Session 5 Klassifikation

Christian Kahmann

12 10 2020

In this tutorial we show the use of supervised machine learning for text classification. The basic idea is to compute a model based on training data. Training data usually are hand-coded documents or text snippets associated with a specific category (class). From these texts, features (e.g. words) are extracted and associated with categories in the model. The model then can be utilized to categorize new texts.

We cover basic principles of the process such as cross-validation and feature engineering in the following steps:

1. Read text data
2. Read training data
3. Build feature matrix
4. Classify (LiblineaR)
5. K-fold cross validation
6. Optimize C
7. Optimize features: stopwords, bi-grams, stemming
8. Final classification

As data, again we use the "State of the Union"-addresses. But this time, we operate on paragraphs instead of documents. The file `data/sotu_paragraphs.csv` provides the speeches in the appropriate format. For each paragraph, we want to know whether it covers content related to **domestic or foreign affairs**.

Read paragraphs

As already known, we read the text source (21334 paragraphs from 231 speeches). For the moment, we apply very basic preprocessing.

```
options(stringsAsFactors = FALSE)
library(quanteda)

textdata <- read.csv("data/sotu_paragraphs.csv", sep = ";", encoding = "UTF-8")

corpus <- corpus(textdata$text, docnames = textdata$doc_id)

# Build a dictionary of lemmas
lemma_data <- read.csv("resources/baseform_en.tsv", encoding = "UTF-8")

# Create a DTM
corpus_token <- corpus %>%
  tokens(remove_punct = TRUE, remove_numbers = TRUE, remove_symbols = TRUE) %>%
  tokens_tolower()
```

Load training data

We provide 300 manually annotated example paragraphs as training data. In a CSV-file, the paragraph id and the corresponding category is stored.

```r
# Read previously annotated training data
trainingData <- read.csv2("data/paragraph_training_data_format.csv", stringsAsFactors = T)
# Training data format
colnames(trainingData)
```

```
## [1] "ID"    "LABEL"
```

```r
# Example paragraph Foreign Affairs
set.seed(13)
domestic_example <- sample(trainingData$ID[trainingData$LABEL == "DOMESTIC"], 1)
as.character(texts(corpus)[domestic_example])
```

```
## [1] "For present purposes, however, the last number given should be considerably\nreduced. Without a
```

```r
foreign_example <- sample(trainingData$ID[trainingData$LABEL == "FOREIGN"], 1)
as.character(texts(corpus)[foreign_example])
```

```
## [1] "Our Army and Navy are being maintained at a high state of efficiency, under\nofficers of high t
```

How is the ratio between domestic and foreign content in the training data?

```r
classCounts <- table(trainingData[, "LABEL"])
print(classCounts)
```

```
##
## DOMESTIC  FOREIGN
##      209       91
```

```r
numberOfDocuments <- nrow(trainingData)
```

For our first classification attempt, we create a Document-Term Matrix from the preprocessed corpus and use the extracted single words (unigrams) as features for the classification. Since the resulting DTM might contain too many words, we restrict the vocabulary to a minimum frequency.

```r
# Base line: create feature set out of unigrams
# Probably the DTM is too big for the classifier. Let us reduce it
minimumFrequency <- 5

DTM <- corpus_token %>%
  dfm() %>%
  dfm_trim(min_docfreq = minimumFrequency, max_docfreq = Inf)

# How many features do we have?
dim(DTM)
```

```
## [1] 21334 10948
```

Classification

Now we build a linear classification model with the LiblineaR package. The package LiblineaR wraps around the open source library LIBLINEAR which provides a very fast implementations for two text classification algorithms: 1. Logistic Regression, and 2. Support Vector Machines (SVM) with a linear kernel. For both algorithms, different regularization methods exist (e.g. L1, and L2-regularization). The combination of algorithms and regularization can be controlled by the `type` parameter of the `LiblineaR` function. We stick to the default type, L2-regularized logistic regression (LR), since it usually achieves good performance in text classification.

First, we load the packages. Since Liblinear requires a special Sparse Matrix format, we also load the "SparseM" package and a conversion function which allows to convert `quanteda's dfm`-matrices into `SparseM`-matrices.

Then, we split the annotated data into a training set (80%) and a test set (20%) using a boolean selector. The expression assigned to `selector_idx` creates a boolean vector of length 300 containing a FALSE value in every fifths position. This selector is used to select to training set. Its inverted vector (!) is used to select the test set.

```
require(LiblineaR)
require(SparseM)
source("resources/utils.R")

annotatedDTM <- DTM[trainingData[, "ID"], ]
annotatedDTM <- convertMatrixToSparseM(annotatedDTM)
annotatedLabels <- trainingData[, "LABEL"]

# split into training and test set
selector_idx <- rep(c(rep(TRUE, 4), FALSE), length.out = numberOfDocuments)
trainingDTM <- annotatedDTM[selector_idx, ]
trainingLabels <- annotatedLabels[selector_idx]
testDTM <- annotatedDTM[!selector_idx, ]
testLabels <- annotatedLabels[!selector_idx]

# create LR classification model
model <- LiblineaR(trainingDTM, trainingLabels)
summary(model)
```

```
##            Length Class  Mode
## TypeDetail     1  -none- character
## Type           1  -none- numeric
## W          10949  -none- numeric
## Bias           1  -none- numeric
## ClassNames     2  factor numeric
## NbClass        1  -none- numeric
```

The model created by the LiblineaR function can now be utilized to predict the labels of the test set. Then we compare the result of the automatic classification to our known labels to determine the accuracy of the process.

```
classification <- predict(model, testDTM)
```

```
## Loading required package: SparseM
```

```
##
## Attaching package: 'SparseM'
```

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

```
predictedLabels <- classification$predictions
contingencyTable <- table(predictedLabels, testLabels)
print(contingencyTable)
```

```
##                testLabels
## predictedLabels DOMESTIC FOREIGN
##        DOMESTIC       24       5
##        FOREIGN        21      10
```

```r
accuracy <- sum(diag(contingencyTable)) / length(testLabels)
print(accuracy) # share of correctly classified paragraphs
```

```
## [1] 0.5666667
```

The accuracy of 0.5666667 appears moderate for a first try. But how does it actually relate to a base line? Think of the imbalanced class proportions in our training set. Let us create a pseudo classification as base line, in which we do not classify at all, but simply assume the label "DOMESTIC" or "FOREIGN" for each paragraph.

We further employ a function called `F.measure` which gives more differentiated measures than simple accuracy (`A`) to determine the classification quality. The F.measure (`F`) is the harmonic mean of Precision (`P`) and Recall (`R`) (see https://en.wikipedia.org/wiki/Precision_and_recall#Definition_.28classification_context.29 for Details).

```r
# Create pseudo classification
pseudoLabelsDOM <- factor(rep("DOMESTIC", length(testLabels)), levels(testLabels))
pseudoLabelsFOR <- factor(rep("FOREIGN", length(testLabels)), levels(testLabels))

# Evaluation of former LR classification with F-measures
F.measure(predictedLabels, testLabels, positiveClassName = "DOMESTIC")
```

```
##          P          R          S          F          A       Pos.
##  0.8275862  0.5333333  0.6666667  0.6486486  0.5666667 45.0000000
```

```r
F.measure(predictedLabels, testLabels, positiveClassName = "FOREIGN")
```

```
##          P          R          S          F          A       Pos.
##  0.3225806  0.6666667  0.5333333  0.4347826  0.5666667 15.0000000
```

```r
# Evaluation of pseudo classification with F-measures
F.measure(pseudoLabelsDOM, testLabels, positiveClassName = "DOMESTIC")
```

```
##          P          R          S          F          A       Pos.
##  0.7500000  1.0000000  0.0000000  0.8571429  0.7500000 45.0000000
```

```r
F.measure(pseudoLabelsFOR, testLabels, positiveClassName = "FOREIGN")
```

```
##     P     R     S     F     A  Pos.
##  0.25  1.00  0.00  0.40  0.25 15.00
```

This little experiment shows that depending on the definition of our positive class, the accuracy is either 25% or 75% if not classifying at all. In both cases the *specificity* (`S`), the true negative rate, is zero. From this, we can learn two things:

1. If classes in training/test sets are imbalanced, accuracy might be a misleading measurement. Other measure should be considered additionally.
2. To utilize accuracy and F-measure in a meaningful way, the less frequent class should be defined as POSITIVE class (FOREIGN in our case).

K-fold cross validation

To evaluate a classifier, the training data can be divided into training and test data. The model learns on the former and is evaluated with the latter. In this procedure, we unfortunately lose the test data to learn from. If there is little training data available, the k-fold cross-validation is a more suitable procedure.

For this, training data is split into e.g. K = 10 parts. Then k-1 parts are used for training and 1 part is used for testing. This process is repeated k times, with another split of the overall data set for testing in each iteration.

The final result is determined from the average of the quality of the k runs. This allows a good approximation to the classification quality, including all training data.

The `get_k_fold_logical_indexes` function introduced below returns a logical vector for the fold `j` for cross-validation. It splits a training data record of the size `n` into `k` folds. The resulting vector and its negated vector can be used for easy training data / test data selection.

```
get_k_fold_logical_indexes <- function(j, k, n) {
  if (j > k) stop("Cannot select fold larger than nFolds")
  fold_lidx <- rep(FALSE, k)
  fold_lidx[j] <- TRUE
  fold_lidx <- rep(fold_lidx, length.out = n)
  return(fold_lidx)
}

# Example usage
get_k_fold_logical_indexes(1, k = 10, n = 12)
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [12] FALSE
```

```
get_k_fold_logical_indexes(2, k = 10, n = 12)
```

```
##  [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12]  TRUE
```

```
get_k_fold_logical_indexes(3, k = 10, n = 12)
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE
```

Now we run 1) splitting of the annotated data, 2) model computation and testing in one for-loop.

```
k <- 10
evalMeasures <- NULL
for (j in 1:k) {
  # create j-th boolean selection vector
  currentFold <- get_k_fold_logical_indexes(j, k, nrow(trainingDTM))

  # select training data split
  foldDTM <- annotatedDTM[!currentFold, ]
  foldLabels <- annotatedLabels[!currentFold]

  # create model
  model <- LiblineaR(foldDTM, foldLabels)

  # select test data split
  testSet <- annotatedDTM[currentFold, ]
  testLabels <- annotatedLabels[currentFold]

  # predict test labels
  predictedLabels <- predict(model, testSet)$predictions

  # evaluate predicted against test labels
  kthEvaluation <- F.measure(predictedLabels, testLabels, positiveClassName = "FOREIGN")
```

```
  # combine evaluation measures for k runs
  evalMeasures <- rbind(evalMeasures, kthEvaluation)
}
# Final evaluation values of k runs:
print(evalMeasures)
```

```
##                       P         R         S         F         A Pos.
## kthEvaluation 0.3333333 0.4000000 0.6000000 0.3636364 0.5333333   10
## kthEvaluation 0.7368421 0.8750000 0.6428571 0.8000000 0.7666667   16
## kthEvaluation 0.3333333 0.4285714 0.7391304 0.3750000 0.6666667    7
## kthEvaluation 0.5000000 0.7777778 0.6666667 0.6086957 0.7000000    9
## kthEvaluation 0.3333333 0.5714286 0.6521739 0.4210526 0.6333333    7
## kthEvaluation 0.3125000 0.7142857 0.5217391 0.4347826 0.5666667    7
## kthEvaluation 0.4705882 0.7272727 0.5263158 0.5714286 0.6000000   11
## kthEvaluation 0.5000000 0.8333333 0.7916667 0.6250000 0.8000000    6
## kthEvaluation 0.5263158 1.0000000 0.5500000 0.6896552 0.7000000   10
## kthEvaluation 0.3846154 0.6250000 0.6363636 0.4761905 0.6333333    8
```

```
# Average over all folds
print(colMeans(evalMeasures))
```

```
##         P         R         S         F         A      Pos.
## 0.4430862 0.6952670 0.6326913 0.5365441 0.6600000 9.1000000
```

Accuracy is around 0.66%, F-measure is around 0.54%. Let's try some approaches to optimize the automatic classification.

Optimization

These first tries have clarified how to utilize and evaluate machine learning functions for text in R. Now we concentrate on optimization strategies to get better results from the automatic classification process.

C-Parameter

For a linear classification model, the cost parameter (C-parameter) is the most important parameter to tweak (for other SVM kernels such as the radial or polynomial kernel there are other important parameters which influence the shape of the kernel function). The **C-parameter** determines the cost of classifications on the training data during training.

High values of C lead to a high costs of misclassification. The decision boundary which the classifier learns, will try to avoid any misclassification. But, values too high can lead to an overfitting of the model. This means, it adapts well to the training data, but classification will more likely fail on new test data.

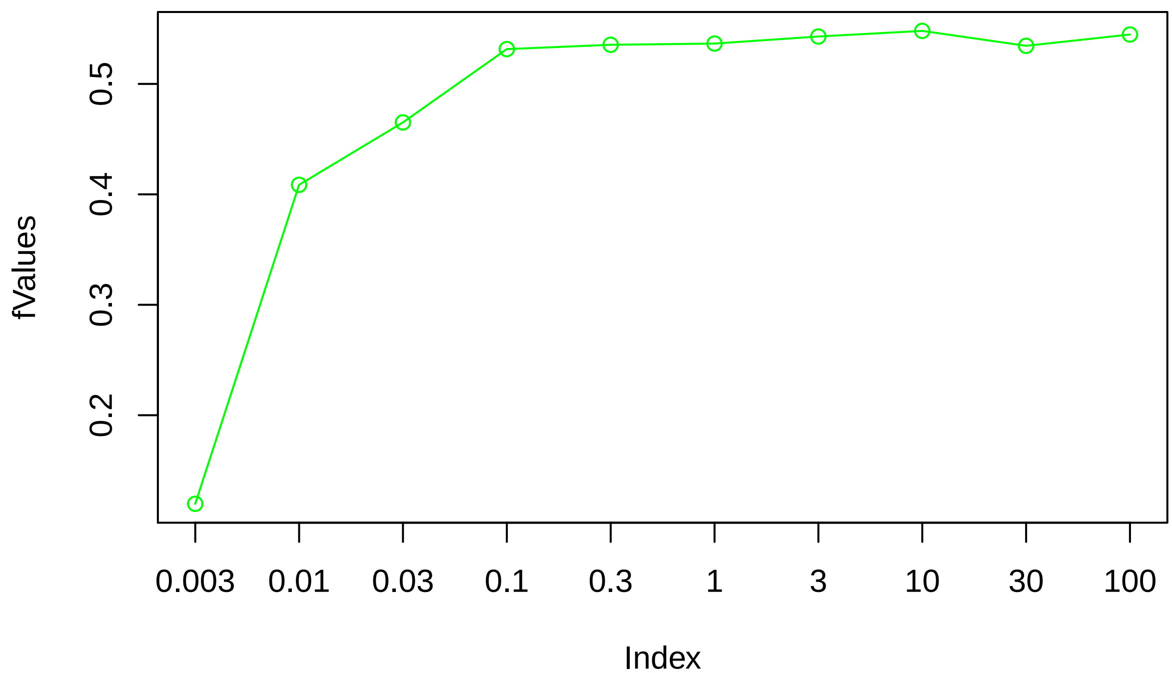
Low values of C lead to less strict decision boundaries, which accepts some misclassifications. Such a model might generalize better on unseen data. But in the end, there is now exact method to determine a good C-value beforehand. It rather is an empirical question. To choose an optimal C-value, we simply try from a range of values, run k-fold-cross-validation for each single value and decide for the C which resulted in the best classification accuracy / F-measure. This is realized in the following for-loop, which utilizes the function `k_fold_cross_validation`. The function (have a look into `F.measure.R`) simply wraps the code for cross-validation, we used above.

```
cParameterValues <- c(0.003, 0.01, 0.03, 0.1, 0.3, 1, 3 , 10, 30, 100)
fValues <- NULL
```

```r
for (cParameter in cParameterValues) {
  print(paste0("C = ", cParameter))
  evalMeasures <- k_fold_cross_validation(annotatedDTM, annotatedLabels, cost = cParameter)
  fValues <- c(fValues, evalMeasures["F"])
}
```

```
## [1] "C = 0.003"
## [1] "C = 0.01"
## [1] "C = 0.03"
## [1] "C = 0.1"
## [1] "C = 0.3"
## [1] "C = 1"
## [1] "C = 3"
## [1] "C = 10"
## [1] "C = 30"
## [1] "C = 100"
```

```r
plot(fValues, type="o", col="green", xaxt="n")
axis(1,at=1:length(cParameterValues), labels = cParameterValues)
```




```r
bestC <- cParameterValues[which.max(fValues)]
print(paste0("Best C value: ", bestC, ", F1 = ", max(fValues)))
```

```
## [1] "Best C value: 10, F1 = 0.548005843586188"
```

From the empirical test, we can obtain C = 10 as optimal choice for the cost parameter. On the current training data set with the current features it achieves 0.5480058 F-score.

Optimized Preprocessing

Not only the classification model has parameters which can be optimized to improve the results. More important are the features used for classification. In our preprocessing chain above, we extracted single types and transformed them into lower case. We now add different preprocessing steps and check on the results. To get an optimal cost parameter for each new feature set, we wrapped the code for C-parameter optimization into the `optimize_C` function.

Stop word removal

Stop words often do not contribute to the meaning of a text. For the decision between DOMESTIC and FOREIGN affairs, we do not expect any useful information from them. So let's remove them and if it improves the classifier.

```
stopwords_extended <- readLines("resources/stopwords_en.txt", encoding = "UTF-8")

# preprocessing
corpus_token_sw <- corpus %>%
  tokens(remove_punct = TRUE, remove_numbers = TRUE, remove_symbols = TRUE) %>%
  tokens_tolower() %>%
  tokens_remove(pattern = stopwords_extended)

print(paste0("1: ", substr(paste(corpus_token_sw[4963],collapse = " "), 0, 400), '...'))
```

```
## [1] "1: legislation extend colony newfoundland articles treaty washington 8th day protocol effect si
```

```
minimumFrequency <- 5

# Compute DTM
DTM <- corpus_token_sw %>%
  dfm() %>%
  dfm_trim(min_docfreq = minimumFrequency,max_docfreq = Inf)

# How many features do we have?
dim(DTM)
```

```
## [1] 21334 10446
```

```
# run cross validation
annotatedDTM <- convertMatrixToSparseM(DTM[trainingData[, "ID"], ])
best_C <- optimize_C(annotatedDTM, annotatedLabels)
```

```
## [1] "C = 0.003"
## [1] "C = 0.01"
## [1] "C = 0.03"
## [1] "C = 0.1"
## [1] "C = 0.3"
## [1] "C = 1"
## [1] "C = 3"
## [1] "C = 10"
## [1] "C = 30"
## [1] "C = 100"
## [1] "Best C value: 0.03, F1 = 0.647465370415286"
```

```
k_fold_cross_validation(annotatedDTM, annotatedLabels, cost = best_C)
```

```
##         P         R         S         F         A      Pos.
## 0.5959710 0.7254419 0.7995344 0.6474654 0.7800000 9.1000000
```

Bigrams

Now let us see, if the use of bigrams, i.e. concatenations of sequences of two words can improve the result. Bigrams, and larger n-Grams can capture important sequential contexts from texts such as negation, at least to a certain extent. For instance, "is not funny" will result in the features "is_not" and "not_funny".

To extract n-gram features, the `tokens_ngrams()` of `quanteda` accepts a list of integers, e.g. `1:2` to create unigram and bigram features.

```
corpus_token_bigrams <- corpus %>%
  tokens(remove_punct = TRUE, remove_numbers = TRUE, remove_symbols = TRUE) %>%
  tokens_tolower() %>%
  tokens_remove(pattern = stopwords_extended) %>%
  tokens_ngrams(1:2)

print(paste0("1: ", substr(paste(corpus_token_bigrams[4963], collapse = " "), 0, 400), '...'))
```

```
## [1] "1: legislation extend colony newfoundland articles treaty washington 8th day protocol effect si
```

```
# Compute DTM
DTM <- corpus_token_bigrams  %>%
  dfm() %>%
  dfm_trim(min_docfreq = minimumFrequency, max_docfreq = Inf)

# How many features do we have now?
dim(DTM)
```

```
## [1] 21334 22799
```

```
# How do they look like?
sample(colnames(DTM), 10)
```

```
##  [1] "agreeing"       "recall"         "utilizing"      "fixed"
##  [5] "lafayette"      "leader"         "freights"       "plunder"
##  [9] "war_powers"     "world_watching"
```

```
# run cross validation
annotatedDTM <- convertMatrixToSparseM(DTM[trainingData[, "ID"], ])
best_C <- optimize_C(annotatedDTM, annotatedLabels)
```

```
## [1] "C = 0.003"
## [1] "C = 0.01"
## [1] "C = 0.03"
## [1] "C = 0.1"
## [1] "C = 0.3"
## [1] "C = 1"
## [1] "C = 3"
## [1] "C = 10"
## [1] "C = 30"
## [1] "C = 100"
## [1] "Best C value: 0.03, F1 = 0.644608227558143"
```

```
k_fold_cross_validation(annotatedDTM, annotatedLabels, cost = best_C)
```

```
##         P         R         S         F         A      Pos.
## 0.5906138 0.7254419 0.7951865 0.6446082 0.7766667 9.1000000
```

In this case, bi-gram and tri-gram features seem to not contribute positively to the classification result.

Minimum feature frequency

Up to this point, we dropped features occurring less than five times in our data. Let's see if we include more features by increasing the minimum frequency.

```
# More features
minimumFrequency <- 2

DTM <- corpus_token_sw  %>%
  dfm() %>%
  dfm_trim(min_docfreq = minimumFrequency, max_docfreq = Inf)

dim(DTM)
```

```
## [1] 21334 16674
```

```
annotatedDTM <- convertMatrixToSparseM(DTM[trainingData[, "ID"], ])
best_C <- optimize_C(annotatedDTM, annotatedLabels)
```

```
## [1] "C = 0.003"
## [1] "C = 0.01"
## [1] "C = 0.03"
## [1] "C = 0.1"
## [1] "C = 0.3"
## [1] "C = 1"
## [1] "C = 3"
## [1] "C = 10"
## [1] "C = 30"
## [1] "C = 100"
## [1] "Best C value: 0.03, F1 = 0.651274894224809"
```

```
k_fold_cross_validation(annotatedDTM, annotatedLabels, cost = best_C)
```

```
##         P         R         S         F         A      Pos.
## 0.6020316 0.7254419 0.8042963 0.6512749 0.7833333 9.1000000
```

It seems that feeding more features into the classifier has a little positive effect on the result.

Lemmatization

As a last method, we utilize lemmatization to unify different variants of the same semantic form (such as *nation* and *nations*).

```
corpus_token_lemma <- corpus %>%
  tokens(remove_punct = TRUE, remove_numbers = TRUE, remove_symbols = TRUE) %>%
  tokens_tolower() %>%
  tokens_replace(lemma_data$inflected_form, lemma_data$lemma) %>%
  tokens_remove(pattern = stopwords_extended) %>%
  tokens_ngrams(1)

print(paste0("1: ", substr(paste(corpus_token_lemma[4963], collapse = " "), 0, 400), '...'))
```

```
## [1] "1: legislation extend colony newfoundland article treaty washington 8th day protocol effect sig
```

```
minimumFrequency <- 2

DTM <- corpus_token_lemma  %>%
  dfm() %>%
  dfm_trim(min_docfreq = minimumFrequency, max_docfreq = Inf)

dim(DTM)
```

```
## [1] 21334 11509
```

```r
# run cross validation
annotatedDTM <- convertSlamToSparseM(DTM[trainingData[, "ID"], ])
best_C <- optimize_C(annotatedDTM, annotatedLabels)
```

```
## [1] "C = 0.003"
## [1] "C = 0.01"
## [1] "C = 0.03"
## [1] "C = 0.1"
## [1] "C = 0.3"
## [1] "C = 1"
## [1] "C = 3"
## [1] "C = 10"
## [1] "C = 30"
## [1] "C = 100"
## [1] "Best C value: 0.03, F1 = 0.69005439005439"
```

```r
k_fold_cross_validation(annotatedDTM, annotatedLabels, cost = best_C)
```

```
##         P         R         S         F         A      Pos.
## 0.6409369 0.7668543 0.8170156 0.6900544 0.8066667 9.1000000
```

Each individual approach to optimize our text features for classification has some effect on the results. It takes some time to engineer an optimal feature set.

Further, testing different features must be done for each new task / language individually, since there is no "one-size fits all" approach to this.

GENERAL ADVISE: For this tutorial we utilized a rather small training set of 300 examples, 91 of them in the positive class. Better classification accuracy can be expected, if more training data is available. Hence, instead of spending too much time on feature optimization, it will probably be a better idea to invest into generation of more training data first.

Final classification

We now apply our best classification model to the entire data set, to determine the occurrence of FORGEIN/DOMESTIC affairs related content in each single speech.

```r
# Final classification
annotatedDTM <- convertMatrixToSparseM(DTM[trainingData[, "ID"], ])

# C parameter tuning
best_C <- optimize_C(annotatedDTM, annotatedLabels)
```

```
## [1] "C = 0.003"
## [1] "C = 0.01"
## [1] "C = 0.03"
## [1] "C = 0.1"
## [1] "C = 0.3"
## [1] "C = 1"
## [1] "C = 3"
## [1] "C = 10"
## [1] "C = 30"
## [1] "C = 100"
## [1] "Best C value: 0.03, F1 = 0.69005439005439"
```

```r
# final classification
final_model <- LiblineaR(annotatedDTM, annotatedLabels, cost = best_C)
final_labels <- predict(final_model, convertSlamToSparseM(DTM))$predictions
table(final_labels) / sum(table(final_labels))
```

```
## final_labels
##  DOMESTIC   FOREIGN
## 0.5945439 0.4054561
```

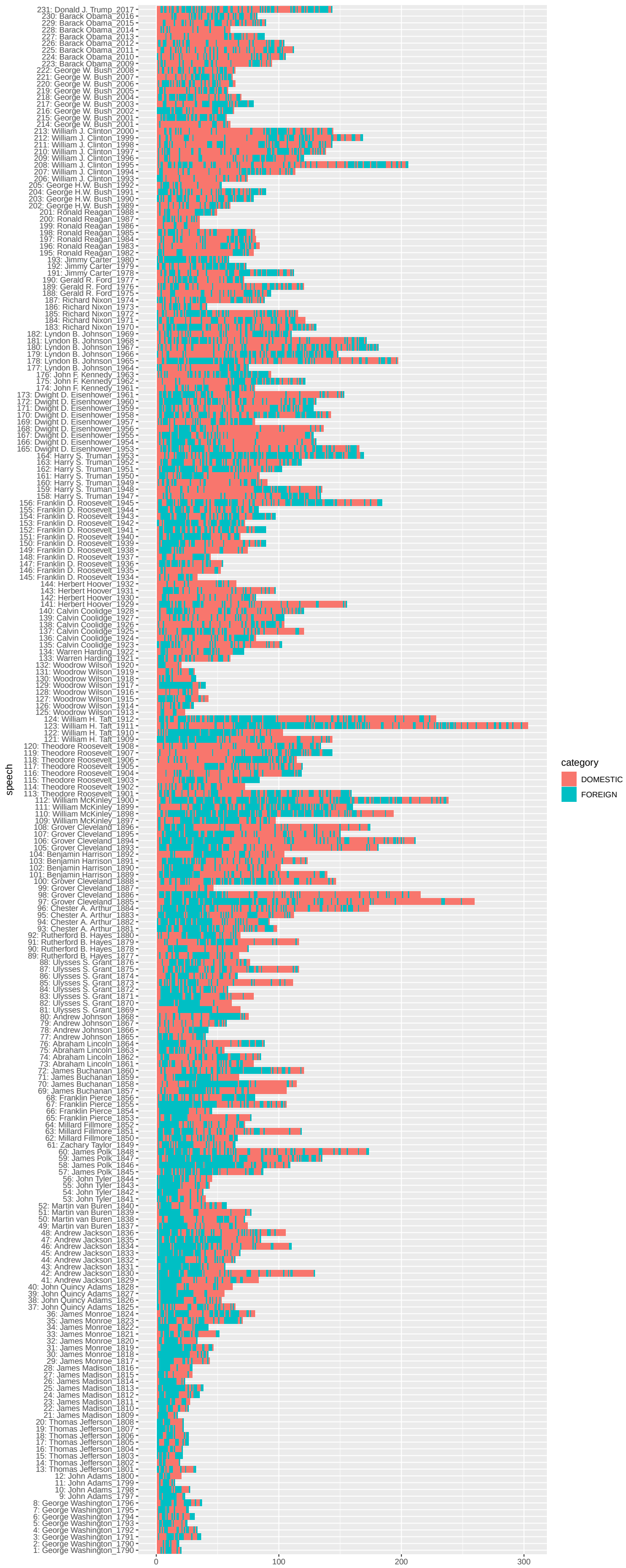
We see that the classifier puts the majority of the around 21,000 paragraphs into the DOMESTIC category. We can visualize the result as a bar chart with `ggplot2`.

```r
speech_year <- substr(textdata$date, 0, 4)
speech_id <- textdata$speech_doc_id
paragraph_position <- unlist(sapply(table(speech_id), FUN = function(x) {1:x}))
presidents_df <- data.frame(
  paragraph_position = paragraph_position,
  speech = paste0(speech_id, ": ", textdata$president, "_", speech_year),
  category = final_labels
)

# preserve speech order in chart by using factors
presidents_df$speech <- factor(presidents_df$speech, levels = unique(presidents_df$speech))

# Remove two very long speeches to beautify the plot (you can also skipt this)
presidents_df <- presidents_df[!grepl("Carter_1981|Truman_1946", presidents_df$speech), ]

# plot classes of paragraphs for each speech as tile
require(ggplot2)
ggplot(presidents_df, aes(x = speech, y = paragraph_position, fill = category)) +
  geom_tile(stat="identity") + coord_flip()
```


speech
paragraph_position
0
100
200
300
category
DOMESTIC
FOREIGN

Can you see how DOMESTIC affairs related content gets more important over the course of centuries? Also the position of FOREIGN policy statements changes around the turn from the 19th to 20th century from the begginning of a speech to more dispersed positions thoughout the speech, and finally a tendency to rather place them at the end of speeches.