This is code that will encompany an article that will appear in a special edition of a German IT magazine. The article is about explaining black-box machine learning models. In that article I’m showcasing three practical examples:

1. Explaining supervised classification models built on tabular data using caret and the iml package
2. Explaining image classification models with keras and lime
3. Explaining text classification models with xgboost and lime

Text classification with lime below:

**An example**

Out of the box lime supports a long range of models, e.g. those created with caret, parsnip, and mlr. Support for unsupported models are easy to achieve by adding a predict\_model and model\_type method for the given model.

The following shows how a random forest model is trained on the iris data set and how lime is then used to explain a set of new observations:

library(caret)

library(lime)

# Split up the data set

iris\_test <- iris[1:5, 1:4]

iris\_train <- iris[-(1:5), 1:4]

iris\_lab <- iris[[5]][-(1:5)]

# Create Random Forest model on iris data

model <- train(iris\_train, iris\_lab, method = 'rf')

# Create an explainer object

explainer <- lime(iris\_train, model)

# Explain new observation

explanation <- explain(iris\_test, explainer, n\_labels = 1, n\_features = 2)

# The output is provided in a consistent tabular format and includes the

# output from the model.

explanation

#> # A tibble: 10 x 13

#> model\_type case label label\_prob model\_r2 model\_intercept model\_prediction

#> <chr> <chr> <chr> <dbl> <dbl> <dbl> <dbl>

#> 1 classific… 1 seto… 1 0.681 0.129 0.987

#> 2 classific… 1 seto… 1 0.681 0.129 0.987

#> 3 classific… 2 seto… 1 0.692 0.123 0.984

#> 4 classific… 2 seto… 1 0.692 0.123 0.984

#> 5 classific… 3 seto… 1 0.686 0.129 0.983

#> 6 classific… 3 seto… 1 0.686 0.129 0.983

#> 7 classific… 4 seto… 1 0.695 0.119 0.985

#> 8 classific… 4 seto… 1 0.695 0.119 0.985

#> 9 classific… 5 seto… 1 0.694 0.123 0.984

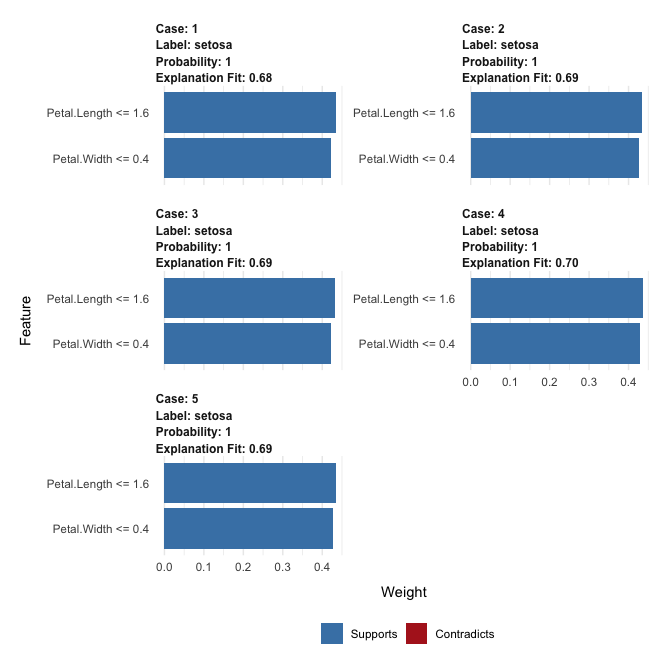
#> 10 classific… 5 seto… 1 0.694 0.123 0.984

#> # … with 6 more variables: feature <chr>, feature\_value <dbl>,

#> # feature\_weight <dbl>, feature\_desc <chr>, data <list>, prediction <list>

# And can be visualised directly

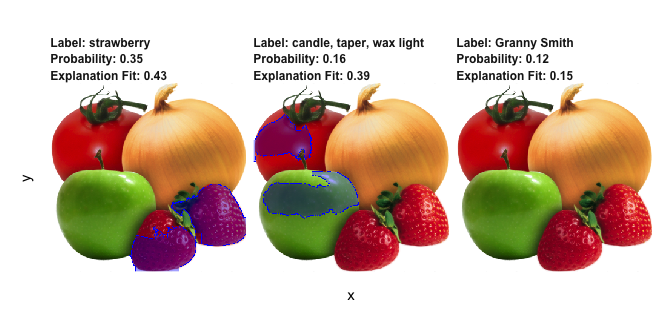
plot\_features(explanation)

[](https://github.com/thomasp85/lime/blob/master/man/figures/README-unnamed-chunk-2-1.png)

lime also supports explaining image and text models. For image explanations the relevant areas in an image can be highlighted:

explanation <- .load\_image\_example()

plot\_image\_explanation(explanation)

[](https://github.com/thomasp85/lime/blob/master/man/figures/README-unnamed-chunk-3-1.png)

Here we see that the second most probably class is hardly true, but is due to the model picking up waxy areas of the produce and interpreting them as wax-light surface.

Below, you will find the code for the third part:

# data wrangling

library(tidyverse)

library(readr)

# plotting

library(ggthemes)

theme\_set(theme\_minimal())

# text prep

library(text2vec)

# ml

library(caret)

library(xgboost)

# explanation

library(lime)

**Text classification models**

Here I am using another Kaggle dataset: [Women’s e-commerce cloting reviews](https://www.kaggle.com/nicapotato/womens-ecommerce-clothing-reviews). The data contains a text review of different items of clothing, as well as some additional information, like rating, division, etc.

In this example, I will use the review title and text in order to classify whether or not the item was liked. I am creating the response variable from the rating: every item rates with 5 stars is considered “liked” (1), the rest as “not liked” (0). I am also combining review title and text.

clothing\_reviews <- read\_csv("/Users/shiringlander/Documents/Github/ix\_lime\_etc/Womens Clothing E-Commerce Reviews.csv") %>%

mutate(Liked = as.factor(ifelse(Rating == 5, 1, 0)),

text = paste(Title, `Review Text`),

text = gsub("NA", "", text))

## Parsed with column specification:

## cols(

## X1 = col\_integer(),

## `Clothing ID` = col\_integer(),

## Age = col\_integer(),

## Title = col\_character(),

## `Review Text` = col\_character(),

## Rating = col\_integer(),

## `Recommended IND` = col\_integer(),

## `Positive Feedback Count` = col\_integer(),

## `Division Name` = col\_character(),

## `Department Name` = col\_character(),

## `Class Name` = col\_character()

## )

glimpse(clothing\_reviews)

## Observations: 23,486

## Variables: 13

## $ X1 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11...

## $ `Clothing ID` 767, 1080, 1077, 1049, 847, 1080, 85...

## $ Age 33, 34, 60, 50, 47, 49, 39, 39, 24, ...

## $ Title NA, NA, "Some major design flaws", "...

## $ `Review Text` "Absolutely wonderful - silky and se...

## $ Rating 4, 5, 3, 5, 5, 2, 5, 4, 5, 5, 3, 5, ...

## $ `Recommended IND` 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, ...

## $ `Positive Feedback Count` 0, 4, 0, 0, 6, 4, 1, 4, 0, 0, 14, 2,...

## $ `Division Name` "Initmates", "General", "General", "...

## $ `Department Name` "Intimate", "Dresses", "Dresses", "B...

## $ `Class Name` "Intimates", "Dresses", "Dresses", "...

## $ Liked 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, ...

## $ text " Absolutely wonderful - silky and s...

Whether an item was liked or not will thus be my response variable or label for classification.

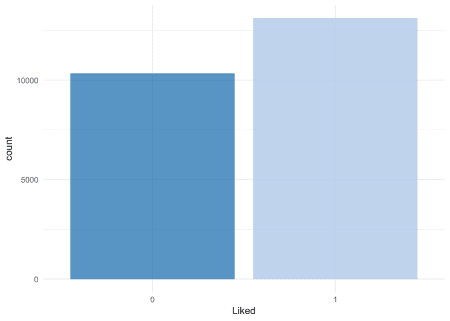
clothing\_reviews %>%

ggplot(aes(x = Liked, fill = Liked)) +

geom\_bar(alpha = 0.8) +

scale\_fill\_tableau(palette = "tableau20") +

guides(fill = FALSE)



Let’s split the data into train and test sets:

set.seed(42)

idx <- createDataPartition(clothing\_reviews$Liked,

p = 0.8,

list = FALSE,

times = 1)

clothing\_reviews\_train <- clothing\_reviews[ idx,]

clothing\_reviews\_test <- clothing\_reviews[-idx,]

**Let’s start simple**

The first text model I’m looking at has been built similarly to the example model in the help for lime::interactive\_text\_explanations().

First, we need to prepare the data for modeling: we will need to convert the text to a document term matrix (dtm). There are different ways to do this. One is be with the text2vec package.

“Because of R’s copy-on-modify semantics, it is not easy to iteratively grow a DTM. Thus constructing a DTM, even for a small collections of documents, can be a serious bottleneck for analysts and researchers. It involves reading the whole collection of text documents into RAM and processing it as single vector, which can easily increase memory use by a factor of 2 to 4. The text2vec package solves this problem by providing a better way of constructing a document-term matrix.”

Text Vectrorization

In this vignette we will primarily discuss the first step. Texts themselves can take up a lot of memory, but vectorized texts usually do not, because they are stored as sparse matrices. Because of R’s copy-on-modify semantics, it is not easy to iteratively grow a DTM. Thus constructing a DTM, even for a small collections of documents, can be a serious bottleneck for analysts and researchers. It involves reading the whole collection of text documents into RAM and processing it as single vector, which can easily increase memory use by a factor of 2 to 4. The text2vec package solves this problem by providing a better way of constructing a document-term matrix.

Let’s demonstrate package core functionality by applying it to a real case problem - sentiment analysis.

text2vec package provides the movie\_review dataset. It consists of 5000 movie reviews, each of which is marked as positive or negative. We will also use the data.table package for data wrangling.

First of all let’s split out dataset into two parts - train and test. We will show how to perform data manipulations on train set and then apply exactly the same manipulations on the test set:

**library**(text2vec)

**library**(data.table)

**library**(magrittr)

data("movie\_review")

setDT(movie\_review)

setkey(movie\_review, id)

set.seed(2017L)

all\_ids = movie\_review$id

train\_ids = sample(all\_ids, 4000)

test\_ids = setdiff(all\_ids, train\_ids)

train = movie\_review[J(train\_ids)]

test = movie\_review[J(test\_ids)]

# **Vectorization**

To represent documents in vector space, we first have to create mappings from terms to term IDS. We call them terms instead of words because they can be arbitrary n-grams not just single words. We represent a set of documents as a sparse matrix, where each row corresponds to a document and each column corresponds to a term. This can be done in 2 ways: using the vocabulary itself or by feature hashing.

## Vocabulary-based vectorization

Let’s first create a vocabulary-based DTM. Here we collect unique terms from all documents and mark each of them with a unique ID using the create\_vocabulary() function. We use an iterator to create the vocabulary.

*# define preprocessing function and tokenization function*

prep\_fun = tolower

tok\_fun = word\_tokenizer

it\_train = itoken(train$review,

preprocessor = prep\_fun,

tokenizer = tok\_fun,

ids = train$id,

progressbar = FALSE)

vocab = create\_vocabulary(it\_train)

What was done here?

1. We created an iterator over tokens with the itoken() function. All functions prefixed with create\_ work with these iterators. R users might find this idiom unusual, but the iterator abstraction allows us to hide most of details about input and to process data in memory-friendly chunks.
2. We built the vocabulary with the create\_vocabulary() function.

Alternatively, we could create list of tokens and reuse it in further steps. Each element of the list should represent a document, and each element should be a character vector of tokens.

train\_tokens = tok\_fun(prep\_fun(train$review))

it\_train = itoken(train\_tokens,

ids = train$id,

*# turn off progressbar because it won't look nice in rmd*

progressbar = FALSE)

vocab = create\_vocabulary(it\_train)

vocab

## Number of docs: 4000

## 0 stopwords: ...

## ngram\_min = 1; ngram\_max = 1

## Vocabulary:

## term term\_count doc\_count

## 1: 0.02 1 1

## 2: 0.3 1 1

## 3: 0.48 1 1

## 4: 0.5 1 1

## 5: 0.89 1 1

## ---

## 38450: to 21891 3796

## 38451: of 23477 3794

## 38452: a 26398 3880

## 38453: and 26917 3868

## 38454: the 53871 3970

Note that text2vec provides a few tokenizer functions (see ?tokenizers). These are just simple wrappers for the base::gsub() function and are not very fast or flexible. If you need something smarter or faster you can use the tokenizers package which will cover most use cases, or write your own tokenizer using the stringi package.

Now that we have a vocabulary, we can construct a document-term matrix.

vectorizer = vocab\_vectorizer(vocab)

t1 = Sys.time()

dtm\_train = create\_dtm(it\_train, vectorizer)

print(difftime(Sys.time(), t1, units = 'sec'))

## Time difference of 0.5483496 secs

Now we have a DTM and can check its dimensions.

dim(dtm\_train)

## [1] 4000 38454

identical(rownames(dtm\_train), train$id)

## [1] TRUE

As you can see, the DTM has 4000 rows, equal to the number of documents, and 38454 columns, equal to the number of unique terms.

Now we are ready to fit our first model. Here we will use the glmnet package to fit a logistic regression model with an L1 penalty and 4 fold cross-validation.

**library**(glmnet)

NFOLDS = 4

t1 = Sys.time()

glmnet\_classifier = cv.glmnet(x = dtm\_train, y = train[['sentiment']],

family = 'binomial',

*# L1 penalty*

alpha = 1,

*# interested in the area under ROC curve*

type.measure = "auc",

*# 5-fold cross-validation*

nfolds = NFOLDS,

*# high value is less accurate, but has faster training*

thresh = 1e-3,

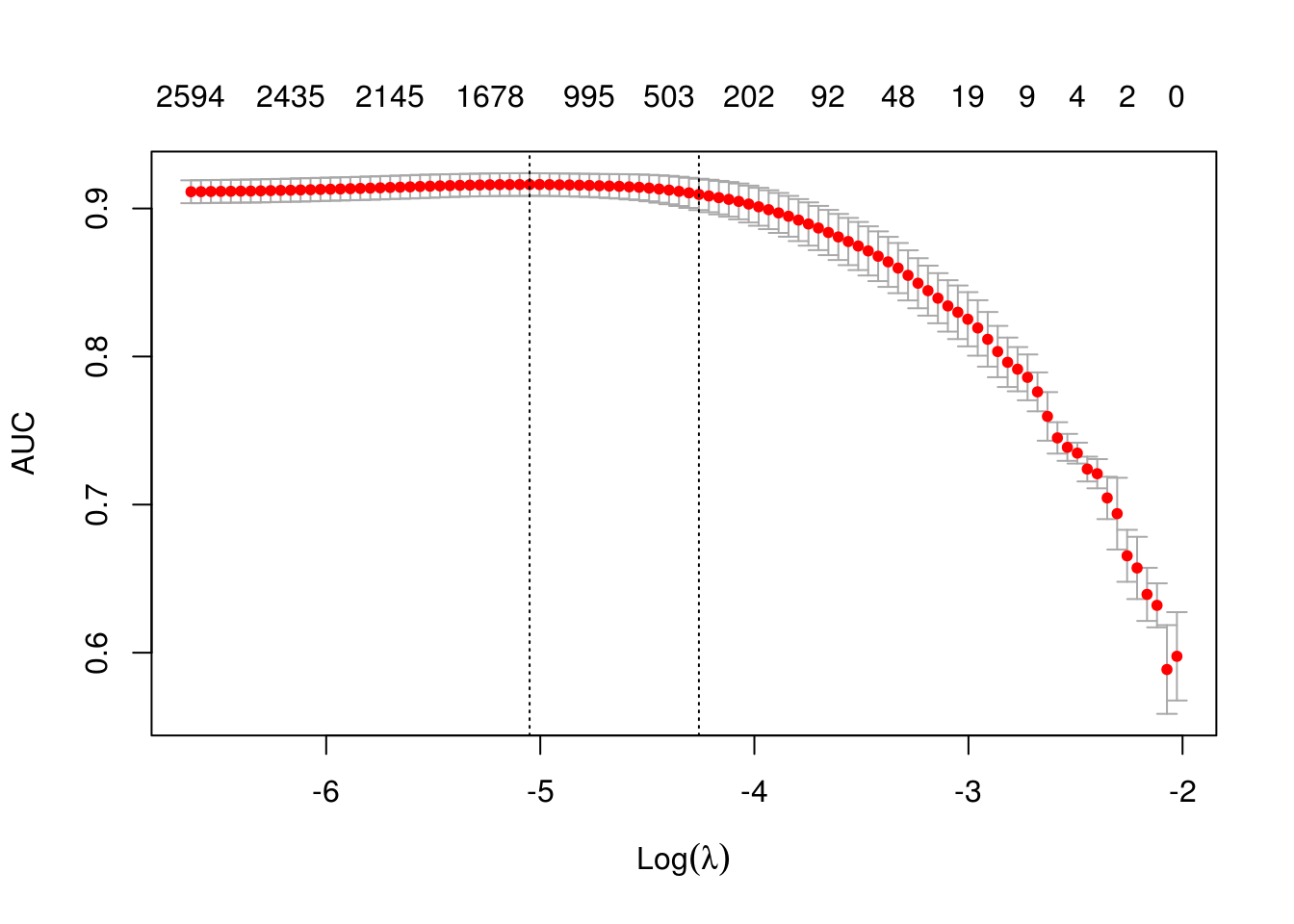
*# again lower number of iterations for faster training*

maxit = 1e3)

print(difftime(Sys.time(), t1, units = 'sec'))

## Time difference of 2.805214 secs

plot(glmnet\_classifier)

****

print(paste("max AUC =", round(max(glmnet\_classifier$cvm), 4)))

## [1] "max AUC = 0.9162"

We have successfully fit a model to our DTM. Now we can check the model’s performance on test data. Note that we use exactly the same functions from prepossessing and tokenization. Also we reuse/use the same vectorizer - function which maps terms to indices.

*# Note that most text2vec functions are pipe friendly!*

it\_test = tok\_fun(prep\_fun(test$review))

*# turn off progressbar because it won't look nice in rmd*

it\_test = itoken(it\_test, ids = test$id, progressbar = FALSE)

dtm\_test = create\_dtm(it\_test, vectorizer)

preds = predict(glmnet\_classifier, dtm\_test, type = 'response')[,1]

glmnet:::auc(test$sentiment, preds)

## [1] 0.9164517

As we can see, performance on the test data is roughly the same as we expect from cross-validation.

### **Pruning vocabulary**

We can note, however, that the training time for our model was quite high. We can reduce it and also significantly improve accuracy by pruning the vocabulary.

For example, we can find words “a”, “the”, “in”, “I”, “you”, “on”, etc in almost all documents, but they do not provide much useful information. Usually such words are called stop words. On the other hand, the corpus also contains very uncommon terms, which are contained in only a few documents. These terms are also useless, because we don’t have sufficient statistics for them. Here we will remove pre-defined stopwords, very common and very unusual terms.

stop\_words = c("i", "me", "my", "myself", "we", "our", "ours", "ourselves", "you", "your", "yours")

t1 = Sys.time()

vocab = create\_vocabulary(it\_train, stopwords = stop\_words)

print(difftime(Sys.time(), t1, units = 'sec'))

## Time difference of 0.3008301 secs

pruned\_vocab = prune\_vocabulary(vocab,

term\_count\_min = 10,

doc\_proportion\_max = 0.5,

doc\_proportion\_min = 0.001)

vectorizer = vocab\_vectorizer(pruned\_vocab)

*# create dtm\_train with new pruned vocabulary vectorizer*

t1 = Sys.time()

dtm\_train = create\_dtm(it\_train, vectorizer)

print(difftime(Sys.time(), t1, units = 'sec'))

## Time difference of 0.2928042 secs

dim(dtm\_train)

## [1] 4000 6542

Note that the new DTM has many fewer columns than the original DTM. This usually leads to both accuracy improvement (because we removed “noise”) and reduction of the training time.

Also we need to create DTM for test data with the same vectorizer:

dtm\_test = create\_dtm(it\_test, vectorizer)

dim(dtm\_test)

## [1] 1000 6542

## N-grams

Can we improve the model? Definitely - we can use n-grams instead of words. Here we will use up to 2-grams:

t1 = Sys.time()

vocab = create\_vocabulary(it\_train, ngram = c(1L, 2L))

print(difftime(Sys.time(), t1, units = 'sec'))

## Time difference of 1.311055 secs

vocab = prune\_vocabulary(vocab, term\_count\_min = 10,

doc\_proportion\_max = 0.5)

bigram\_vectorizer = vocab\_vectorizer(vocab)

dtm\_train = create\_dtm(it\_train, bigram\_vectorizer)

t1 = Sys.time()

glmnet\_classifier = cv.glmnet(x = dtm\_train, y = train[['sentiment']],

family = 'binomial',

alpha = 1,

type.measure = "auc",

nfolds = NFOLDS,

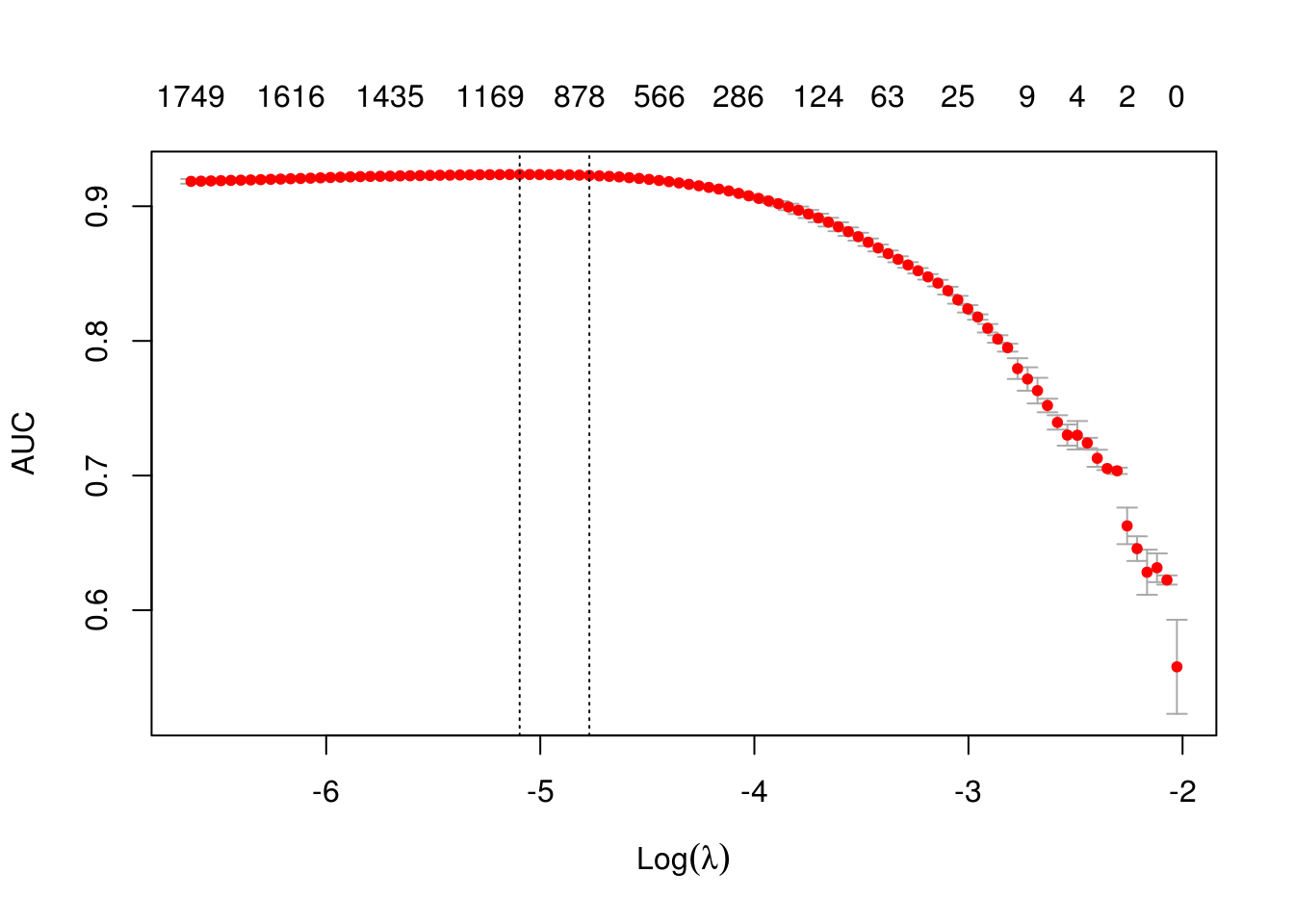
thresh = 1e-3,

maxit = 1e3)

print(difftime(Sys.time(), t1, units = 'sec'))

## Time difference of 2.535554 secs

plot(glmnet\_classifier)

****

print(paste("max AUC =", round(max(glmnet\_classifier$cvm), 4)))

## [1] "max AUC = 0.9236"

Seems that usage of n-grams improved our model a little bit more. Let’s check performance on test dataset:

*# apply vectorizer*

dtm\_test = create\_dtm(it\_test, bigram\_vectorizer)

preds = predict(glmnet\_classifier, dtm\_test, type = 'response')[,1]

glmnet:::auc(test$sentiment, preds)

## [1] 0.9294806

Further tuning is left up to the reader.

## Feature hashing

If you are not familiar with feature hashing (the so-called “hashing trick”) I recommend you start with the Wikipedia article, then read the original paper by a Yahoo! research team. This technique is very fast because we don’t have to perform a lookup over an associative array. Another benefit is that it leads to a very low memory footprint, since we can map an arbitrary number of features into much more compact space. This method was popularized by Yahoo! and is widely used in Vowpal Wabbit.

Here is how to use feature hashing in text2vec.

h\_vectorizer = hash\_vectorizer(hash\_size = 2 ^ 14, ngram = c(1L, 2L))

t1 = Sys.time()

dtm\_train = create\_dtm(it\_train, h\_vectorizer)

print(difftime(Sys.time(), t1, units = 'sec'))

## Time difference of 0.883918 secs

t1 = Sys.time()

glmnet\_classifier = cv.glmnet(x = dtm\_train, y = train[['sentiment']],

family = 'binomial',

alpha = 1,

type.measure = "auc",

nfolds = 5,

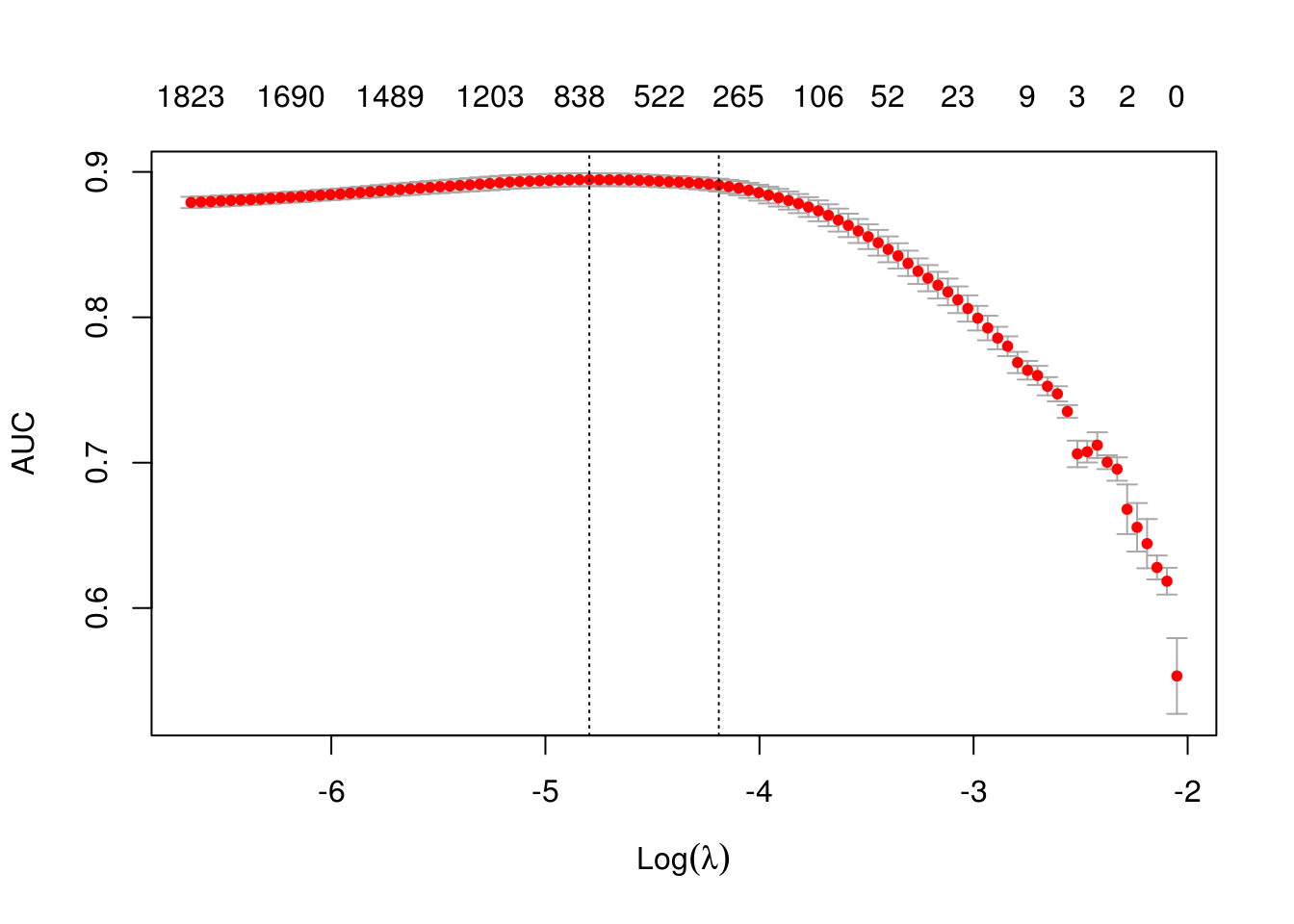
thresh = 1e-3,

maxit = 1e3)

print(difftime(Sys.time(), t1, units = 'sec'))

## Time difference of 4.401476 secs

plot(glmnet\_classifier)

****

print(paste("max AUC =", round(max(glmnet\_classifier$cvm), 4)))

## [1] "max AUC = 0.8947"

dtm\_test = create\_dtm(it\_test, h\_vectorizer)

preds = predict(glmnet\_classifier, dtm\_test , type = 'response')[, 1]

glmnet:::auc(test$sentiment, preds)

## [1] 0.898901

As you can see our AUC is a bit worse but DTM construction time is considerably lower. On large collections of documents this can be a significant advantage.

# **Basic transformations**

Before doing analysis it usually can be useful to transform DTM. For example lengths of the documents in collection can significantly vary. In this case it can be useful to apply normalization.

## Normalization

By “normalization” we assume transformation of the rows of DTM so we adjust values measured on different scales to a notionally common scale. For the case when length of the documents vary we can apply “L1” normalization. It means we will transform rows in a way that sum of the row values will be equal to 1:

dtm\_train\_l1\_norm = normalize(dtm\_train, "l1")

By this transformation we should improve the quality of data preparation.

## TF-IDF

Another popular technique is TF-IDF transformation. We can (and usually should) apply it to our DTM. It will not only normalize DTM, but also increase the weight of terms which are specific to a single document or handful of documents and decrease the weight for terms used in most documents:

vocab = create\_vocabulary(it\_train)

vectorizer = vocab\_vectorizer(vocab)

dtm\_train = create\_dtm(it\_train, vectorizer)

*# define tfidf model*

tfidf = TfIdf$new()

*# fit model to train data and transform train data with fitted model*

dtm\_train\_tfidf = fit\_transform(dtm\_train, tfidf)

*# tfidf modified by fit\_transform() call!*

*# apply pre-trained tf-idf transformation to test data*

dtm\_test\_tfidf = create\_dtm(it\_test, vectorizer)

dtm\_test\_tfidf = transform(dtm\_test\_tfidf, tfidf)

Note that here we first time touched model object in text2vec. At this moment the user should remember several important things about text2vec models:

1. Models can be fitted on a given data (train) and applied to unseen data (test)
2. **Models are mutable** - once you will pass model to fit() or fit\_transform() function, model will be modifed by it.
3. After model is fitted, it can be applied to a new data with transform(new\_data, fitted\_model) method.

More detailed overview of models and models API will be available soon in a separate vignette.

Once we have tf-idf reweighted DTM we can fit our linear classifier again:

t1 = Sys.time()

glmnet\_classifier = cv.glmnet(x = dtm\_train\_tfidf, y = train[['sentiment']],

family = 'binomial',

alpha = 1,

type.measure = "auc",

nfolds = NFOLDS,

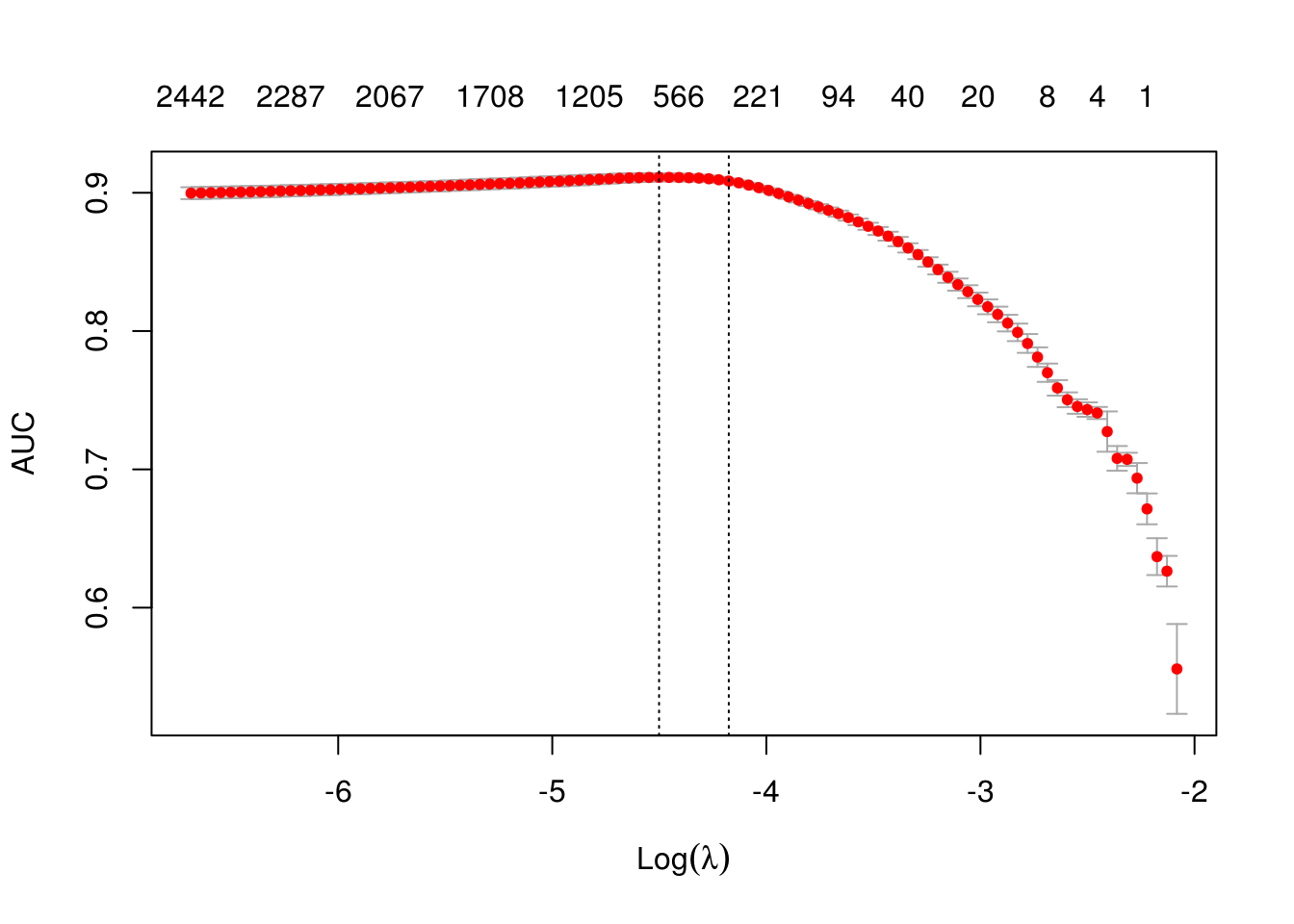
thresh = 1e-3,

maxit = 1e3)

print(difftime(Sys.time(), t1, units = 'sec'))

## Time difference of 2.561174 secs

plot(glmnet\_classifier)

****

print(paste("max AUC =", round(max(glmnet\_classifier$cvm), 4)))

## [1] "max AUC = 0.9111"

Let’s check the model performance on the test dataset:

preds = predict(glmnet\_classifier, dtm\_test\_tfidf, type = 'response')[,1]

glmnet:::auc(test$sentiment, preds)

## [1] 0.9119419

Usually tf-idf transformation **significantly** improve performance on most of the dowstream tasks.

Alternatives to text2vec would be tm + SnowballC or you could work with the tidytext package.

The itoken() function creates vocabularies (here stemmed words), from which we can create the dtm with the create\_dtm() function.

All preprocessing steps, starting from the raw text, need to be wrapped in a function that can then be pasted into the lime::lime() function; this is only necessary if you want to use your model with lime.

get\_matrix <- function(text) {

it <- itoken(text, progressbar = FALSE)

create\_dtm(it, vectorizer = hash\_vectorizer())

}

Now, this preprocessing function can be applied to both training and test data.

dtm\_train <- get\_matrix(clothing\_reviews\_train$text)

str(dtm\_train)

## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots

## ..@ i : int [1:889012] 304 764 786 788 793 794 1228 2799 2819 3041 ...

## ..@ p : int [1:262145] 0 0 0 0 0 0 0 0 0 0 ...

## ..@ Dim : int [1:2] 18789 262144

## ..@ Dimnames:List of 2

## .. ..$ : chr [1:18789] "1" "2" "3" "4" ...

## .. ..$ : NULL

## ..@ x : num [1:889012] 1 1 2 1 2 1 1 1 1 1 ...

## ..@ factors : list()

dtm\_test <- get\_matrix(clothing\_reviews\_test$text)

str(dtm\_test)

## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots

## ..@ i : int [1:222314] 2793 400 477 622 2818 2997 3000 4500 3524 2496 ...

## ..@ p : int [1:262145] 0 0 0 0 0 0 0 0 0 0 ...

## ..@ Dim : int [1:2] 4697 262144

## ..@ Dimnames:List of 2

## .. ..$ : chr [1:4697] "1" "2" "3" "4" ...

## .. ..$ : NULL

## ..@ x : num [1:222314] 1 1 1 1 1 1 1 1 1 1 ...

## ..@ factors : list()

And we use it to train a model with the xgboost package (just as in the example of the lime package).

xgb\_model <- xgb.train(list(max\_depth = 7,

eta = 0.1,

objective = "binary:logistic",

eval\_metric = "error", nthread = 1),

xgb.DMatrix(dtm\_train,

label = clothing\_reviews\_train$Liked == "1"),

nrounds = 50)

Let’s try it on the test data and see how it performs:

pred <- predict(xgb\_model, dtm\_test)

confusionMatrix(clothing\_reviews\_test$Liked,

as.factor(round(pred, digits = 0)))

## Confusion Matrix and Statistics

##

## Reference

## Prediction 0 1

## 0 1370 701

## 1 421 2205

##

## Accuracy : 0.7611

## 95% CI : (0.7487, 0.7733)

## No Information Rate : 0.6187

## P-Value [Acc > NIR] : < 2.2e-16

##

## Kappa : 0.5085

## Mcnemar's Test P-Value : < 2.2e-16

##

## Sensitivity : 0.7649

## Specificity : 0.7588

## Pos Pred Value : 0.6615

## Neg Pred Value : 0.8397

## Prevalence : 0.3813

## Detection Rate : 0.2917

## Detection Prevalence : 0.4409

## Balanced Accuracy : 0.7619

##

## 'Positive' Class : 0

##

Okay, not a perfect score but good enough for me – right now, I’m more interested in the explanations of the model’s predictions. For this, we need to run the lime() function and give it

* the text input that was used to construct the model
* the trained model
* the preprocessing function

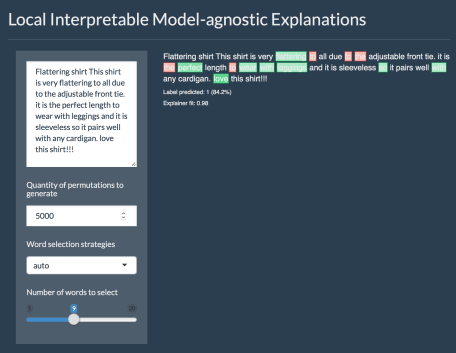
explainer <- lime(clothing\_reviews\_train$text,

xgb\_model,

preprocess = get\_matrix)

With this, we could right away call the interactive explainer Shiny app, where we can type any text we want into the field on the left and see the explanation on the right: words that are underlined green support the classification, red words contradict them.

interactive\_text\_explanations(explainer)



What happens in the background in the app, we can do explicitly by calling the explain() function and give it

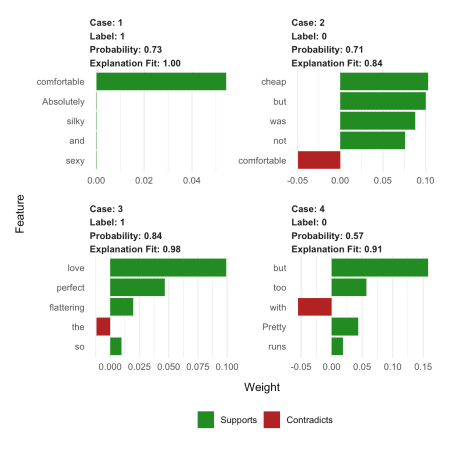
* the test data (here the first four reviews of the test set)
* the explainer defined with the lime() function
* the number of labels we want to have explanations for (alternatively, you set the label by name)
* and the number of features (in this case words) that should be included in the explanations

We can plot them either with the plot\_text\_explanations() function, which gives an output like in the Shiny app or we use the regular plot\_features() function.

explanations <- lime::explain(clothing\_reviews\_test$text[1:4], explainer, n\_labels = 1, n\_features = 5)

plot\_text\_explanations(explanations)

plot\_features(explanations)



As we can see, our explanations contain a lot of stop-words that don’t really make much sense as features in our model. So…

**… let’s try a more complex example**

Okay, our model above works but there are still common words and stop words in our model that LIME picks up on. Ideally, we would want to remove them before modeling and keep only relevant words. This we can accomplish by using additional steps and options in our preprocessing function.

Important to know is that whatever preprocessing we do with our text corpus, train and test data has to have the same features (i.e. words)! If we were to incorporate all the steps shown below into one function and call it separately on train and test data, we would end up with different words in our dtm and the predict() function won’t work any more. In the simple example above, it works because we have been using the hash\_vectorizer().

Nevertheless, the lime::explain() function expects a preprocessing function that takes a character vector as input.

How do we go about this? First, we will need to create the vocabulary just from the training data. To reduce the number of words to only the most relevant I am performing the following steps:

* stem all words
* remove step-words
* prune vocabulary
* transform into vector space

stem\_tokenizer <- function(x) {

lapply(word\_tokenizer(x),

SnowballC::wordStem,

language = "en")

}

stop\_words = tm::stopwords(kind = "en")

# create prunded vocabulary

vocab\_train <- itoken(clothing\_reviews\_train$text,

preprocess\_function = tolower,

tokenizer = stem\_tokenizer,

progressbar = FALSE)

v <- create\_vocabulary(vocab\_train,

stopwords = stop\_words)

pruned\_vocab <- prune\_vocabulary(v,

doc\_proportion\_max = 0.99,

doc\_proportion\_min = 0.01)

vectorizer\_train <- vocab\_vectorizer(pruned\_vocab)

This vector space can now be added to the preprocessing function, which we can then apply to both train and test data. Here, I am also transforming the word counts to [tfidf](https://en.wikipedia.org/wiki/Tf%E2%80%93idf) values.

# preprocessing function

create\_dtm\_mat <- function(text, vectorizer = vectorizer\_train) {

vocab <- itoken(text,

preprocess\_function = tolower,

tokenizer = stem\_tokenizer,

progressbar = FALSE)

dtm <- create\_dtm(vocab,

vectorizer = vectorizer)

tfidf = TfIdf$new()

fit\_transform(dtm, tfidf)

}

dtm\_train2 <- create\_dtm\_mat(clothing\_reviews\_train$text)

str(dtm\_train2)

## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots

## ..@ i : int [1:415770] 26 74 169 294 588 693 703 708 727 759 ...

## ..@ p : int [1:506] 0 189 380 574 765 955 1151 1348 1547 1740 ...

## ..@ Dim : int [1:2] 18789 505

## ..@ Dimnames:List of 2

## .. ..$ : chr [1:18789] "1" "2" "3" "4" ...

## .. ..$ : chr [1:505] "ad" "sandal" "depend" "often" ...

## ..@ x : num [1:415770] 0.177 0.135 0.121 0.17 0.131 ...

## ..@ factors : list()

dtm\_test2 <- create\_dtm\_mat(clothing\_reviews\_test$text)

str(dtm\_test2)

## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots

## ..@ i : int [1:103487] 228 304 360 406 472 518 522 624 732 784 ...

## ..@ p : int [1:506] 0 53 113 151 186 216 252 290 323 360 ...

## ..@ Dim : int [1:2] 4697 505

## ..@ Dimnames:List of 2

## .. ..$ : chr [1:4697] "1" "2" "3" "4" ...

## .. ..$ : chr [1:505] "ad" "sandal" "depend" "often" ...

## ..@ x : num [1:103487] 0.263 0.131 0.135 0.109 0.179 ...

## ..@ factors : list()

And we will train another gradient boosting model:

xgb\_model2 <- xgb.train(params = list(max\_depth = 10,

eta = 0.2,

objective = "binary:logistic",

eval\_metric = "error", nthread = 1),

data = xgb.DMatrix(dtm\_train2,

label = clothing\_reviews\_train$Liked == "1"),

nrounds = 500)

pred2 <- predict(xgb\_model2, dtm\_test2)

confusionMatrix(clothing\_reviews\_test$Liked,

as.factor(round(pred2, digits = 0)))

## Confusion Matrix and Statistics

##

## Reference

## Prediction 0 1

## 0 1441 630

## 1 426 2200

##

## Accuracy : 0.7752

## 95% CI : (0.763, 0.787)

## No Information Rate : 0.6025

## P-Value [Acc > NIR] : < 2.2e-16

##

## Kappa : 0.5392

## Mcnemar's Test P-Value : 4.187e-10

##

## Sensitivity : 0.7718

## Specificity : 0.7774

## Pos Pred Value : 0.6958

## Neg Pred Value : 0.8378

## Prevalence : 0.3975

## Detection Rate : 0.3068

## Detection Prevalence : 0.4409

## Balanced Accuracy : 0.7746

##

## 'Positive' Class : 0

##

Unfortunately, this didn’t really improve the classification accuracy but let’s look at the explanations again:

explainer2 <- lime(clothing\_reviews\_train$text,

xgb\_model2,

preprocess = create\_dtm\_mat)

explanations2 <- lime::explain(clothing\_reviews\_test$text[1:4], explainer2, n\_labels = 1, n\_features = 4)

plot\_text\_explanations(explanations2)

The words that get picked up now make much more sense! So, even though making my model more complex didn’t improve “the numbers”, this second model is likely to be much better able to generalize to new reviews because it seems to pick up on words that make intuitive sense.

That’s why I’m sold on the benefits of adding explainer functions to most machine learning workflows – and why I love the lime package in R!

Here is the full code for the third Part

# data wrangling

library(tidyverse)

library(readr)

# plotting

library(ggthemes)

theme\_set(theme\_minimal())

# text prep

library(text2vec)

# ml

library(caret)

library(xgboost)

# explanation

library(lime)

**Text classification models**

Here I am using another Kaggle dataset: Women’s e-commerce cloting reviews. The data contains a text review of different items of clothing, as well as some additional information, like rating, division, etc.

In this example, I will use the review title and text in order to classify whether or not the item was liked. I am creating the response variable from the rating: every item rates with 5 stars is considered “liked” (1), the rest as “not liked” (0). I am also combining review title and text.

clothing\_reviews <- read\_csv("/Users/shiringlander/Documents/Github/ix\_lime\_etc/Womens Clothing E-Commerce Reviews.csv") %>%

mutate(Liked = as.factor(ifelse(Rating == 5, 1, 0)),

text = paste(Title, `Review Text`),

text = gsub("NA", "", text))

## Parsed with column specification:

## cols(

## X1 = col\_integer(),

## `Clothing ID` = col\_integer(),

## Age = col\_integer(),

## Title = col\_character(),

## `Review Text` = col\_character(),

## Rating = col\_integer(),

## `Recommended IND` = col\_integer(),

## `Positive Feedback Count` = col\_integer(),

## `Division Name` = col\_character(),

## `Department Name` = col\_character(),

## `Class Name` = col\_character()

## )

glimpse(clothing\_reviews)

## Observations: 23,486

## Variables: 13

## $ X1 <int> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11...

## $ `Clothing ID` <int> 767, 1080, 1077, 1049, 847, 1080, 85...

## $ Age <int> 33, 34, 60, 50, 47, 49, 39, 39, 24, ...

## $ Title <chr> NA, NA, "Some major design flaws", "...

## $ `Review Text` <chr> "Absolutely wonderful - silky and se...

## $ Rating <int> 4, 5, 3, 5, 5, 2, 5, 4, 5, 5, 3, 5, ...

## $ `Recommended IND` <int> 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, ...

## $ `Positive Feedback Count` <int> 0, 4, 0, 0, 6, 4, 1, 4, 0, 0, 14, 2,...

## $ `Division Name` <chr> "Initmates", "General", "General", "...

## $ `Department Name` <chr> "Intimate", "Dresses", "Dresses", "B...

## $ `Class Name` <chr> "Intimates", "Dresses", "Dresses", "...

## $ Liked <fct> 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, ...

## $ text <chr> " Absolutely wonderful - silky and s...

Whether an item was liked or not will thus be my response variable or label for classification.

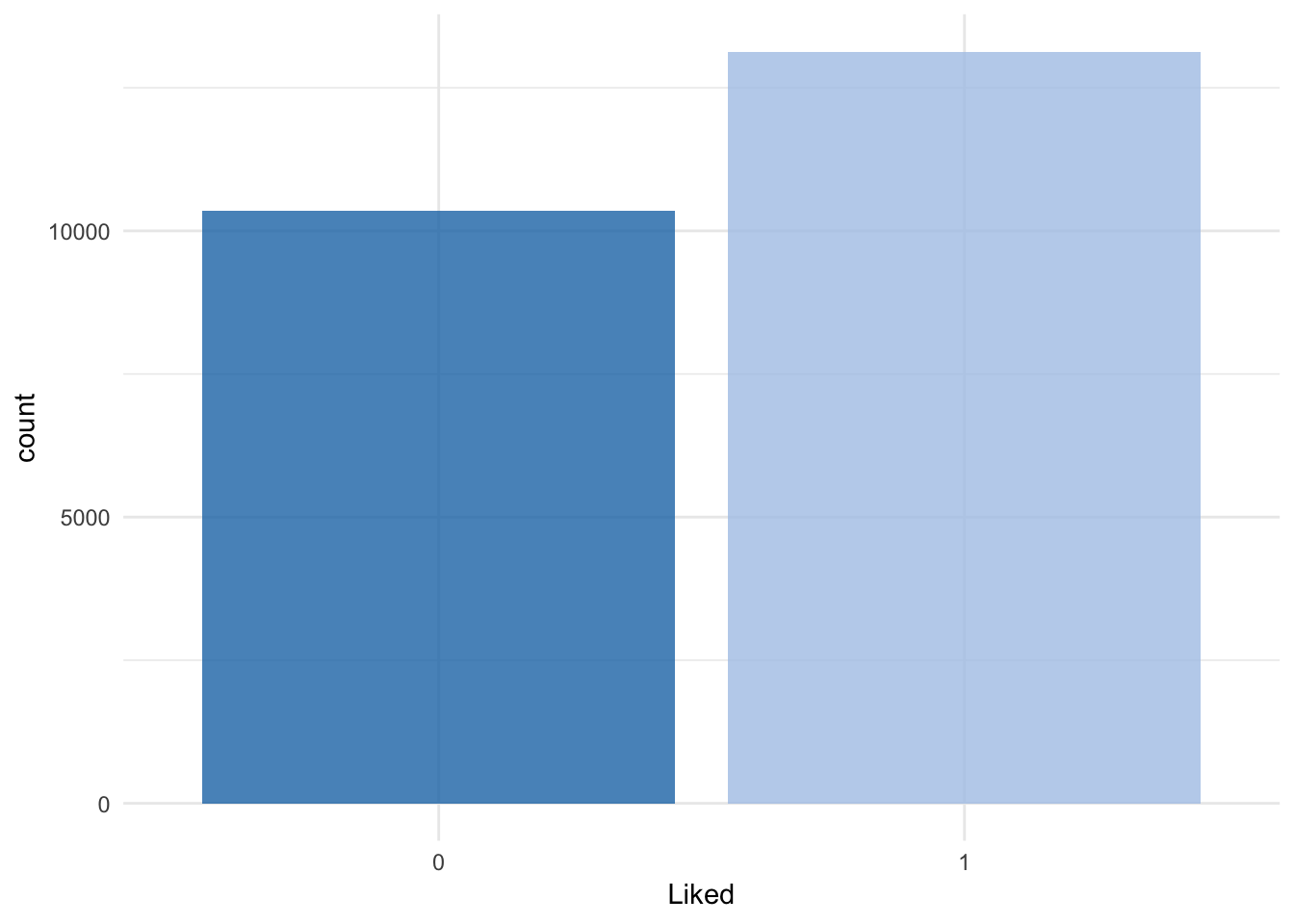
clothing\_reviews %>%

ggplot(aes(x = Liked, fill = Liked)) +

geom\_bar(alpha = 0.8) +

scale\_fill\_tableau(palette = "tableau20") +

guides(fill = FALSE)



Let’s split the data into train and test sets:

set.seed(42)

idx <- createDataPartition(clothing\_reviews$Liked,

p = 0.8,

list = FALSE,

times = 1)

clothing\_reviews\_train <- clothing\_reviews[ idx,]

clothing\_reviews\_test <- clothing\_reviews[-idx,]

**Let’s start simple**

The first text model I’m looking at has been built similarly to the example model in the help for lime::interactive\_text\_explanations().

First, we need to prepare the data for modeling: we will need to convert the text to a document term matrix (dtm). There are different ways to do this. One is be with the text2vec package.

*“Because of R’s copy-on-modify semantics, it is not easy to iteratively grow a DTM. Thus constructing a DTM, even for a small collections of documents, can be a serious bottleneck for analysts and researchers. It involves reading the whole collection of text documents into RAM and processing it as single vector, which can easily increase memory use by a factor of 2 to 4. The text2vec package solves this problem by providing a better way of constructing a document-term matrix.” https://cran.r-project.org/web/packages/text2vec/vignettes/text-vectorization.html*

Alternatives to text2vec would be tm + SnowballC or you could work with the tidytext package.

The itoken() function creates vocabularies (here stemmed words), from which we can create the dtm with the create\_dtm() function.

All preprocessing steps, starting from the raw text, need to be wrapped in a function that can then be pasted into the lime::lime() function; this is only necessary if you want to use your model with lime.

get\_matrix <- function(text) {

it <- itoken(text, progressbar = FALSE)

create\_dtm(it, vectorizer = hash\_vectorizer())

}

Now, this preprocessing function can be applied to both training and test data.

dtm\_train <- get\_matrix(clothing\_reviews\_train$text)

str(dtm\_train)

## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots

## ..@ i : int [1:889012] 304 764 786 788 793 794 1228 2799 2819 3041 ...

## ..@ p : int [1:262145] 0 0 0 0 0 0 0 0 0 0 ...

## ..@ Dim : int [1:2] 18789 262144

## ..@ Dimnames:List of 2

## .. ..$ : chr [1:18789] "1" "2" "3" "4" ...

## .. ..$ : NULL

## ..@ x : num [1:889012] 1 1 2 1 2 1 1 1 1 1 ...

## ..@ factors : list()

dtm\_test <- get\_matrix(clothing\_reviews\_test$text)

str(dtm\_test)

## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots

## ..@ i : int [1:222314] 2793 400 477 622 2818 2997 3000 4500 3524 2496 ...

## ..@ p : int [1:262145] 0 0 0 0 0 0 0 0 0 0 ...

## ..@ Dim : int [1:2] 4697 262144

## ..@ Dimnames:List of 2

## .. ..$ : chr [1:4697] "1" "2" "3" "4" ...

## .. ..$ : NULL

## ..@ x : num [1:222314] 1 1 1 1 1 1 1 1 1 1 ...

## ..@ factors : list()

And we use it to train a model with the xgboost package (just as in the example of the lime package).

xgb\_model <- xgb.train(list(max\_depth = 7,

eta = 0.1,

objective = "binary:logistic",

eval\_metric = "error", nthread = 1),

xgb.DMatrix(dtm\_train,

label = clothing\_reviews\_train$Liked == "1"),

nrounds = 50)

Let’s try it on the test data and see how it performs:

pred <- predict(xgb\_model, dtm\_test)

confusionMatrix(clothing\_reviews\_test$Liked,

as.factor(round(pred, digits = 0)))

## Confusion Matrix and Statistics

##

## Reference

## Prediction 0 1

## 0 1370 701

## 1 421 2205

##

## Accuracy : 0.7611

## 95% CI : (0.7487, 0.7733)

## No Information Rate : 0.6187

## P-Value [Acc > NIR] : < 2.2e-16

##

## Kappa : 0.5085

## Mcnemar's Test P-Value : < 2.2e-16

##

## Sensitivity : 0.7649

## Specificity : 0.7588

## Pos Pred Value : 0.6615

## Neg Pred Value : 0.8397

## Prevalence : 0.3813

## Detection Rate : 0.2917

## Detection Prevalence : 0.4409

## Balanced Accuracy : 0.7619

##

## 'Positive' Class : 0

##

Okay, not a perfect score but good enough for me - right now, I’m more interested in the explanations of the model’s predictions. For this, we need to run the lime() function and give it

* the text input that was used to construct the model
* the trained model
* the preprocessing function

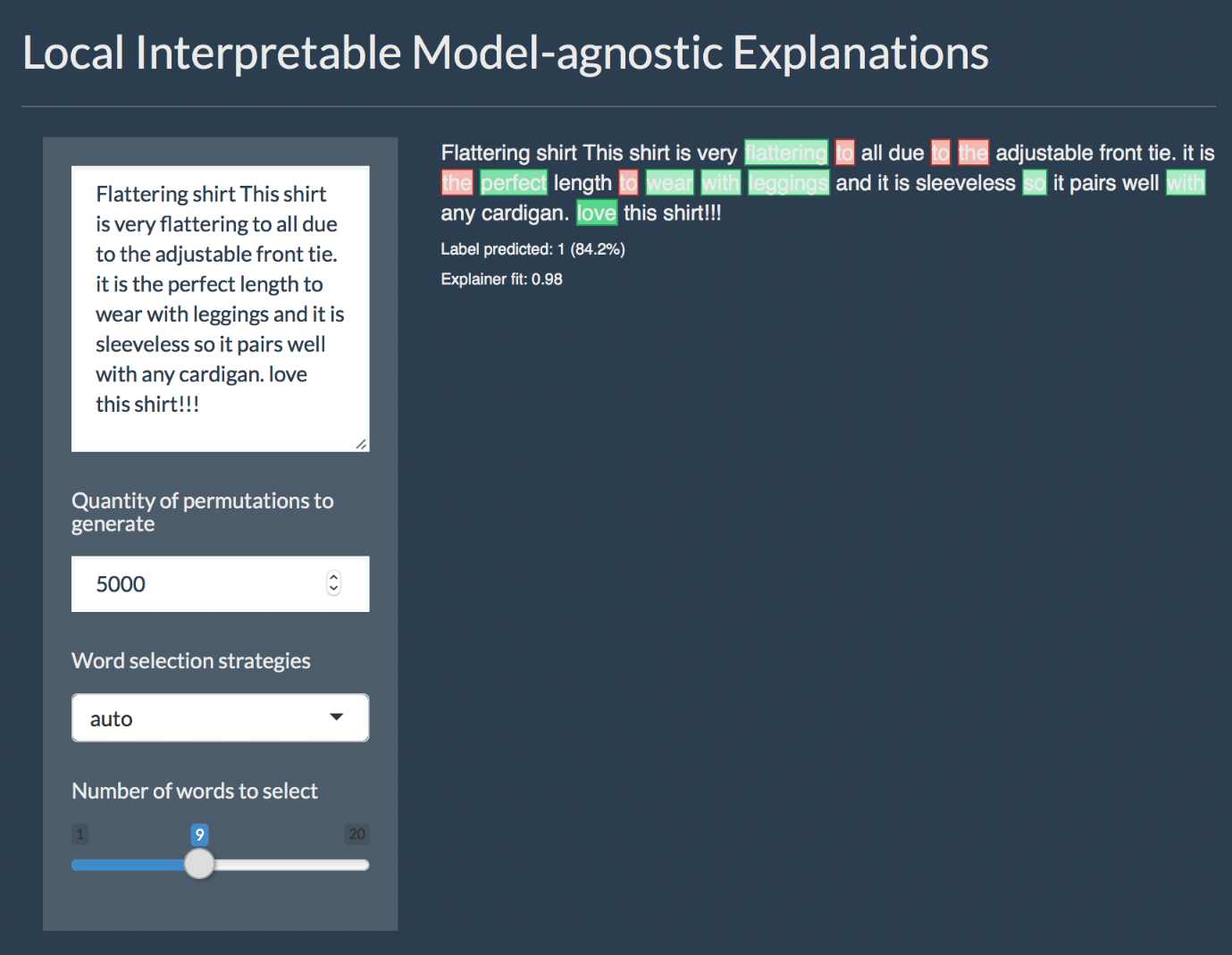
explainer <- lime(clothing\_reviews\_train$text,

xgb\_model,

preprocess = get\_matrix)

With this, we could right away call the interactive explainer Shiny app, where we can type any text we want into the field on the left and see the explanation on the right: words that are underlined green support the classification, red words contradict them.

interactive\_text\_explanations(explainer)



What happens in the background in the app, we can do explicitly by calling the explain() function and give it

* the test data (here the first four reviews of the test set)
* the explainer defined with the lime() function
* the number of labels we want to have explanations for (alternatively, you set the label by name)
* and the number of features (in this case words) that should be included in the explanations

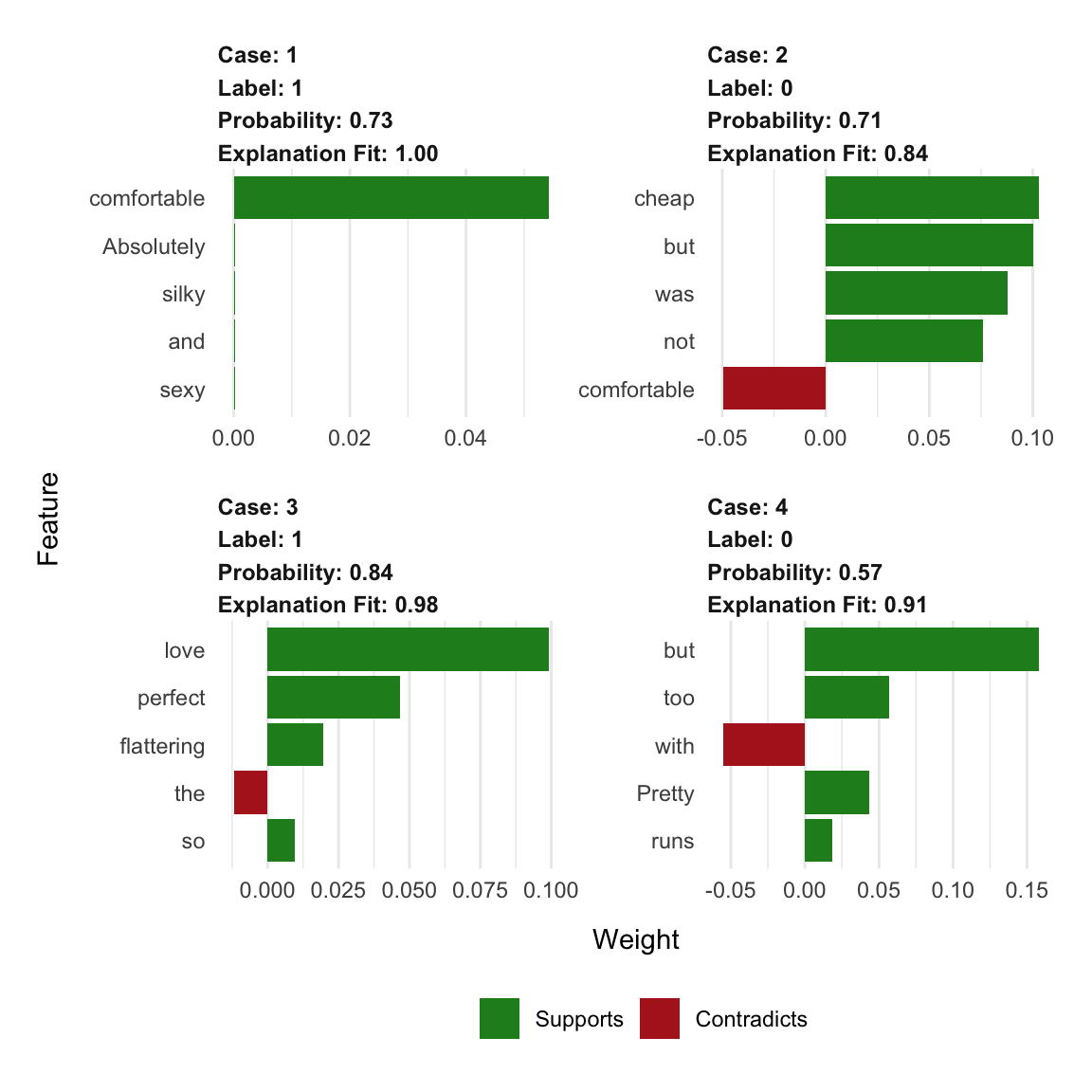
We can plot them either with the plot\_text\_explanations() function, which gives an output like in the Shiny app or we use the regular plot\_features() function.

explanations <- lime::explain(clothing\_reviews\_test$text[1:4], explainer, n\_labels = 1, n\_features = 5)

plot\_text\_explanations(explanations)



plot\_features(explanations)



As we can see, our explanations contain a lot of stop-words that don’t really make much sense as features in our model. So…

**… let’s try a more complex example**

Okay, our model above works but there are still common words and stop words in our model that LIME picks up on. Ideally, we would want to remove them before modeling and keep only relevant words. This we can accomplish by using additional steps and options in our preprocessing function.

Important to know is that whatever preprocessing we do with our text corpus, train and test data has to have the same features (i.e. words)! If we were to incorporate all the steps shown below into one function and call it separately on train and test data, we would end up with different words in our dtm and the predict() function won’t work any more. In the simple example above, it works because we have been using the hash\_vectorizer().

Nevertheless, the lime::explain() function expects a preprocessing function that takes a character vector as input.

How do we go about this? First, we will need to create the vocabulary just from the training data. To reduce the number of words to only the most relevant I am performing the following steps:

* stem all words
* remove step-words
* prune vocabulary
* transform into vector space

stem\_tokenizer <- function(x) {

lapply(word\_tokenizer(x),

SnowballC::wordStem,

language = "en")

}

stop\_words = tm::stopwords(kind = "en")

# create prunded vocabulary

vocab\_train <- itoken(clothing\_reviews\_train$text,

preprocess\_function = tolower,

tokenizer = stem\_tokenizer,

progressbar = FALSE)

v <- create\_vocabulary(vocab\_train,

stopwords = stop\_words)

pruned\_vocab <- prune\_vocabulary(v,

doc\_proportion\_max = 0.99,

doc\_proportion\_min = 0.01)

vectorizer\_train <- vocab\_vectorizer(pruned\_vocab)

This vector space can now be added to the preprocessing function, which we can then apply to both train and test data. Here, I am also transforming the word counts to tfidf values.

# preprocessing function

create\_dtm\_mat <- function(text, vectorizer = vectorizer\_train) {

vocab <- itoken(text,

preprocess\_function = tolower,

tokenizer = stem\_tokenizer,

progressbar = FALSE)

dtm <- create\_dtm(vocab,

vectorizer = vectorizer)

tfidf = TfIdf$new()

fit\_transform(dtm, tfidf)

}

dtm\_train2 <- create\_dtm\_mat(clothing\_reviews\_train$text)

str(dtm\_train2)

## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots

## ..@ i : int [1:415770] 26 74 169 294 588 693 703 708 727 759 ...

## ..@ p : int [1:506] 0 189 380 574 765 955 1151 1348 1547 1740 ...

## ..@ Dim : int [1:2] 18789 505

## ..@ Dimnames:List of 2

## .. ..$ : chr [1:18789] "1" "2" "3" "4" ...

## .. ..$ : chr [1:505] "ad" "sandal" "depend" "often" ...

## ..@ x : num [1:415770] 0.177 0.135 0.121 0.17 0.131 ...

## ..@ factors : list()

dtm\_test2 <- create\_dtm\_mat(clothing\_reviews\_test$text)

str(dtm\_test2)

## Formal class 'dgCMatrix' [package "Matrix"] with 6 slots

## ..@ i : int [1:103487] 228 304 360 406 472 518 522 624 732 784 ...

## ..@ p : int [1:506] 0 53 113 151 186 216 252 290 323 360 ...

## ..@ Dim : int [1:2] 4697 505

## ..@ Dimnames:List of 2

## .. ..$ : chr [1:4697] "1" "2" "3" "4" ...

## .. ..$ : chr [1:505] "ad" "sandal" "depend" "often" ...

## ..@ x : num [1:103487] 0.263 0.131 0.135 0.109 0.179 ...

## ..@ factors : list()

And we will train another gradient boosting model:

xgb\_model2 <- xgb.train(params = list(max\_depth = 10,

eta = 0.2,

objective = "binary:logistic",

eval\_metric = "error", nthread = 1),

data = xgb.DMatrix(dtm\_train2,

label = clothing\_reviews\_train$Liked == "1"),

nrounds = 500)

pred2 <- predict(xgb\_model2, dtm\_test2)

confusionMatrix(clothing\_reviews\_test$Liked,

as.factor(round(pred2, digits = 0)))

## Confusion Matrix and Statistics

##

## Reference

## Prediction 0 1

## 0 1441 630

## 1 426 2200

##

## Accuracy : 0.7752

## 95% CI : (0.763, 0.787)

## No Information Rate : 0.6025

## P-Value [Acc > NIR] : < 2.2e-16

##

## Kappa : 0.5392

## Mcnemar's Test P-Value : 4.187e-10

##

## Sensitivity : 0.7718

## Specificity : 0.7774

## Pos Pred Value : 0.6958

## Neg Pred Value : 0.8378

## Prevalence : 0.3975

## Detection Rate : 0.3068

## Detection Prevalence : 0.4409

## Balanced Accuracy : 0.7746

##

## 'Positive' Class : 0

##

Unfortunately, this didn’t really improve the classification accuracy but let’s look at the explanations again:

explainer2 <- lime(clothing\_reviews\_train$text,

xgb\_model2,

preprocess = create\_dtm\_mat)

explanations2 <- lime::explain(clothing\_reviews\_test$text[1:4], explainer2, n\_labels = 1, n\_features = 4)

plot\_text\_explanations(explanations2)



The words that get picked up now make much more sense! So, even though making my model more complex didn’t improve “the numbers”, this second model is likely to be much better able to generalize to new reviews because it seems to pick up on words that make intuitive sense.

That’s why I’m sold on the benefits of adding explainer functions to most machine learning workflows - and why I love the lime package in R!