In this post we will return to the Pitchfork music review data. Our goal here will be to use text mining and natural language processing (NLP) to understand linguistic signals of album quality. This type of analysis helps us understand what Pitchfork reviewers appreciate or dislike, and gives us a sense of the criteria which distinguish good albums from bad ones. We will use the R package **Quanteda**, developed by Ken Benoit and colleagues, to do the text mining and NLP. We will use the **glmnet** package to build a LASSO regression model to predict the album review score from the review text.

**The Data**

These data were obtained from the [Kaggle website](https://www.kaggle.com/nolanbconaway/pitchfork-data). In this analysis, we will use all 18,389 unique reviews in the data. For the scope of this analysis, we are only going to be interested in a few pieces of information. The first is the text of the review, which is contained in a column called “content.” The second is the score (from 0 to 10) that was assigned to the album by the Pitchfork reviewer, contained in a column called “score\_to\_predict”. And the final columns are dummy variables which indicate the genre of the album being reviewed (with the following 9 options: *electronic*, *experimental*, *folk/country*, *global*, *jazz*, *metal*, *pop/rnb*, *rap* and *rock*). Our dataset is called “*reviews*,” and the first 5 rows are shown below (not all dummy genre variables shown due to space considerations):

|  | **content** | **score\_to\_predict** | **genre\_electronic** | **genre\_rock** |
| --- | --- | --- | --- | --- |
| 1 | “Trip-hop” eventually became a ’90s punchline, a music-press shorthand for “overhyped hotel lounge music”… | 9.3 | 1 | 0 |
| 2 | Eight years, five albums, and two EPs in, the New York-based outfit Krallice have long since shut up purists about their “hipster black metal”… | 7.9 | 0 | 0 |
| 3 | Minneapolis’ Uranium Club seem to revel in being aggressively obtuse… | 7.3 | 0 | 1 |
| 4 | Kleenex began with a crash It transpired one night not long after they’d formed, in Zurich of 1978, while the germinal punk group was onstage… | 9 | 0 | 1 |
| 5 | It is impossible to consider a given release by a footwork artist without confronting the long shadow cast by DJ Rashad’s catalog… | 8.1 | 1 | 0 |
| 6 | In the pilot episode of “Insecure,” the critically lauded HBO comedy series created by Issa Rae and Larry Wilmore, Rae’s eponymous character Issa is at a crossroads… | 7.4 | 0 | 0 |

**Data Preparation**

Our data preparation will consist of four main steps, following the logic of the Quanteda package. We will first turn our data frame into a corpus object, which contains our texts and related meta-data (e.g. the other information in our data frame). We then create a tokens object from the corpus. The tokens object contains the featurized text, with each text stored as a list of character vectors. The characters are simply the individual elements that make up the texts (words, numbers, etc.). We will then turn our tokens object into a document-feature matrix (DFM). The DFM represents frequencies of features in documents in a matrix. We then convert the DFM to a matrix, adding in additional features we’ll use in our analysis. This matrix will serve as input for our model of album review score.

A schematic overview of the data preparation steps is thus:

*Data Frame -> Corpus -> Tokens Object -> DFM -> Model Matrix*

*Step 1: Turn our Data Frame Into a Corpus.*

In Quanteda, the basic representation of text data is the corpus. The idea of the corpus is that it is a general repository of the text data and additional meta-data that describes the corpus as a whole and the individual documents.. In the Quanteda way of doing things, the corpus is a static container; as it should remain unchanged, we will transform the corpus into a tokens object, which we will use to further clean and process the text for modelling.

We transform our data frame (called *reviews*) into a corpus with the following code:

# load the packages we'll need   
 library(plyr)

library(dplyr)   
 library(quanteda)   
   
 # transform the data frame into a corpus   
 reviews\_corpus <- corpus(reviews, docid\_field = "reviewid",   
 text\_field = "content")

*Step 2: Make a Tokens Object and Clean the Text*

Now that we have our text and meta-data stored in the corpus, we are ready to create a tokens object. We can think of tokens as an intermediate object, which consists of a list of character vectors containing the tokens of a given text. Each element of the list corresponds to an input document; there are as many list elements as there are texts in the corpus. We will use this tokens object to perform our text cleaning.

One of the great things about Quanteda is that it makes it very straightforward to do many of the cleaning operations that are necessary for any good NLP pipeline. The text cleaning and preparing functions are included in the package, and you can combine them in sequence to clean text in a very easy and straightforward manner.

In the current case, we would like to clean the text in a number of different ways. As we are primarily interested in linguistic markers of album quality, we can safely remove numbers, punctuation, symbols, and stopwords (e.g. words which occur frequently but have little or no meaning, such as “the”). We will also make all letters lowercase, and [stem](https://en.wikipedia.org/wiki/Stemming) the words (which removes the end of the word; e.g. *argue*, *argued*, *argues*, *arguing* are all truncated to *argu*). We then remove all words with less than 3 characters, as these are typically unimportant. Finally, we extract unigrams and bi-grams (e.g. 1 and 2-word combinations).

I created a function, called *make\_clean\_tokens*, which performs all of these steps. We then apply the function to our corpus object, which returns the cleaned tokens object called *clean\_toks*. This functional way of working with data makes the code easier to read and avoids creating many small objects that are used only a couple of times, and which consume memory and clutter our workspace.

The function to make the clean tokens object is shown below:

# make a tokens object and clean the text  
make\_clean\_tokens <- function(corpus\_f){  
 # make the base tokens object  
 # remove punctuation, numbers, symbos  
 # and stopwords (note: default stopword matching is case-insensitive)  
 clean\_toks\_f <- tokens\_remove(tokens(corpus\_f, what = 'word', remove\_punct = TRUE,   
 remove\_numbers = TRUE, remove\_symbols = TRUE),   
 stopwords("english"))  
 # convert all letters to lower case  
 clean\_toks\_f <- tokens\_tolower(clean\_toks\_f)  
 # then stem  
 clean\_toks\_f <- tokens\_wordstem(clean\_toks\_f, language = quanteda\_options("language\_stemmer"))  
 # remove words less than three characters  
 clean\_toks\_f <- tokens\_select(clean\_toks\_f, selection = "keep", min\_nchar = 3)  
 # select bigrams and unigrams  
 clean\_toks\_f <- tokens(clean\_toks\_f, ngrams = 1:2)  
 return(clean\_toks\_f)  
}  
  
# apply the function to our review corpus   
clean\_toks <- make\_clean\_tokens(reviews\_corpus)  
  
# what does the first element of our tokens object  
# look like?  
clean\_toks[1]

Which returns:

tokens from 1 document.  
22703 :  
 [1] "trip-hop" "eventu" "becam" "90s"   
 [5] "punchlin" "music-press" "shorthand" "overhyp"   
 [9] "hotel" "loung" "music" "today"   
 [13] "much-malign" "subgenr" "almost" "feel"   
 [17] "like" "secret" "preced" "listen" ...

This is the first element of our cleaned tokens object. Specifically, it is a character vector for review 22703 (the first review ID in the original dataset), and each word is a separate element in the vector. We can see that the words have been stemmed and stopwords have been removed. A nice aspect of Quanteda is that it preserves internal hyphens (though this functionality can be turned off), as we see with “trip-hop,” “much-malign” and “music-press.” Many less-intelligent text processing routines would remove the hyphen and create two separate words. There are a number of well-thought-out functions like this in Quanteda, making it much more efficient (and safer) to use than quickly-written regular expressions one often uses when cleaning data.

*Steps 3 and 4: DFM / Feature Selection & Producing the Modelling Data*

Next, we will take our tokens object and use it to create a *dfm*, or document-feature matrix. This representation considers documents as rows and “features” (single words, bi-grams, etc.) as columns. One of the nice things about the *dfm* is that it retains meta-data from the original input dataset. We will use the *dfm* and the meta-data to create the final matrix which we will use in modelling.

The code below contains a function that executes a series of operations on our tokens object. We first create a dfm, to which we apply tf-idf (term frequency, inverse document frequency) weighting. [Tf-idf](https://en.wikipedia.org/wiki/Tf%E2%80%93idf) is a weighting system that assigns a lower weight to words that occur in many documents, and a higher weight to words that occur frequently in fewer documents.

The number of features (text elements – both unigrams and bigrams) in the *dfm* is enormous – 3,954,138 to be precise. We cannot use all of them in modelling due to computational (memory limits) and statistical (many of the features occur very infrequently and therefore make poor predictors) considerations. In short, we need a way to reduce the number of features in our *dfm* for modelling. There are many approaches for selecting features (essentially reducing the width of the *dfm*) in text mining and NLP. Here, we will use the tf-idf score to select words to include in our analysis.

The function below calculates the average tf-idf score for every feature in our *dfm*. It then uses the quantile function to select the words with the largest average tf-idf scores. (I had to play around with the quantile value a bit. I selected the final value so that I would retain only around 2300 words out of the total 3.9 million.) Included in the function are several print statements, which show us some sample results of the calculations, and the size of the intermediate and final pieces of our modelling data.

# make a dataset for modelling: dfm + model matrix   
 make\_model\_data\_tfidf <- function(tokens\_f){   
 # make a dfm object from the tokens   
 myDfm\_f <- dfm(tokens\_f, verbose = TRUE)   
 # add TFIDF weight   
 myDfm\_f <- dfm\_tfidf(myDfm\_f)   
 # because the dfm object is a type of sparse matrix,   
 # we can use matrix commands on it.   
 # we first calculate the mean tfidf score for each word across the reviews   
 # with col\_sums from the slam package   
 word\_mean\_tfidf\_f <- slam::col\_sums(myDfm\_f, na.rm = T) / dim(myDfm\_f)[1]   
 print('first 5 tfidf scores:')   
 print(word\_mean\_tfidf\_f[1:5])   
 # then we extract the words with the highest tfidf scores   
 # based on quantile   
 words\_above\_threshold\_f <- word\_mean\_tfidf\_f[word\_mean\_tfidf\_f > quantile(word\_mean\_tfidf\_f,.9994)]   
 print('first 5 words above the threshold:')   
 print(words\_above\_threshold\_f[1:5])   
 # finally we subset dfm to text features above the threshold   
 myDfm\_f <- dfm\_select(myDfm\_f, names(words\_above\_threshold\_f), selection = "keep")   
 # and make boolean (0/1) weights for terms   
 myDfm\_f <- dfm\_weight(myDfm\_f, scheme = "boolean")   
 # turn the reduced dfm into a matrix   
 # to use the text features for modeling   
 text\_features\_f <- as.matrix(myDfm\_f)   
 print('size of text features:')   
 print(dim(text\_features\_f))   
 # we now extract the non-text features   
 # (genre indicators and our review score to predict)   
 # here we extract the names of the features in our   
 # original data that start with 'genre': these are   
 # our dummy variables, one per genre   
 genre\_features\_f <- grep('genre\_', names(myDfm\_f@docvars), value = TRUE)   
 # we add our output variable to the genre variable names   
 all\_non\_text\_feature\_names\_f<- c('score\_to\_predict', genre\_features\_f)   
 # and extract all of these features from the original data   
 # which is embedded in the dfm (as a type of meta-data)   
 non\_text\_features\_f <- myDfm\_f@docvars[all\_non\_text\_feature\_names\_f]   
 print('size of non-text features:')   
 print(dim(non\_text\_features\_f))   
 # bind genres/review score and text features together into a single matrix   
 modelling\_data\_f <- cbind(text\_features\_f, as.matrix(non\_text\_features\_f))   
 print('size of modelling data:')   
 print(dim(modelling\_data\_f))   
 # return this final matrix   
 return(modelling\_data\_f)   
 }   
   
 # make the modeling data   
 modelling\_data <- make\_model\_data\_tfidf(clean\_toks)

Selecting Text Features

The code calculates the average tf-idf score for each word across all the documents, and selects the words with the highest average tf-idf scores. We can then pass this vector of words directly to the *dfm\_select* function in Quanteda, which subsets the total *dfm* to just these words. Rather than using the tf-idf scores in modelling, I booleanize all of the values (e.g. code them in a 0/1 format to indicate whether the word was present in the document). This simplifies the interpretation of the final model results. We can interpret the Lasso model coefficients as the change in the album review score if the given word is used in the review. Finally, we convert the *dfm* to a simple matrix, which stores the booleanized values for our chosen features.

Adding Non-Text Features

We’ll also need some other data in order to make our model. First, we’ll need our outcome variable, the Pitchfork review score for each album (called *score\_to\_predict* in our data). Second, we’ll need to include the dummy variables for music review genre.

A very useful feature of the *dfm* is that it contains meta-data from our original data frame. We can therefore extract the additional data we need directly from the *dfm*!

In the code above, I extract the Pitchfork review score and the dummy columns from the original data, and return them together in a matrix (called *non\_text\_features\_f*). I then concatenate the text features and the features from our original data; this constitutes the final matrix that we’ll use in modelling. The function then returns this matrix.

As the function advances, it prints out the following output to the console:

[1] "first 5 tfidf scores:"  
 trip-hop eventu becam 90s punchlin   
0.03244423 0.10223274 0.09059614 0.10508835 0.03514300   
[1] "first 5 words above the threshold:"  
 eventu becam 90s hotel think   
0.10223274 0.09059614 0.10508835 0.04895410 0.20718557   
[1] "size of text features:"  
[1] 18389 2373  
[1] "size of non-text features:"  
[1] 18389 10  
[1] "size of modelling data:"  
[1] 18389 2383

This output helps us understand how the data are transformed throughout the process. The first line above shows the first 5 tf-idf scores, and confirms that we have indeed weighted the words in our corpus. After we calculate the average tf-idf scores and make a selection of words based on these scores, we print the first 5 words that are above the chosen threshold. We can see that some (but not all) of the first 5 words in our first document are above the threshold. After we make our selection of features and turn it into a matrix, we can see that we have extracted 2,373 different text features. We also get confirmation that we have extracted 10 non-text features; the Pitchfork album review score and the 9 dummy indicators for album genre. After both matrices are concatenated, our final model matrix contains 18,389 rows and 2,383 columns.

Our model matrix looks like this (only first 5 rows and first 7 columns shown):

|  | **eventu** | **becam** | **90s** | **hotel** | **think** | **music** | **today** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 22703 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 22721 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 22659 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 22661 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 22725 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

**Modelling**

We will use [Lasso regression](https://en.wikipedia.org/wiki/Lasso_(statistics)) to analyze the relationship between our features and our output variable (album review score). Lasso regression is a form of penalized regression that performs automatic feature selection, only retaining the most-predictive features in the final model.  
We will split the data into training and test sets. We make the model on the training set and evaluate its performance in the holdout test set. Let’s first set up an index variable that will allow us to sample 70% of our data for training:

# train-test split   
 # training data will be 70% of the sample size   
 smp\_size <- floor(0.70 \* nrow(modelling\_data))   
 # set the seed to make your partition reproducible   
 set.seed(123)   
 # calculate the indexes of the observations in   
 # the training sample   
 train\_idx <- sample(seq\_len(nrow(modelling\_data)), size = smp\_size)

And then compute the model:

# load the glmnet package   
 library(glmnet)   
 # function to make the Lasso model   
 make\_model <- function(modelling\_data\_f){   
 # make a subset of the training data   
 train\_f <- modelling\_data\_f[train\_idx, ]   
 # make a subset of the test data   
 test\_f <- modelling\_data\_f[-train\_idx, ]   
 # extract feature names   
 feature\_names\_f <- colnames(train\_f)[!colnames(train\_f) %in% c('reviewid', 'score\_to\_predict')]   
 # produce the model, using 10-fold cross-validation   
 mymodel\_f <- cv.glmnet(y = train\_f[,'score\_to\_predict'],   
 x = train\_f[,feature\_names\_f],   
 family = "gaussian", nfolds = 10, alpha = 1)   
   
 # predict on the test data   
 pred\_f <- predict(mymodel\_f, s="lambda.1se", newx = test\_f[,feature\_names\_f], type="response")   
 # and calculate model performance metrics   
 # error   
 error\_f <- test\_f[,'score\_to\_predict'] - pred\_f   
 # root mean squared error   
 rmse\_f <- sqrt(mean(error\_f^2))   
 print('RMSE:')   
 print(rmse\_f)   
 # mean absolute error   
 mae\_f <- mean(abs(error\_f))   
 print('MAE:')   
 print(mae\_f)   
 # return the model object   
 return(mymodel\_f)   
 }   
   
 # pass the data to our modelling function   
 # and return the model object   
 lasso\_model <- make\_model(modelling\_data)

During the execution of the function, the following output is printed to the console:

[1] "RMSE:"  
[1] 1.070765  
[1] "MAE:"  
[1] 0.7661584

We will discuss the meaning of these two model performance statistics below.

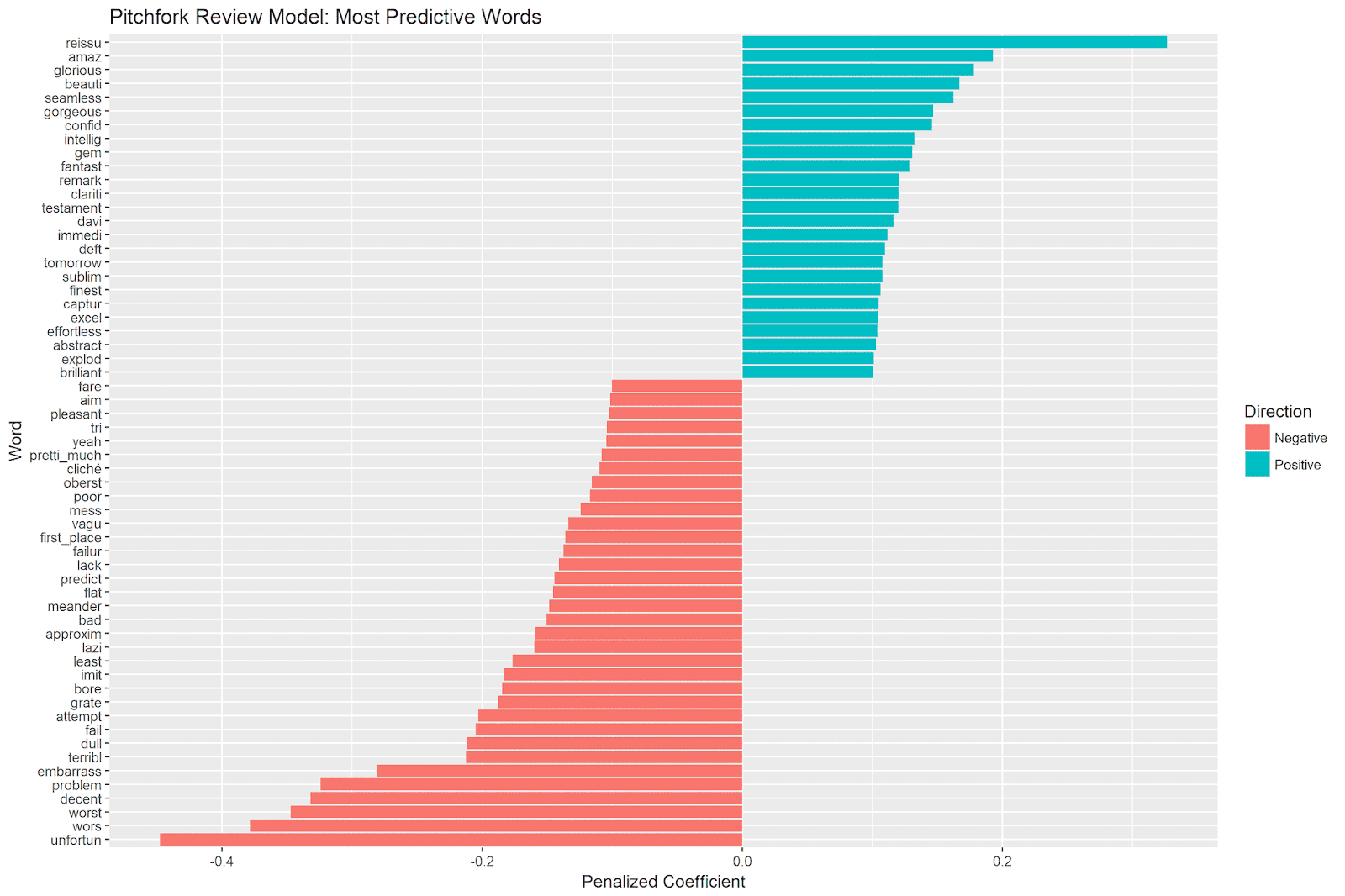
**Understanding and Evaluating the Model**

*Understanding the Model*

What features did the Lasso model identify as being most predictive of the album review scores? In order to answer this question, we can extract the most important coefficients from the model object and plot them using **ggplot2**. The following function accomplishes this:

# plot the top features   
 # load ggplot2 package for plotting   
 library(ggplot2)   
 plot\_top\_features <- function(model\_f){   
 # extract the coefficients from the model   
 coefficients\_f <- predict(model\_f, s="lambda.1se", type = "coefficients")[, 1]   
 # make it a data frame   
 coef\_df\_f <- as.data.frame(coefficients\_f)   
 # extract the feature names from the row names   
 coef\_df\_f$feature <- row.names(coef\_df\_f)   
 # and reset the row names   
 row.names(coef\_df\_f) <- NULL   
 # name the columns in our coefficient data frame   
 names(coef\_df\_f) <- c('coefficient', 'feature')   
 # dplyr chain: remove coefficient for intercept and   
 # remove coefficients with an absolute value of lower than .1   
 coef\_df\_f %>% filter(feature != '(Intercept)') %>% filter(abs(coefficient) > .1) %>%   
 mutate(Direction = ifelse(coefficient >0, 'Positive', 'Negative')) %>%   
 mutate(word = reorder(feature, coefficient)) %>%   
 ggplot(aes(x = word, y = coefficient, fill = Direction)) +   
 geom\_col() +   
 coord\_flip() +   
 labs(x = 'Word', y = 'Penalized Coefficient',   
 title = 'Pitchfork Review Model: Most Predictive Words')   
 }   
   
 # make the plot   
 plot\_top\_features(lasso\_model)

Which gives us the following plot:

[](https://i2.wp.com/1.bp.blogspot.com/-o-IpnxzwS0w/W542K9dxBTI/AAAAAAAAAcI/hbc4-wh0gN8k3dnHlL_aIS7h4kVhq_CtwCLcBGAs/s1600/top_features.png?ssl=1)

Top Positive Features

The top positive feature is *reissue*. When this word is present in a review text, the model estimates that the review score will be .33 points higher. This seems logical – only noteworthy albums are likely to be given a reissue, and this underlying quality that leads to a higher review score.

Many of the top predictive features are simply synonyms for “good.” Examples include *amazing*, *glorious*, *beautiful*, *fantastic*, *remarkable*, etc. This makes sense and gives us confidence in the logic of the model. But it doesn’t really help us understand what specific artistic or musical qualities make for a great album.

There are some clues among some of the features, however. *Seamless* indicates the importance of an artistic whole; the songs of more highly-reviewed albums fit together as a whole and transition from one to the other in an easy way. In other words, the totality of the album package seems important to Pitchfork reviewers.

Other features that indicate positive musical or artistic qualities of good albums include *confident,* *intelligent, clarity, capture*, and *effortless*. *Confidence* is no doubt critical on both an artistic and a musical level (e.g. having a conceptual and musical vision and executing it in a direct and competent manner). Intelligence speaks to the translation of a conceptual idea to a musical execution. An *intelligent* album is successful in transforming a larger idea or concept into an album-length execution, including song content and layout, production, and the synthesis that is the artistic statement encapsulated in the album whole. A successful album *captures* the *clarity* of the ideas in an aesthetically pleasant, *effortless* way.   
    
Top Negative Features

As we saw with the positive features, most of the negative features are simply synonyms for “bad.” Examples include *unfortunate*, *worst*, *terrible*, etc.

However, some of the features give us a sense of the artistic and musical qualities that signal a poor album. For example, *attempt, try* and *aim* suggest a *failed* execution of an artistic or musical idea. The musicians were trying to realize a specific vision, but did not succeed. The end result is *dull, boring,* or *grating*.

Bad albums are not focused. They are *meandering*, *vague* in their intent, and in worst cases a *mess*. The lack of originality in an album’s content is also a signal of a poor review. Bad albums are *cliché* (unoriginal), *predictable*, and rather than realize a unique vision, they *imitate* other work (*poorly*).

Interestingly, none of the genre features (e.g. rock, rap, jazz, etc.) come up in the most predictive features. This suggests that, at least when accounting for the text features, the different genres are not systematically rated more positively or negatively from one another.

*Evaluating the Model*

The above function prints out some figures that give us a first indication of model quality. Our root mean squared error is 1.07 and our mean absolute error is .77. In other words, we are off on average by .77 points. On a scale from 1-10, this seems to be not too bad.  
  
Let’s plot the actual review scores versus the model predictions for the test set. With a perfect model, these values would match completely.

We first make a function to compute the model predictions on the test set, and return a data frame with the actual and predicted scores:

# this function computes the predictions   
 # and returns a data frame with these values   
 # and the original album review scores   
 return\_model\_predictions <- function(model\_f,data\_f){   
 # first model   
 test\_data\_f <- data\_f[-train\_idx, ]   
 feature\_names\_f <- colnames(test\_data\_f)[!colnames(test\_data\_f) %in% c('reviewid', 'score\_to\_predict')]   
 pred\_f <- predict(model\_f, s="lambda.1se", newx = test\_data\_f[,feature\_names\_f], type="response")   
 # bind together actual and predicted scores   
 actual\_predictions\_f <- cbind(test\_data\_f[,'score\_to\_predict'], pred\_f)   
 # fix the column names   
 dimnames(actual\_predictions\_f)[2][[1]] <- c('score\_to\_predict','model\_prediction')   
 # return data frame with the true rating and the model prediction   
 return(as.data.frame(actual\_predictions\_f))   
 }   
   
 # compute the predictions and store in a data frame   
 # called prediction\_df   
 prediction\_df <- return\_model\_predictions(lasso\_model, modelling\_data)

Our prediction data frame is called *prediction\_df*, and looks like this (only first 5 rows shown):

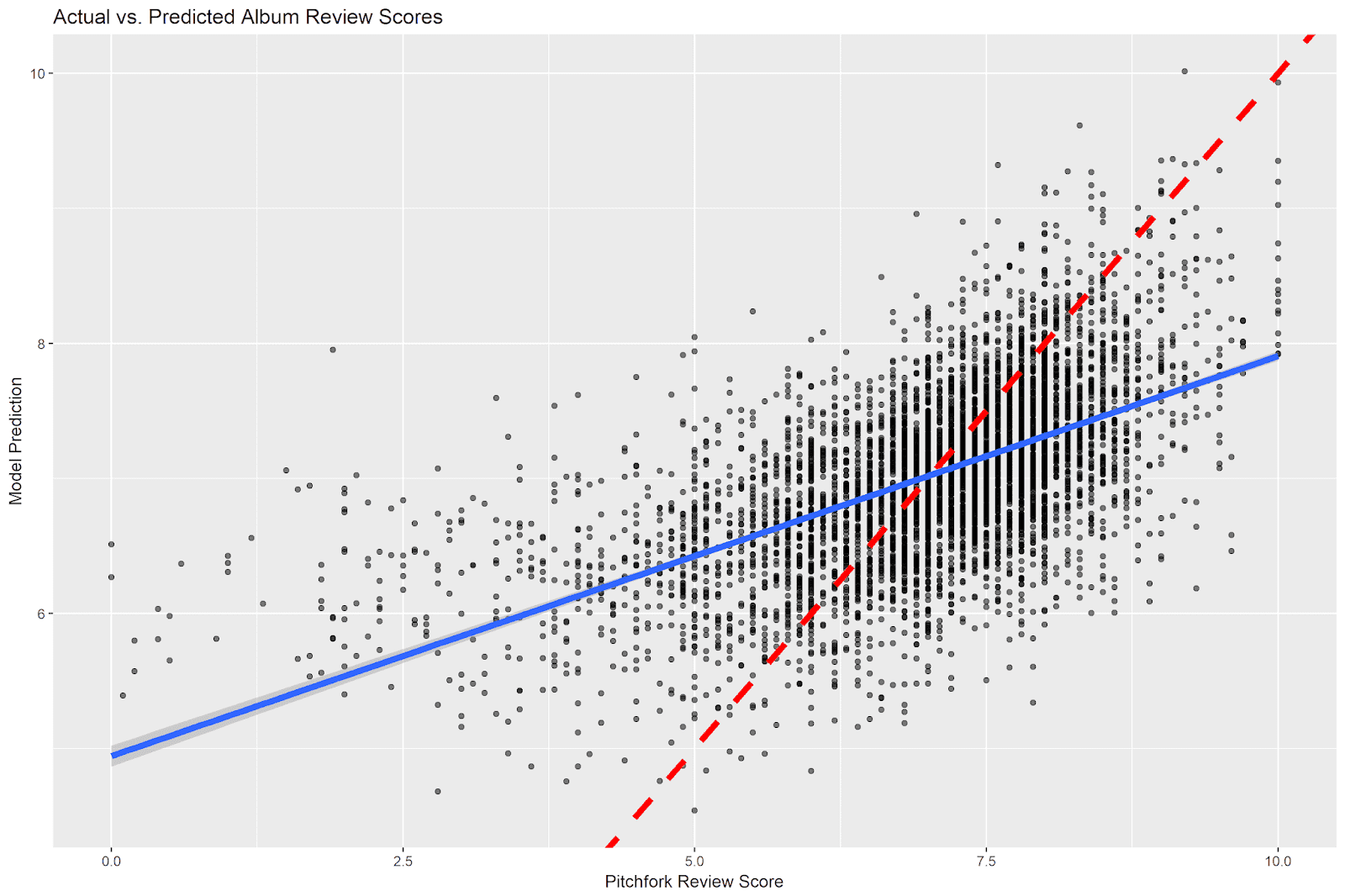
|  | **score\_to\_predict** | **model\_prediction** |
| --- | --- | --- |
| 22721 | 7.9 | 6.99 |
| 22659 | 7.3 | 7.16 |
| 22725 | 8.1 | 7.23 |
| 22720 | 3.5 | 6.31 |
| 22699 | 7.4 | 7.49 |
| 22665 | 6.6 | 7.41 |

The row names are the document id in our original data, *score\_to\_predict* is the actual Pitchfork review score, and *model\_prediction* is the expected value from the model for each review.

We can plot the actual review scores versus the predictions via ggplot2 with the following code:

# plot the actual vs. predicted review scores   
 # add a red line that would indicate perfect predictions   
 # add a blue regression line for actual vs. predicted scores   
 ggplot(prediction\_df, aes(x= score\_to\_predict, y = model\_prediction)) +   
 geom\_point(alpha = .5) +   
 geom\_abline(intercept = 0, slope = 1, color = 'red',   
 linetype = 2, size = 2, show.legend = TRUE) +   
 geom\_smooth(method = lm, se = TRUE, size = 2) +   
 labs(x = 'Pitchfork Review Score', y = 'Model Prediction',   
 title = 'Actual vs. Predicted Album Review Scores')

Which yields the following plot:

[](https://i1.wp.com/4.bp.blogspot.com/-FxlTxvUnMaw/W55Ip9LTtBI/AAAAAAAAAcU/yVsOocwvcIUM-S7PZZSCdZdsAUpqPcbwwCLcBGAs/s1600/actual_predicted.png?ssl=1)

The red dashed line is the [identity line](https://en.wikipedia.org/wiki/Identity_line); it indicates equality between the actual and predicted review scores (e.g. the same values on the x and y axes). If every prediction were perfect, all of the data points would lie on this line. The blue line indicates the regression line between the actual and predicted scores. This shows the predicted relationship between the two scores across their ranges.

Our model is clearly more accurate in some regions of the data than in others. At the extremes of the Pitchfork review scores (e.g. albums with very high or very low scores), our model does not perform well. There are relatively few very low and very high Pitchfork album reviews, and our model has a hard time understanding when a review will have a score at these extremes. This problem is worse at the low end of the Pitchfork review spectrum. While actual Pitchfork reviews can have a score of zero, our model never predicts a score lower than 4.5.

The model does better in the middle-to-high range (e.g. from around 5 to around 8.5). The bulk of our data is contained within this region, making it easier for the model to pick up on signals of quality within this range. We would expect that our model is most accurate at the point where the two lines cross: e.g. at around Pitchfork review scores of 7. This is the place where our average model prediction is closest to the true Pitchfork album review score.

**Summary and Conclusion**

In this post we did text mining and natural language processing on Pitchfork album reviews and built a model to predict linguistic signals of album quality. We used the Quanteda package to clean our text data and to extract text and non-text features to predict the Pitchfork album review score. The Quanteda package made it straightforward to execute basic (e.g. removing punctuation, stemming words) and advanced (e.g. feature selection via tf-idf weighting) text processing steps. We then built a Lasso regression model which used the text and non-text features to predict the album review scores.

The model results gave some clues as to what makes a good vs. bad album, according to Pitchfork reviewers. Good albums are seamless artistic packages. They are confident, intelligent, and capture the clarity of a larger idea with music in a succinct, effortless way. Bad albums, in contrast, attempt to impose a conceptual and musical vision, but fail in the execution. They are unfocused, unoriginal, and predictable. As a result, bad albums are boring or grating.

On average, the model performance was acceptable, with a mean average error of .77 points (on a scale from 0 to 10). However, the model did not perform equally well in all ranges of the Pitchfork review score data. As there were relatively few albums with very low scores, the model was unable to find distinctive features that signaled signaled very poor quality reviews.