Predictive Lyric Analytics with NLP

Harini

02/01/2021

INTRODUCTION

Lyric analysis is slowly finding its way into data science communities as the possibility of predicting “Hit Songs” approaches reality. There are variety of machine learning (ML) classification algorithms to build models step-by-step that predict the genre of a song.The project work focus on variety of analytic tasks on a case study of musical lyrics by the legendary artist Prince, as well as other artists and authors. The project build models to classify songs into their associated genre and to investigate the possibility of using lyrics to determine commercial success.

DATA INGESTION

```
  The train and test dataset has five authors/artist and their books/songs. It has michael jackson,pastyline
,prince,amy grant,,,,chris tomuin,eminem,jay Z,jonny cash,ml dummies,mlearning r. The project explains and p
rovides a musical use case for a form of supervised learning, specifically classification, that is based on
the lyrics of a variety of artists (and a couple of book authors)
```

Then use unnest() from tidytext to create the tidy version with one word per record. ## DATA PREPROCESSING

Since the dataset has songs and book pages, I’ll refer to them each as a document. The features that are created based on documents and their associated metadata.There are artists and authors, I will refer to them as the source of each document.

```
five_sources_data %>%
  group_by(genre, source) %>%
  summarise(doc_count = n()) %>%
  my_kable_styling("Training Dataset")
```

```
## `summarise()` regrouping output by 'genre' (override with `.groups` argument)
```

Training Dataset

genre	source	doc_count
christian	amy-grant	279
country	johnny-cash	279
data-science	machine_learning	279
hip-hop-rap	eminem	279
pop-rock	prince	279

```
five_sources_data_test %>%
  group_by(genre, source) %>%
  summarise(doc_count = n()) %>%
  my_kable_styling("Test Dataset")
```

```
## `summarise()` regrouping output by 'genre' (override with `.groups` argument)
```

Test Dataset

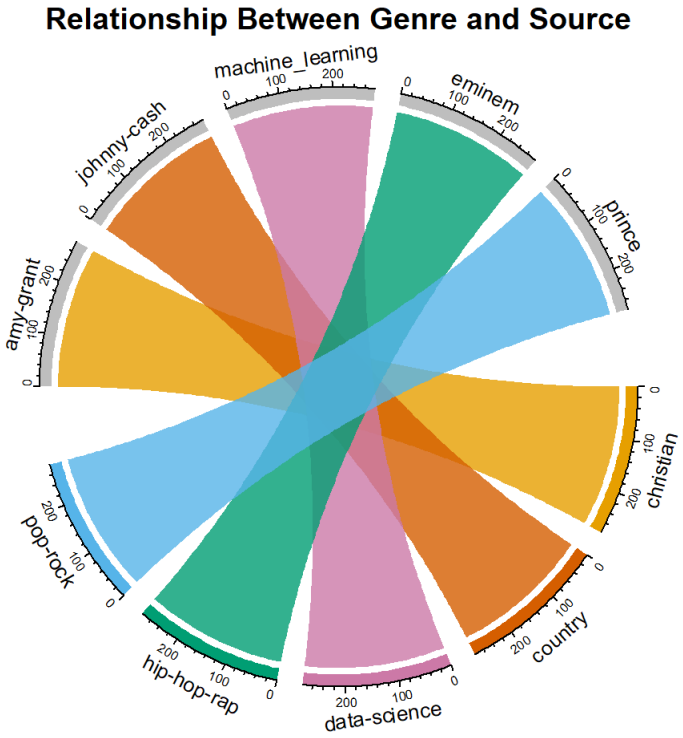
genre	source	doc_count
christian	chris-tomlin	165
country	patsy-cline	165
data-science	machine_learning_r	165
hip-hop-rap	jay-z	165
pop-rock	michael-jackson	165

To see the genre, source, and the number of documents, the following chord diagram is a better way to view these relationships. There is a one-to-one relationship between source and genre because the artists are classified ; however, cross-over artists are very common.

```
#get SONG count per genre/source. Order determines top or bottom.
genre_chart <-  five_sources_data %>%
  count(genre, source)

circos.clear() #very important! Reset the circular layout parameters
#assign chord colors
grid.col = c("christian" = my_colors[1], "pop-rock" = my_colors[2],
             "hip-hop-rap" = my_colors[3], "data-science" = my_colors[4],
             "country" = my_colors[5],
             "amy-grant" = "grey", "eminem" = "grey",
             "johnny-cash" = "grey", "machine_learning" = "grey",
             "prince" = "grey")
# set the global parameters for the circular layout. Specifically the gap size
circos.par(gap.after = c(rep(5, length(unique(genre_chart[[1]])) - 1), 15,
                         rep(5, length(unique(genre_chart[[2]])) - 1), 15))

chordDiagram(genre_chart, grid.col = grid.col, transparency = .2)
title("Relationship Between Genre and Source")
```



This diagram would look like if there was not a one-to-one relationship.If prediction is made on genre based only on lyrics, would have gone wrong.

Predict Genre

Think about the song and what makes one different from another, lyrically. A common theme in all music is repetition. Do some genres use repetition more than others? What about word length? Do some use larger or smaller words more often? What other factors drive to describe lyrics? Note that all of the variables listed above are quantitative features, (counts, lengths, etc.) based on words per song. But what about individual words specific to genres?

Feature Engineering

Get the most common (frequently used) words per genre. Start by getting the total number of words per genre. Then group by words per genre and get a count of how often each word is used. Now select the top n most frequent words defined by the number_of_words variable. I came up with a total of 5500 as the most optimal number of words. Play around with this value and see how it impacts outcome. Many words are very common to more than one genre (such as time, life, etc.) and I have removed these words with the multi_genre variable below. This makes for a cleaner list of distinct words that create a better distinction between the sources.

```
#play with this number until getting the best results for the model.
number_of_words = 5500

top_words_per_genre <- five_sources_tidy %>%
  group_by(genre) %>%
  mutate(genre_word_count = n()) %>%
  group_by(genre, word) %>%
  #note that the percentage is also collected, when really you
  #could have just used the count, but it's good practice to use a %
  mutate(word_count = n(),
         word_pct = word_count / genre_word_count * 100) %>%
  select(word, genre, genre_word_count, word_count, word_pct) %>%
  distinct() %>%
  ungroup() %>%
  arrange(desc(word_pct)) %>%
  top_n(number_of_words) %>%
  select(genre, word, word_pct)
```

```
## Selecting by word_pct
```

```
#remove words that are in more than one genre
top_words <- top_words_per_genre %>%
  ungroup() %>%
  group_by(word) %>%
  mutate(multi_genre = n()) %>%
  filter(multi_genre < 2) %>%
  select(genre, top_word = word)

#create lists of the top words per genre
book_words <- lapply(top_words[top_words$genre == "data-science",], as.character)
country_words <- lapply(top_words[top_words$genre == "country",], as.character)
hip_hop_words <- lapply(top_words[top_words$genre == "hip-hop-rap",], as.character)
pop_rock_words <- lapply(top_words[top_words$genre == "pop-rock",], as.character)
christian_words <- lapply(top_words[top_words$genre == "christian",], as.character)
```

Think about each feature below and how it could vary according to the genre.country_word_count is merely a count of the top country words that appear in each song. Notice that I have assigned more weight to explicit and book words (see the 10 and 20 used in the sum() function). I did this because they are very distinctive and help to classify documents. Again, this was a trial and error process!

```
features_func_genre <- function(data) {
  features <- data %>%
  group_by(document) %>%
  mutate(word_frequency = n(),
         lexical_diversity = n_distinct(word),
         lexical_density = lexical_diversity/word_frequency,
         repetition = word_frequency/lexical_diversity,
         document_avg_word_length = mean(nchar(word)),
         title_word_count = lengths(gregexpr("[A-z]\\W+",
                                             document)) + 1L,
         title_length = nchar(document),
         large_word_count =
           sum(ifelse((nchar(word) > 7), 1, 0)),
         small_word_count =
           sum(ifelse((nchar(word) < 3), 1, 0)),
         #assign more weight to these words using "10" below
         explicit_word_count =
          sum(ifelse(word %in% explicit_words$explicit_word,10,0)),
         #assign more weight to these words using "20" below
         book_word_count =
           sum(ifelse(word %in% book_words$top_word,20,0)),
         christian_word_count =
           sum(ifelse(word %in% christian_words$top_word,1,0)),
         country_word_count =
           sum(ifelse(word %in% country_words$top_word,1,0)),
         hip_hop_word_count =
           sum(ifelse(word %in% hip_hop_words$top_word,1,0)),
         pop_rock_word_count =
           sum(ifelse(word %in% pop_rock_words$top_word,1,0))
         ) %>%
  select(-word) %>%
  distinct() %>% #to obtain one record per document
  ungroup()

features$genre <- as.factor(features$genre)
return(features)
}
```

Now calling features() function for training and test datasets.

```
train <- features_func_genre(five_sources_tidy)
test  <- features_func_genre(five_sources_test_tidy)
```

Machine Learning Process

mlr (Machine Learning for R), is a framework that includes all of the frequently used machine learning algorithms. The ML process is simple: create a task, make a learner, train it, test it. Here are the steps :

Create the classifier task: declare the datasets and outcome (target) variable Normalize the data: pre-processing (scale and center) Create a list of learners: choose the learning algorithm(s) Choose a resampling method: choose a method to assess performance using a validation set during training Select the measures: create a list of measures such as accuracy or error rate Perform training/benchmarking: compare the results of the models based on the tasks and learners Tune the best model: choose the best performing learner and tune the hyperparameters Test on new data: run your model against data it has never seen before.

The Classifier Task

A task is merely the dataset on which a learner learns. Since this is a classification problem, create a classification task by using makeClassifTask(). Specifying classification outcome variable, genre, by passing it as the target argument. There is also a dataset that uses only the basic quantitative features consisting of document summaries and counts. This task, task_train_subset is created without the genre word count features (i.e., country_word_count, pop_rock_word_count, etc.) to illustrate the importance of these contextual predictors in the final model. So using the dataset train[3:13] removes these variables.

```
#create classification tasks to use for modeling

#this dataset does not include genre specific words
task_train_subset <- makeClassifTask(id = "Five Sources Feature Subset",
                                     data = train[3:13], target = "genre")
```

```
## Warning in makeTask(type = type, data = data, weights = weights, blocking =
## blocking, : Provided data is not a pure data.frame but from class tbl_df, hence
## it will be converted.
```

```
#create the training dataset task
task_train <- makeClassifTask(id = "Five Sources",
                              data = train[-c(1:2)], target = "genre")
```

```
## Warning in makeTask(type = type, data = data, weights = weights, blocking =
## blocking, : Provided data is not a pure data.frame but from class tbl_df, hence
## it will be converted.
```

```
#create the testing dataset task
task_test <- makeClassifTask(id = "New Data Test",
                             data = test[-c(1:2)], target = "genre")
```

```
## Warning in makeTask(type = type, data = data, weights = weights, blocking =
## blocking, : Provided data is not a pure data.frame but from class tbl_df, hence
## it will be converted.
```

Normalize Data

It is simply a method of scaling data such that all the values are normalized between values of zero and one (or whatever values you pass). If some variables are much larger in value and are on a different scale than the others, they will throw off model by giving more weight to those variables. Normalization takes care of this problem. mlr provides a simple function called normalizeFeatures() for this step.

```
#scale and center the training and test datasets
task_train_subset <- normalizeFeatures(task_train_subset, method = "standardize",
  cols = NULL, range = c(0, 1), on.constant = "quiet")

task_train <- normalizeFeatures(task_train, method = "standardize",
  cols = NULL, range = c(0, 1), on.constant = "quiet")

task_test <- normalizeFeatures(task_test, method = "standardize",
  cols = NULL, range = c(0, 1), on.constant = "quiet")
```

Create a List of Learners

A learner in mlr is generated by calling makeLearner().We can obtain a list of possible classification algorithms by calling listLearners("classif")[c("class","package")].This shows the algorithms you have to choose from as well as their dependent packages (which may need to install separately).Currently there are over 80 classification learners in mlr.

```
#create a list of learners using algorithms you'd like to try out
lrns = list(
makeLearner("classif.randomForest", id = "Random Forest", predict.type = "prob"),
makeLearner("classif.rpart", id = "RPART", predict.type = "prob"),
makeLearner("classif.xgboost", id = "xgBoost", predict.type = "prob"),
makeLearner("classif.kknn", id = "KNN"),
makeLearner("classif.lda", id = "LDA"),
makeLearner("classif.ksvm", id = "SVM"),
makeLearner("classif.naiveBayes", id = "Naive Bayes"),
makeLearner("classif.nnet", id = "Neural Net", predict.type = "prob")
)
```

Resampling

It involves repeatedly drawing samples from a training set and refitting a model on each sample. This may allow you to obtain information not available from fitting the model only once with the original training data.k-fold cross validation indicated by the "CV" in the call to makeResampleDesc() which returns a resample description object (rdesc). This approach randomly divides the dataset into k groups (folds) of equal size. The first fold is used as a validation set, and the remaining folds are used for training. This is repeated k times on each of the folds. The error rate is captured for each iteration and averaged at the end.

```
# n-fold cross-validation use stratify for categorical outcome variables
rdesc = makeResampleDesc("CV", iters = 10, stratify = TRUE)
```

Performance Measures

The typical objective of classification is to obtain a high prediction accuracy and minimize the number of errors. Accuracy is the number of correct predictions from all predictions made. Confusion matrix to see how things were classified (predicted vs. actual).

```
#let the benchmark function know which measures to obtain
#accuracy, time to train
meas = list(acc, timetrain)
```

Train Models / Benchmark

Benchmark experiment in which different algorithms (learning methods) are applied to dataset. It will compare algorithms according to the specific measures of interest (i.e., accuracy).benchmark() to train your model and generate a BenchmarkResult object from which we can access the models and results. Take the dataset which does not include the genre-specific word counts. Pass to benchmark list of learners, the subset task, the resampling strategy (rdesc), and the list of measures we would like to see.

```
## # weights:  53
## initial  value 2121.238474
## iter  10 value 1247.681184
## iter  20 value 1061.465919
## iter  30 value 959.217868
## iter  40 value 915.929911
## iter  50 value 893.309532
## iter  60 value 877.316434
## iter  70 value 867.879333
## iter  80 value 865.164422
## iter  90 value 863.329171
## iter 100 value 862.854645
## final  value 862.854645
## stopped after 100 iterations
## # weights:  53
## initial  value 2297.242401
## iter  10 value 1289.940050
## iter  20 value 1054.297126
## iter  30 value 948.902952
## iter  40 value 904.326376
## iter  50 value 875.799703
## iter  60 value 868.780991
## iter  70 value 861.445448
## iter  80 value 860.281054
## iter  90 value 858.369899
## iter 100 value 858.173163
## final  value 858.173163
## stopped after 100 iterations
## # weights:  53
## initial  value 2037.511804
## iter  10 value 1167.751558
## iter  20 value 984.414029
## iter  30 value 933.942871
## iter  40 value 896.553563
## iter  50 value 881.276543
## iter  60 value 871.204847
## iter  70 value 863.902499
## iter  80 value 861.449526
## iter  90 value 861.190190
## iter 100 value 860.376513
## final  value 860.376513
## stopped after 100 iterations
## # weights:  53
## initial  value 1953.156975
## iter  10 value 1122.429875
## iter  20 value 957.615394
## iter  30 value 921.577457
## iter  40 value 911.130084
## iter  50 value 906.010690
## iter  60 value 894.925266
## iter  70 value 886.138151
## iter  80 value 878.917182
## iter  90 value 874.540033
## iter 100 value 874.246630
```

```
## final  value 874.246630
## stopped after 100 iterations
## # weights:  53
## initial  value 2318.640226
## iter  10 value 1300.217822
## iter  20 value 1109.254914
## iter  30 value 1042.547409
## iter  40 value 961.039680
## iter  50 value 904.801734
## iter  60 value 878.992666
## iter  70 value 869.283761
## iter  80 value 863.580611
## iter  90 value 861.956799
## iter 100 value 861.384568
## final  value 861.384568
## stopped after 100 iterations
## # weights:  53
## initial  value 2146.923708
## iter  10 value 1048.294464
## iter  20 value 933.923794
## iter  30 value 891.739725
## iter  40 value 867.221163
## iter  50 value 858.932502
## iter  60 value 854.256915
## iter  70 value 853.297942
## iter  80 value 852.543395
## iter  90 value 852.160875
## iter 100 value 851.894444
## final  value 851.894444
## stopped after 100 iterations
## # weights:  53
## initial  value 2134.223222
## iter  10 value 1104.499428
## iter  20 value 998.857773
## iter  30 value 946.869656
## iter  40 value 918.309524
## iter  50 value 908.911254
## iter  60 value 903.653904
## iter  70 value 901.846729
## iter  80 value 901.343290
## iter  90 value 898.991190
## iter 100 value 895.585845
## final  value 895.585845
## stopped after 100 iterations
## # weights:  53
## initial  value 2207.561004
## iter  10 value 1110.321000
## iter  20 value 937.948608
## iter  30 value 894.719608
## iter  40 value 880.388885
## iter  50 value 872.424088
## iter  60 value 866.335419
## iter  70 value 861.991576
## iter  80 value 859.983547
## iter  90 value 857.742823
## iter 100 value 856.475265
## final  value 856.475265
## stopped after 100 iterations
## # weights:  53
## initial  value 2071.038811
## iter  10 value 1143.906534
## iter  20 value 954.479958
## iter  30 value 906.435669
## iter  40 value 879.959383
## iter  50 value 870.832602
## iter  60 value 868.787082
## iter  70 value 867.588093
## iter  80 value 866.775112
## iter  90 value 866.122603
## iter 100 value 864.821881
## final  value 864.821881
## stopped after 100 iterations
## # weights:  53
```

```
## initial  value 2080.616069
## iter  10 value 1214.700649
## iter  20 value 1059.274138
## iter  30 value 986.783566
## iter  40 value 944.988471
## iter  50 value 921.131229
## iter  60 value 909.319893
## iter  70 value 906.529881
## iter  80 value 904.831576
## iter  90 value 904.219262
## iter 100 value 904.066692
## final  value 904.066692
## stopped after 100 iterations
```

```
## [1] "BenchmarkResult"
```

This is the average accuracy across all samples for the validation datasets used in cross-validation when it is trained the model.This is not the same as the test dataset.There is a series of validation datasets that are held out during cross-validation, and a test dataset that you'll use after you decide on an algorithm and tune the model.Random Forest performs the best at r rf_perf percent (and took the longest to train). Keeping that value in mind, rerun this experiment on the full feature set. I recommend using the getBMR* getter functions.plotBMRSummary() getter function and do this by creating a list of tasks to pass to benchmark().

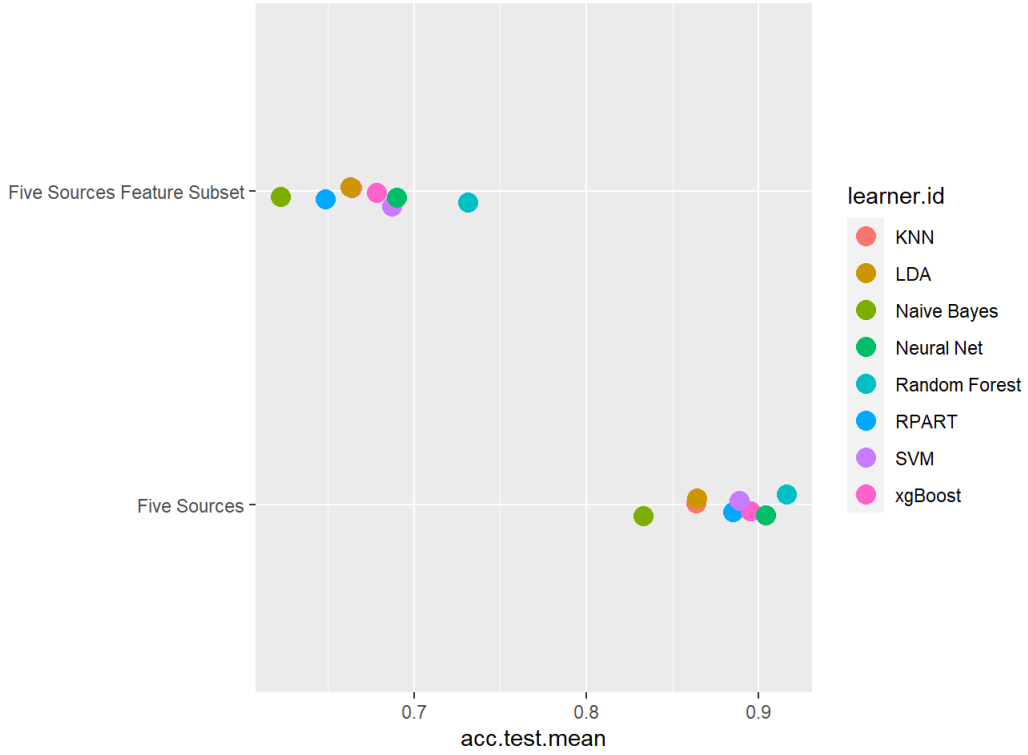
```
## # weights:  68
## initial  value 2137.856738
## iter  10 value 572.739576
## iter  20 value 368.106604
## iter  30 value 304.576692
## iter  40 value 283.561414
## iter  50 value 273.976512
## iter  60 value 267.384591
## iter  70 value 262.467582
## iter  80 value 257.608911
## iter  90 value 255.975384
## iter 100 value 255.213292
## final  value 255.213292
## stopped after 100 iterations
## # weights:  68
## initial  value 2072.141801
## iter  10 value 560.950452
## iter  20 value 382.030453
## iter  30 value 297.841396
## iter  40 value 280.799833
## iter  50 value 271.788096
## iter  60 value 267.704884
## iter  70 value 263.509588
## iter  80 value 260.707425
## iter  90 value 259.336474
## iter 100 value 258.766254
## final  value 258.766254
## stopped after 100 iterations
## # weights:  68
## initial  value 2105.125684
## iter  10 value 741.956370
## iter  20 value 431.377459
## iter  30 value 324.825147
## iter  40 value 284.640461
## iter  50 value 270.607195
## iter  60 value 266.528347
## iter  70 value 264.098539
## iter  80 value 261.904806
## iter  90 value 260.340707
## iter 100 value 259.323200
## final  value 259.323200
## stopped after 100 iterations
## # weights:  68
## initial  value 2121.727153
## iter  10 value 668.914465
## iter  20 value 434.558197
## iter  30 value 373.653806
## iter  40 value 325.010725
## iter  50 value 301.152026
```

```
## iter  60 value 288.847844
## iter  70 value 284.057500
## iter  80 value 282.531401
## iter  90 value 281.457509
## iter 100 value 279.929729
## final  value 279.929729
## stopped after 100 iterations
## # weights:  68
## initial  value 2141.663520
## iter  10 value 610.065939
## iter  20 value 424.084955
## iter  30 value 347.952667
## iter  40 value 312.341734
## iter  50 value 293.830669
## iter  60 value 287.424682
## iter  70 value 281.039598
## iter  80 value 279.266424
## iter  90 value 277.113514
## iter 100 value 273.913553
## final  value 273.913553
## stopped after 100 iterations
## # weights:  68
## initial  value 2156.815254
## iter  10 value 710.188067
## iter  20 value 462.990817
## iter  30 value 329.748532
## iter  40 value 290.227585
## iter  50 value 279.283731
## iter  60 value 270.713968
## iter  70 value 263.883594
## iter  80 value 259.833151
## iter  90 value 258.055533
## iter 100 value 256.366341
## final  value 256.366341
## stopped after 100 iterations
## # weights:  68
## initial  value 2110.633981
## iter  10 value 724.228829
## iter  20 value 447.770700
## iter  30 value 355.406986
## iter  40 value 307.936087
## iter  50 value 271.349416
## iter  60 value 258.070274
## iter  70 value 253.682612
## iter  80 value 251.600088
## iter  90 value 249.632765
## iter 100 value 249.251640
## final  value 249.251640
## stopped after 100 iterations
## # weights:  68
## initial  value 2048.423985
## iter  10 value 664.640562
## iter  20 value 412.831303
## iter  30 value 313.618729
## iter  40 value 292.839197
## iter  50 value 279.951774
## iter  60 value 268.806732
## iter  70 value 265.854747
## iter  80 value 264.133553
## iter  90 value 262.934624
## iter 100 value 261.847321
## final  value 261.847321
## stopped after 100 iterations
## # weights:  68
## initial  value 2150.200563
## iter  10 value 735.784404
## iter  20 value 382.905696
## iter  30 value 317.501778
## iter  40 value 293.075770
## iter  50 value 269.753501
## iter  60 value 266.156461
## iter  70 value 265.058295
## iter  80 value 264.417313
```

```
## iter  90 value 263.829146
## iter 100 value 263.567634
## final  value 263.567634
## stopped after 100 iterations
## # weights:  68
## initial  value 2188.713120
## iter  10 value 732.127957
## iter  20 value 479.098539
## iter  30 value 368.451237
## iter  40 value 307.590932
## iter  50 value 288.770126
## iter  60 value 282.324740
## iter  70 value 275.183128
## iter  80 value 272.401700
## iter  90 value 269.293289
## iter 100 value 266.092908
## final  value 266.092908
## stopped after 100 iterations
## # weights:  53
## initial  value 2142.741731
## iter  10 value 1228.312847
## iter  20 value 1033.235300
## iter  30 value 946.692712
## iter  40 value 896.957439
## iter  50 value 877.223290
## iter  60 value 868.168343
## iter  70 value 864.815965
## iter  80 value 863.381555
## iter  90 value 862.791632
## iter 100 value 862.558957
## final  value 862.558957
## stopped after 100 iterations
## # weights:  53
## initial  value 2175.393150
## iter  10 value 1212.883753
## iter  20 value 1021.037741
## iter  30 value 954.554604
## iter  40 value 899.605232
## iter  50 value 876.140333
## iter  60 value 871.261496
## iter  70 value 869.655045
## iter  80 value 868.920141
## iter  90 value 868.261603
## iter 100 value 867.553669
## final  value 867.553669
## stopped after 100 iterations
## # weights:  53
## initial  value 2210.660890
## iter  10 value 1137.014506
## iter  20 value 1024.443251
## iter  30 value 938.460339
## iter  40 value 874.457383
## iter  50 value 863.630887
## iter  60 value 858.641716
## iter  70 value 853.113750
## iter  80 value 849.835914
## iter  90 value 847.979366
## iter 100 value 847.749306
## final  value 847.749306
## stopped after 100 iterations
## # weights:  53
## initial  value 2087.315994
## iter  10 value 1077.121994
## iter  20 value 954.069634
## iter  30 value 920.191328
## iter  40 value 897.794757
## iter  50 value 878.480882
## iter  60 value 874.818652
## iter  70 value 870.911107
## iter  80 value 870.007017
## iter  90 value 869.743227
## iter 100 value 869.689781
## final  value 869.689781
```

```
## final  value 869.089761
## stopped after 100 iterations
## # weights:  53
## initial  value 2139.944597
## iter  10 value 1138.111335
## iter  20 value 966.522950
## iter  30 value 911.058411
## iter  40 value 872.995765
## iter  50 value 856.274307
## iter  60 value 850.888115
## iter  70 value 849.563935
## iter  80 value 848.161267
## iter  90 value 844.066155
## iter 100 value 843.352381
## final  value 843.352381
## stopped after 100 iterations
## # weights:  53
## initial  value 2167.405493
## iter  10 value 1078.108376
## iter  20 value 956.211922
## iter  30 value 927.399354
## iter  40 value 899.233088
## iter  50 value 887.853732
## iter  60 value 874.784127
## iter  70 value 862.007402
## iter  80 value 857.452542
## iter  90 value 855.471023
## iter 100 value 854.024365
## final  value 854.024365
## stopped after 100 iterations
## # weights:  53
## initial  value 2140.667866
## iter  10 value 1171.890082
## iter  20 value 982.209190
## iter  30 value 932.990562
## iter  40 value 910.094211
## iter  50 value 886.400780
## iter  60 value 872.271084
## iter  70 value 862.196547
## iter  80 value 860.020483
## iter  90 value 859.048484
## iter 100 value 858.135984
## final  value 858.135984
## stopped after 100 iterations
## # weights:  53
## initial  value 2062.560693
## iter  10 value 1133.982967
## iter  20 value 1001.724042
## iter  30 value 923.352043
## iter  40 value 894.832851
## iter  50 value 881.661073
## iter  60 value 874.571462
## iter  70 value 871.235304
## iter  80 value 869.115227
## iter  90 value 867.719137
## iter 100 value 866.557628
## final  value 866.557628
## stopped after 100 iterations
## # weights:  53
## initial  value 2285.058896
## iter  10 value 1227.089544
## iter  20 value 1077.372339
## iter  30 value 1016.970966
## iter  40 value 972.751959
## iter  50 value 939.257615
## iter  60 value 915.246129
## iter  70 value 908.954277
## iter  80 value 901.214757
## iter  90 value 898.655387
## iter 100 value 895.182722
## final  value 895.182722
## stopped after 100 iterations
## # weights:  53
```

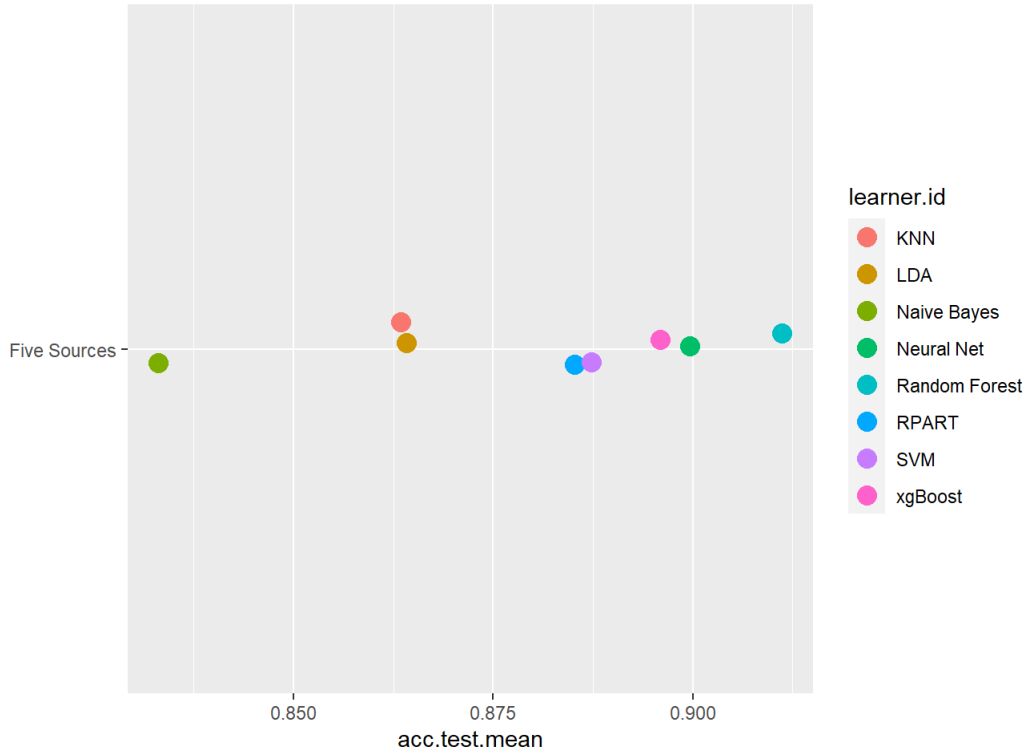
```
## initial  value 2183.634321
## iter  10 value 1099.215271
## iter  20 value 996.128494
## iter  30 value 949.717062
## iter  40 value 925.244760
## iter  50 value 909.125383
## iter  60 value 897.710834
## iter  70 value 890.200954
## iter  80 value 877.595881
## iter  90 value 874.808670
## iter 100 value 873.387235
## final  value 873.387235
## stopped after 100 iterations
```

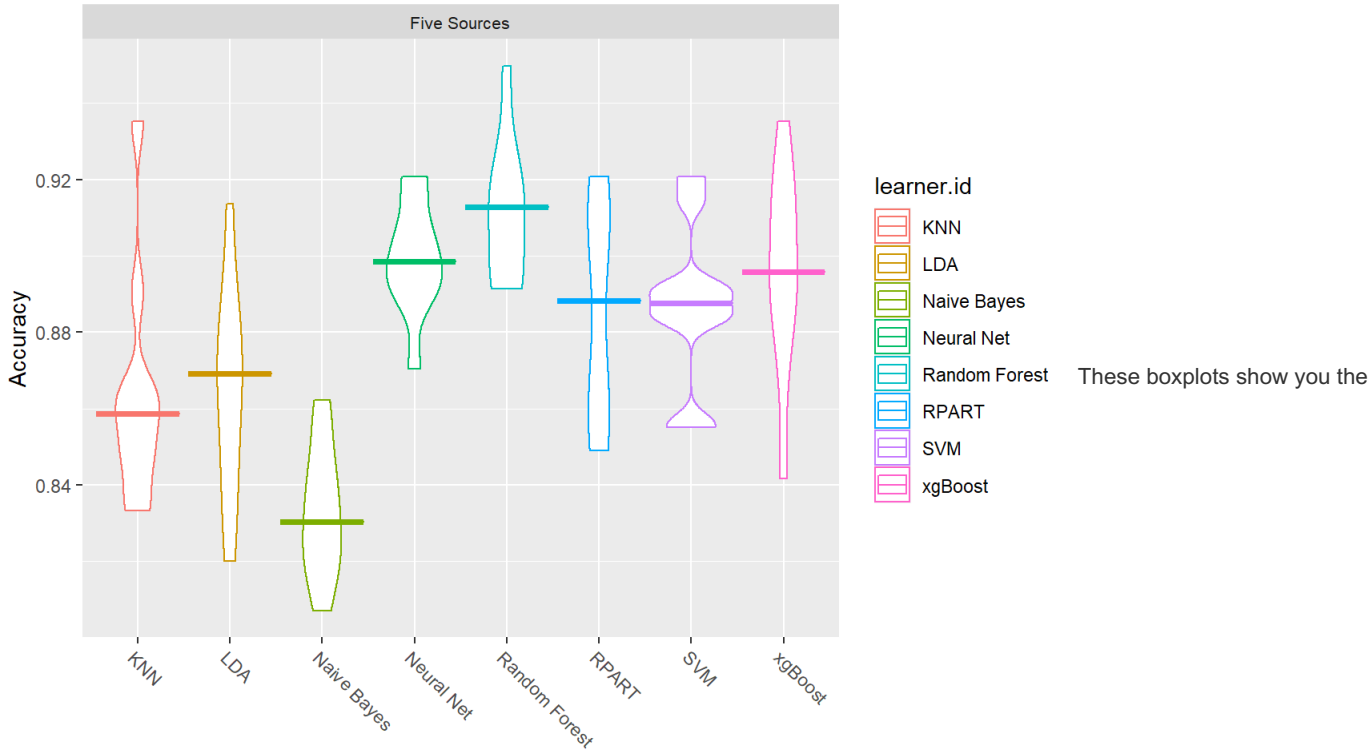



```
## # weights:  68
## initial  value 2225.636771
## iter  10 value 989.470218
## iter  20 value 493.075248
## iter  30 value 377.921724
## iter  40 value 335.570019
## iter  50 value 293.695638
## iter  60 value 262.775882
## iter  70 value 253.894457
## iter  80 value 249.758715
## iter  90 value 246.229092
## iter 100 value 244.743830
## final  value 244.743830
## stopped after 100 iterations
## # weights:  68
## initial  value 2178.625211
## iter  10 value 774.973302
## iter  20 value 448.865617
## iter  30 value 353.696698
## iter  40 value 310.456155
## iter  50 value 279.876860
## iter  60 value 267.704485
## iter  70 value 261.658008
## iter  80 value 258.583558
## iter  90 value 256.701128
## iter 100 value 255.621295
## final  value 255.621295
## stopped after 100 iterations
## # weights:  68
## initial  value 2025.191535
## iter  10 value 769.267598
```

```
## iter  20 value 532.055189
## iter  30 value 355.490048
## iter  40 value 294.056460
## iter  50 value 283.265456
## iter  60 value 272.952650
## iter  70 value 270.478545
## iter  80 value 268.361299
## iter  90 value 267.326499
## iter 100 value 266.668059
## final  value 266.668059
## stopped after 100 iterations
## # weights:  68
## initial  value 2174.802098
## iter  10 value 664.491833
## iter  20 value 451.569321
## iter  30 value 338.536605
## iter  40 value 308.484695
## iter  50 value 295.177247
## iter  60 value 286.583386
## iter  70 value 276.522746
## iter  80 value 273.534753
## iter  90 value 271.617222
## iter 100 value 270.936723
## final  value 270.936723
## stopped after 100 iterations
## # weights:  68
## initial  value 2102.763187
## iter  10 value 519.092487
## iter  20 value 369.358434
## iter  30 value 303.963908
## iter  40 value 279.604938
## iter  50 value 265.919382
## iter  60 value 257.572916
## iter  70 value 252.425120
## iter  80 value 249.196622
## iter  90 value 247.943522
## iter 100 value 247.433082
## final  value 247.433082
## stopped after 100 iterations
## # weights:  68
## initial  value 2002.190135
## iter  10 value 680.673999
## iter  20 value 399.197157
## iter  30 value 312.123226
## iter  40 value 284.553171
## iter  50 value 280.398047
## iter  60 value 276.524064
## iter  70 value 272.441798
## iter  80 value 269.228698
## iter  90 value 267.241157
## iter 100 value 265.970402
## final  value 265.970402
## stopped after 100 iterations
## # weights:  68
## initial  value 2059.917074
## iter  10 value 756.510271
## iter  20 value 453.067560
## iter  30 value 334.740021
## iter  40 value 292.501442
## iter  50 value 277.119669
## iter  60 value 272.322120
## iter  70 value 265.181948
## iter  80 value 262.279825
## iter  90 value 260.070582
## iter 100 value 258.891592
## final  value 258.891592
## stopped after 100 iterations
## # weights:  68
## initial  value 2278.889054
## iter  10 value 570.927444
## iter  20 value 357.994333
## iter  30 value 310.354079
## iter  40 value 285.365250
```

```
## iter  50 value 276.368671
## iter  60 value 273.156868
## iter  70 value 268.695690
## iter  80 value 263.393101
## iter  90 value 259.857628
## iter 100 value 258.938265
## final  value 258.938265
## stopped after 100 iterations
## # weights:  68
## initial  value 2036.712679
## iter  10 value 527.398678
## iter  20 value 364.219104
## iter  30 value 308.817531
## iter  40 value 276.616846
## iter  50 value 269.071542
## iter  60 value 264.967410
## iter  70 value 259.669680
## iter  80 value 256.338609
## iter  90 value 254.988848
## iter 100 value 254.015350
## final  value 254.015350
## stopped after 100 iterations
## # weights:  68
## initial  value 2069.702657
## iter  10 value 860.790052
## iter  20 value 449.694881
## iter  30 value 355.028303
## iter  40 value 325.983526
## iter  50 value 303.803106
## iter  60 value 295.577629
## iter  70 value 289.811130
## iter  80 value 287.616766
## iter  90 value 280.583391
## iter 100 value 277.681789
## final  value 277.681789
## stopped after 100 iterations
```





These boxplots show you the results for each method across several iterations performed by benchmarking.

Validation Set Model Comparison

ModelType	Accuracy
Random Forest	0.9111
Neural Net	0.8996
xgBoost	0.8959
SVM	0.8872
RPART	0.8852
LDA	0.8642
KNN	0.8634
Naive Bayes	0.8331

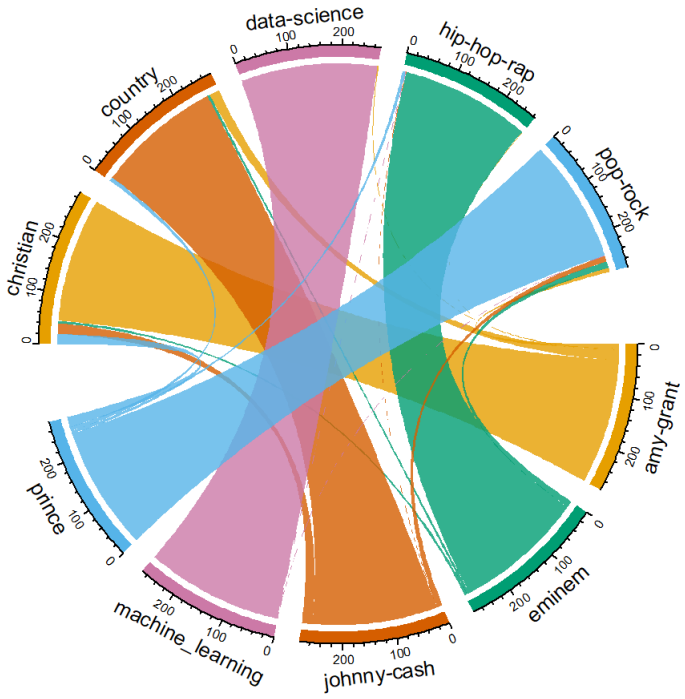
```
predictions <- getBMRPredictions(bmr)

calculateConfusionMatrix(predictions$`Five Sources`$`Random Forest`)$result  %>%
  my_kable_styling("Random Forest Confusion Matrix: Rows are True, Columns are Predictions")
```

Random Forest Confusion Matrix: Rows are True, Columns are Predictions

	christian	country	data-science	hip-hop-rap	pop-rock	-err.-
christian	246	20	2	1	8	31
country	21	246	0	1	11	33
data-science	0	0	269	1	1	2
hip-hop-rap	5	5	0	256	13	23
pop-rock	20	8	0	6	244	34
-err.-	46	33	2	9	33	123

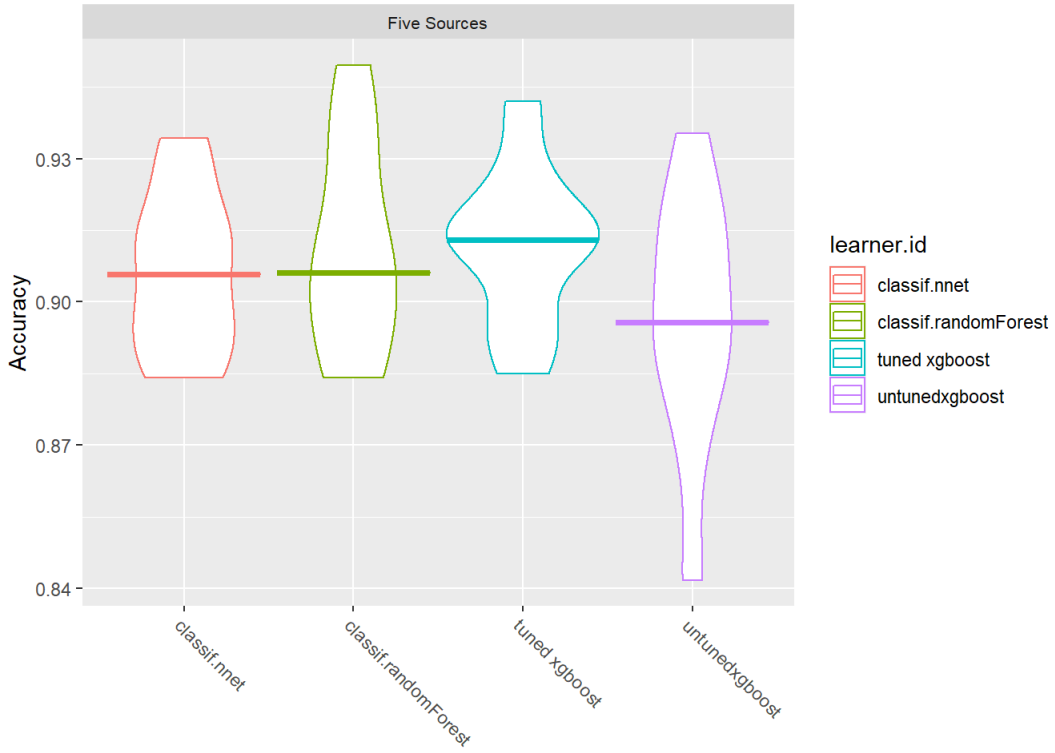
As you may expect, with only one misclassification, it is relatively easy to distinguish data science documents from song lyrics using the textual metadata features engineered. In addition, hip-hop-rap is very distinctive from other musical genres with only eight misclassifications. However, although performance is quite impressive, there is less distinction between country, Christian and pop-rock

musical lyrics. 

Now you can see how well you did on the training data. Once again the results are impressive. The smaller lines represent the misclassifications. First model data with simple models and analyze data for errors. These errors signify data points that are difficult to fit by a simple model. Then for later models, particularly focus on those hard to fit data to get them right. In the end, combine all the predictors by giving some weights to each predictor. ## Tune the model - xgBoost

```
## $booster
## [1] "gbtree"
##
## $nrounds
## [1] 19
##
## $max_depth
## [1] 4
##
## $min_child_weight
## [1] 2.102768
##
## $subsample
## [1] 0.6236946
##
## $colsample_bytree
## [1] 0.9520378
##
## $eta
## [1] 0.1259807
```

Examine the optimized parameters by looking at the output of tuneParams(). Now you want to create a new model using the tuned hyperparameters and then re-train on the training dataset.



Validation Set Model Comparison

ModelType	Accuracy
tuned xgboost	0.9119
classif.randomForest	0.9112
classif.nnet	0.9061
untunedxgboost	0.8959

The tuned xgBoost model is only slightly higher than the untuned one and is still not as accurate as random forest. ## The Real Test: New Data Now that you have your tuned model and benchmarks, you can call predict() for your top three models on the test dataset that has never been seen before. This includes five completely different sources as shown previously. Take a look at its performance and actual classifications.

```
## # weights:  68
## initial  value 2500.439406
## iter  10 value 876.420114
## iter  20 value 531.977942
## iter  30 value 412.756436
## iter  40 value 334.449379
## iter  50 value 304.202353
## iter  60 value 295.229853
## iter  70 value 286.347519
## iter  80 value 282.524065
## iter  90 value 280.438058
## iter 100 value 278.713625
## final  value 278.713625
## stopped after 100 iterations
```

```
##      acc
## 0.631068
```

```
##       acc
## 0.6201456
```

Even though random forest had a higher accuracy on the training data than the tuned xgBoost, it was slightly less accurate on the test dataset. The test accuracy for neural net is much lower than on training as well. Neural nets can be very flexible models and as a result, can overfit the training set.

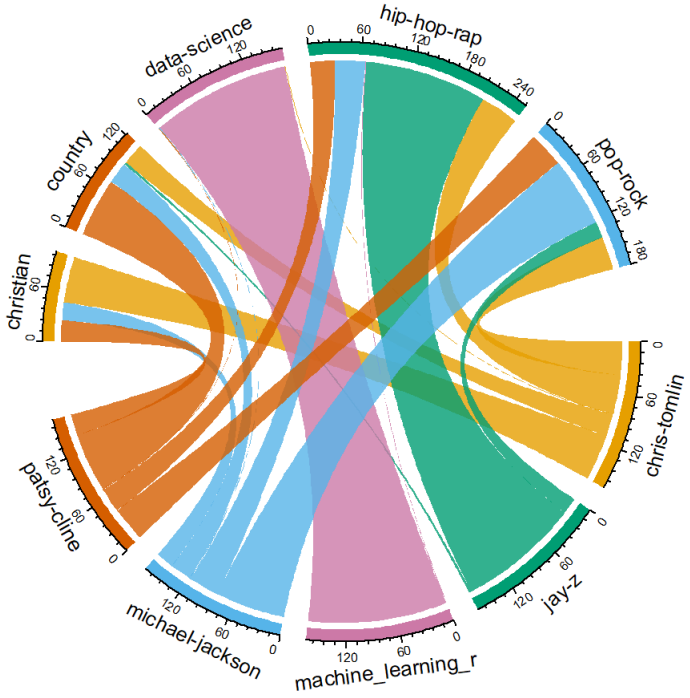
At an accuracy rate of r round(test_perf,2)*100% for tuned xgBoost, there was still a dramatic decrease in performance on the test data compared to training and tuning only made a slight improvement (using this minimal configuration!). This drop in accuracy between train

and test is widespread and is precisely why you should test your model on new data.

TEST: xgBoost Confusion Matrix: Rows are True, Columns are Predictions

	christian	country	data-science	hip-hop-rap	pop-rock	-err.-
christian	54	26	1	45	39	111
country	25	69	1	29	41	96
data-science	0	0	163	1	0	1
hip-hop-rap	0	3	0	143	19	22
pop-rock	21	25	1	36	82	83
-err.-	46	54	3	111	99	313

Predicted Relationship Between Genre and Source - Test



New chord diagram actually gives a more realistic version of the lyrical classification than shown in the original dataset! There actually isn't a one-to-one relationship between artist, and genre in real life and that flexibility is built into model. ## Conclusion We have built a model to predict the genre of a song based entirely on lyrics. We used supervised machine learning classification algorithms and trained models on a set of five different artists and five different genres. By using the mlr framework, created tasks, learners and resampling strategies to train and then tune a model(s). Then ran the model against an unseen test dataset of different artists. we were able to identify which algorithms work better with the default settings, and eventually, predict the genre of new songs that our model has never seen.