need elements where dataType is "Test"
  predictions = model_predictions[model_predictions$dataType=='Test',]
  train_predictions = model_predictions[model_predictions$dataType=='Training',]

  # We can now use this data to calculate model accuracy, precision, specificity, etc.
  accuracy = Accuracy(predictions$obs, predictions$pred)
  precision = Precision(predictions$obs, predictions$pred, positive = 1)
  recall = Recall(predictions$obs, predictions$pred)
  f1_score = F1_Score(predictions$obs, predictions$pred, positive = 1)
  specificity <- Specificity(predictions$obs, predictions$pred, positive = 1)

  cat(sprintf("| %20s |     %1.4f |     %1.4f |     %1.4f |     %1.4f |     %1.4f |\n",
              method, accuracy, precision, recall, f1_score, specificity))
}
```

```
##                  knn |     0.7457 |     0.5663 |     0.9503 |     0.2725 |     0.9503 |
##          naive_bayes |     0.7234 |     0.3922 |     0.9572 |     0.1278 |     0.9572 |
##           LogitBoost |     0.7649 |     0.6230 |     0.9366 |     0.3958 |     0.9366 |
##            kernelpls |     0.7710 |     0.8600 |     0.9903 |     0.2756 |     0.9903 |
##                  pls |     0.7710 |     0.8600 |     0.9903 |     0.2756 |     0.9903 |
##            svmLinear |     0.7832 |     0.8429 |     0.9848 |     0.3554 |     0.9848 |
##     svmLinearWeights |     0.7832 |     0.8429 |     0.9848 |     0.3554 |     0.9848 |
```

As we can see from the table above, all algorithms have almost the same accuracy. If we look at the *k-nearest neighbors* algorithm (or any other algorithm for that matter), we can see that it has a high recall, which means that our model has a low number of false negatives That means that the number of documents that were insults and were not classified as insults is low.

We can also see from our table above, that or *knn* model has a relatively low precision, which means that we have a lot of false positives. That means that we have a lot of documents that were not insults, but were classified as insults.

The best trained model is the *svmLinear* model, which has the highest accuracy, precision, recall, f1 score and specificity.

Understanding

For feature ranking we have selected the following filter methods: *Relief*, *ReliefFBestK*, *Informational Gain*, *Gini-index*, *(M)inimum (D)escription (L)ength.*

We can see, that *InfGain*, *Gini* and *MDL* give us similar results.

```
library(COREelearn)

relief_evals <- sort(attrEval(train_data.label ~ ., model_train_data, "Relief"), decreasing = T)
relief_features <- names(relief_evals)

reliefFBestK_evals <- sort(attrEval(train_data.label ~ ., model_train_data, "ReliefFbestK"), decreasing
reliefFBestK_features <- names(reliefFBestK_evals)

infGain_evals <- sort(attrEval(train_data.label ~ ., model_train_data, "InfGain"), decreasing = T)
infGain_features <- names(infGain_evals)

gini_evals <- sort(attrEval(train_data.label ~ ., model_train_data, "Gini"), decreasing = T)
gini_features <- names(gini_evals)

mdl_evals <- sort(attrEval(train_data.label ~ ., model_train_data, "MDL"), decreasing = T)
mdl_features <- names(mdl_evals)

relief_features[1:10]
```

```
##  [1] "idiot" "time"  "one"   "right" "know"  "dont"  "make"  "year"
##  [9] "good"  "man"
```

```
reliefFBestK_features[1:10]
```

```
##  [1] "idiot"  "dont"   "get"    "bitch"  "fuck"   "back"   "ass"
##  [8] "like"   "stupid" "littl"
```

```
infGain_features[1:10]
```

```
##  [1] "idiot"  "bitch"  "stupid" "ass"    "fuck"   "life"   "back"
##  [8] "right"  "shit"   "thing"
```

```
gini_features[1:10]
```

```
##  [1] "idiot"  "bitch"  "stupid" "ass"    "life"   "fuck"   "back"
##  [8] "shit"   "stop"   "right"
```

```
mdl_features[1:10]
```

```
##  [1] "idiot"  "bitch"  "stupid" "ass"    "life"   "fuck"   "back"
##  [8] "right"  "shit"   "gay"
```

Model re-evaluation with top n number of features

For model re-evaluation we have chosen the Gini-index filter method.

```
nofFeatures <- c(1, 2, 3, 5, 7, 9, 15, 20, 30, 40)

method = "svmLinear";
control <- trainControl(method = 'repeatedcv', number = 10, repeats = 3)

acc <- vector()
```

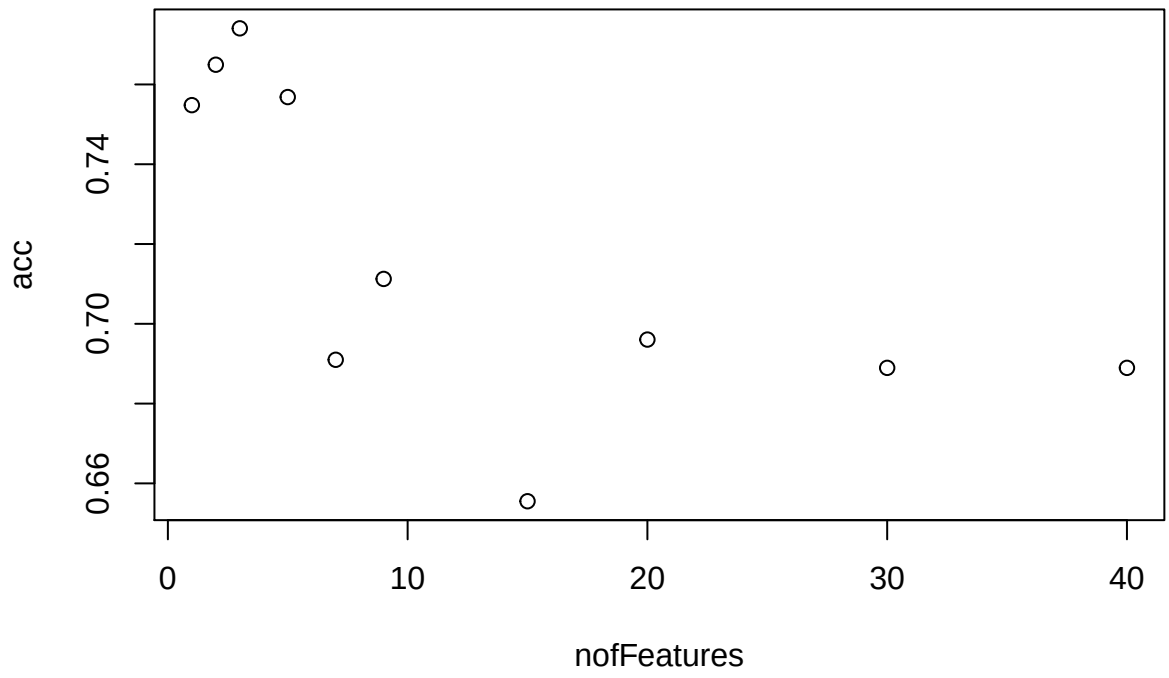
```
prec <- vector()
rec <- vector()
f1 <- vector()
spec <- vector()


for (n in nofFeatures) {

  # First extract a new dtm with only the top n features
  top_train_dtm <- dtm[,intersect(colnames(dtm), gini_features[1:n])]
  tmp_train_mat <- as.matrix(top_train_dtm)

  # Do the same for test data
  top_test_dtm <- DocumentTermMatrix(test_corpus)
  top_test_dtm <- top_test_dtm[, intersect(colnames(top_test_dtm), gini_features[1:n])]
  tmp_test_mat <- as.matrix(top_test_dtm)
  model_train_data <- data.frame(tmp_train_mat, train_data$label)

  # Train the model using the training data and repeated cv training control
  model <- train(
    train_data.label ~ .,
    data = model_train_data,
    method=method,
    trControl=control
  )

  model_predictions = extractPrediction(
    list(mode_name=model),
    testX = tmp_test_mat,
    testY = as.factor(test_data$label),
  )

  predictions = model_predictions[model_predictions$dataType=='Test',]

  acc <- c(acc, Accuracy(predictions$obs, predictions$pred))
  prec <- c(prec, Precision(predictions$obs, predictions$pred))
  rec <- c(rec, Recall(predictions$obs, predictions$pred))
  f1 <- c(f1, F1_Score(predictions$obs, predictions$pred))
  spec <- c(spec, Specificity(predictions$obs, predictions$pred))
}
```

As we can see, the model's performance is highest at 3 selected features and then drops rapidly.

The model's accuracy does not improve with feature selection, but it is comparable to the results of the previous model. However, the training time of the model is reduced with lower number of features.

This can also lower the chances of data-overfitting.

```
plot(x = nofFeatures, y = acc)
```

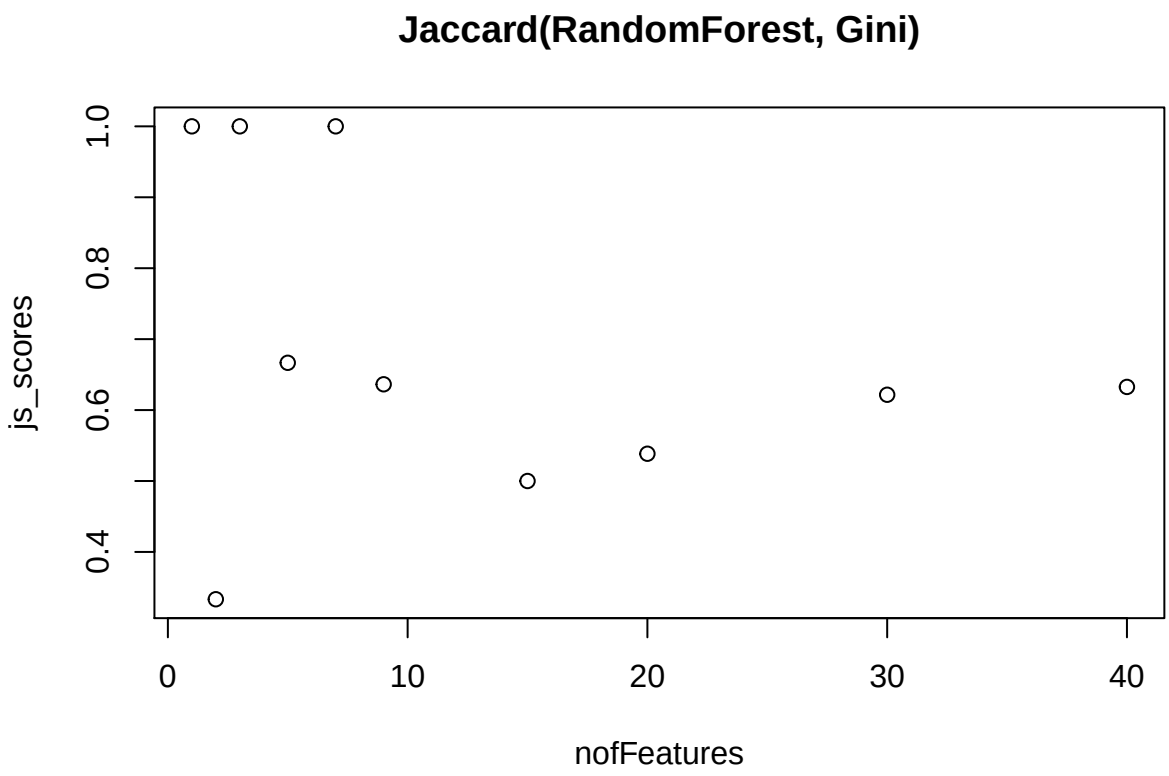


Feature selection with wrapper methods (random forest)

Jaccard's score for Random Forest and Gini-index is generaly high for low n's but gets lower as n increases.

This means that these two methods select features similarly.

```
rf_model = train(x = mat, y = train_data$label, method = "rf", metric = "Accuracy", trControl = control

rf_imp <- varImp(rf_model, scale = FALSE)

rf_imp_values <- rf_imp$importance
rf_imp_names <- rownames(rf_imp$importance)
rf_imp_whole <- data.frame(score = rf_imp_values, cnames = rf_imp_names)
rf_imp_whole <- rf_imp_whole[order(rf_imp_whole$Overall, decreasing = T),]

rf_features <- rf_imp_whole$cnames[1:50]
gini_features <- names(gini_evals)

js_scores <- vector()

for (n in nofFeatures) {
  gini_js <- length(intersect(rf_features[1:n], gini_features[1:n])) / length(union(rf_features[1:n], g
  js_scores <- c(js_scores, gini_js)
}

plot(x=nofFeatures, y=js_scores, title(main='Jaccard(RandomForest, Gini)'))
```


Jaccard(RandomForest, Gini)
js_scores
1.0
0.8
0.6
0.4
0
10
20
30
40
nofFeatures