Predicting Song Popularity from Characteristics of Lyrics

Table of Contents

Introduction

I listen to a lot of music. I've never really had a musical bone in my body so I tend to gravitate toward the lyrics of a song (which is a reason why you'll never find an EDM song in my playlists). After finding a dataset on Kaggle containing the lyrics of Billboard's Hot100 year-end chart from 1964-2015, I decided that it would be interesting to use the lyrics to try to predict a given song's success on the BillBoard Hot100. The Hot100 is just a list of the most popular songs of the given year ranked from 1 (most popular) to 100 (100-most popular).

Here's what my original data set looked like.

```
original_data = read.csv("../Report1/billboard_lyrics_1964-2015.csv")
names(original_data)
```

```
## [1] "Rank"   "Song"   "Artist" "Year"   "Lyrics" "Source"
```

```
# head(original_data)
original_data[1:5,] %>% select(Rank,Song,Artist,Year) # Excluding Lyrics
```

```
##   Rank                                    Song
## 1    1                             wooly bully
## 2    2 i cant help myself sugar pie honey bunch
## 3    3               i cant get no satisfaction
## 4    4                     you were on my mind
## 5    5            youve lost that lovin feelin
##                         Artist Year
## 1 sam the sham and the pharaohs 1965
## 2                    four tops 1965
## 3           the rolling stones 1965
## 4                      we five 1965
## 5       the righteous brothers 1965
```

I decided that I wanted to come up with more columns to better interpret lyrics and had to come up with some independent variables on my own. I wrote a python script to help me. The script, which is below, imports the lyrics from the data set and finds the number of words, average word length, average rhyme length, and number of unique words in the given lyrics.

(I've moved the code to this github if you want to see the new and updated python scripts.)

https://github.com/kevinterwilliger/Lyric-Analysis

After saving as a csv, I imported the new data frame into R and used a package called SentimentAnalysis to calculate a sentiment score for each of the lyrics. Here's the code:

Here are the Xs broken down:

1. **Number of words** - This one should be pretty self-explanatory: How many words (non-unique included) does the song contain?
2. **Average Word Length** - Does the songwriter use lengthier words or smaller words? The script gets this by adding all the characters in the lyrics and dividing by the total number of words in the lyrics.
3. **Average Rhyme Length** - I had to use a handy library I found to compute this one. I'll link the github at the end, but here's a quick rundown of how it works:

- Go through lyrics word for word, using eSpeak to get the phonetic make-up of each word.
- For every word, find the longest matching rhyme sequence from the last 15 words. This is what captures the length of the rhyme. Since eSpeak converts all the words to phonetics (and I think just vowel phonetics) only vowels are compared, which is what gives the program something to match. Essentially,

the program searches for the longest matching vowel sequence for last X words, where I used an arbitrary 15 words for X.

- Find the average rhyme length by summing all the lengths and dividing by the number of rhymes. Here's the github

4. **Number of Unique Words** - I used python's dictionaries to keep track of the number of unique words in each song. Easy enough.
5. **Sentiment** - I'm not entirely sure what algorithm is used in this package, but I used the analyzeSentiment function from SentimentAnalysis to get each number. The sentiment is a score that gives the general "emotion" score of a word or series of words from -1 to 1. A negative score correlates to a more negative emotion, whether angry, sad, or both. Positive scores would work in the reverse, the higher the score, the more positive the sentiment expressed in the lyrics is.

I noticed that there were numerous repeated songs.

```
data2 = read.csv("data_clean.csv")
# get the count of every song/artist combination
proof = data2 %>%
  mutate(track = paste(Song,Artist,sep=" ")) %>%
  group_by(track) %>%
  summarise(n = n()) %>%
  filter(n > 1)
# Checking for duplicates
proof
```

```
## Registered S3 method overwritten by 'cli':
##   method     from
##   print.tree tree
```

```
## # A tibble: 195 x 2
##    track                                  n
##    <chr>                              <int>
##  1 100 pure love crystal waters           2
##  2 3 britney spears                       2
##  3 4 seasons of loneliness boyz ii men    2
##  4 adorn miguel                           2
##  5 again janet jackson                    2
##  6 all about that bass meghan trainor     2
##  7 all cried out allure featuring 112     2
##  8 all for you sister hazel               2
##  9 all i wanna do sheryl crow             2
## 10 all that she wants ace of base         2
## # ... with 185 more rows
```

So there are 195 duplicates. Out of curiosity, are these duplicates an error by whoever created the dataset?

```
data2$Year[data2$Song == "100 pure love"]
```

```
## [1] 1994 1995
```

I'm going to say that the creator did nothing wrong and that the vast majority, if not all, duplicates are caused by songs being popular for more than just one year. I think I will remove duplicates in the future.

Due to the fact that any linear regressions trying to predict the popularity of a song from the aforementioned independent variables are very underwhelming, I wrote some more python scripts (that can be found at https://github.com/kevinterwilliger/Lyric-Analysis) to collect more variables such as the key or mode of

a song and the genres of a song. I used the Spotipy library to collect an audio analysis of the song that contains the following variables:

1. **Loudness** - The overall loudness (decibels, dB) of a song.
2. **Tempo** - The beats per minute of the song.
3. **Key** - The estimated overall key of the song. Ranges in values from 0-11 mapping to pitches, using the standard Pitch Class Notation.
4. **Mode** - Binary variable where "0" is Minor and "1" is in Major
5. **Time Signiture** - The "meter" of a song is measured by the number beats in each bar. Values range from 3-7 and indicate "$\frac{3}{4}$"-"$\frac{7}{4}$".

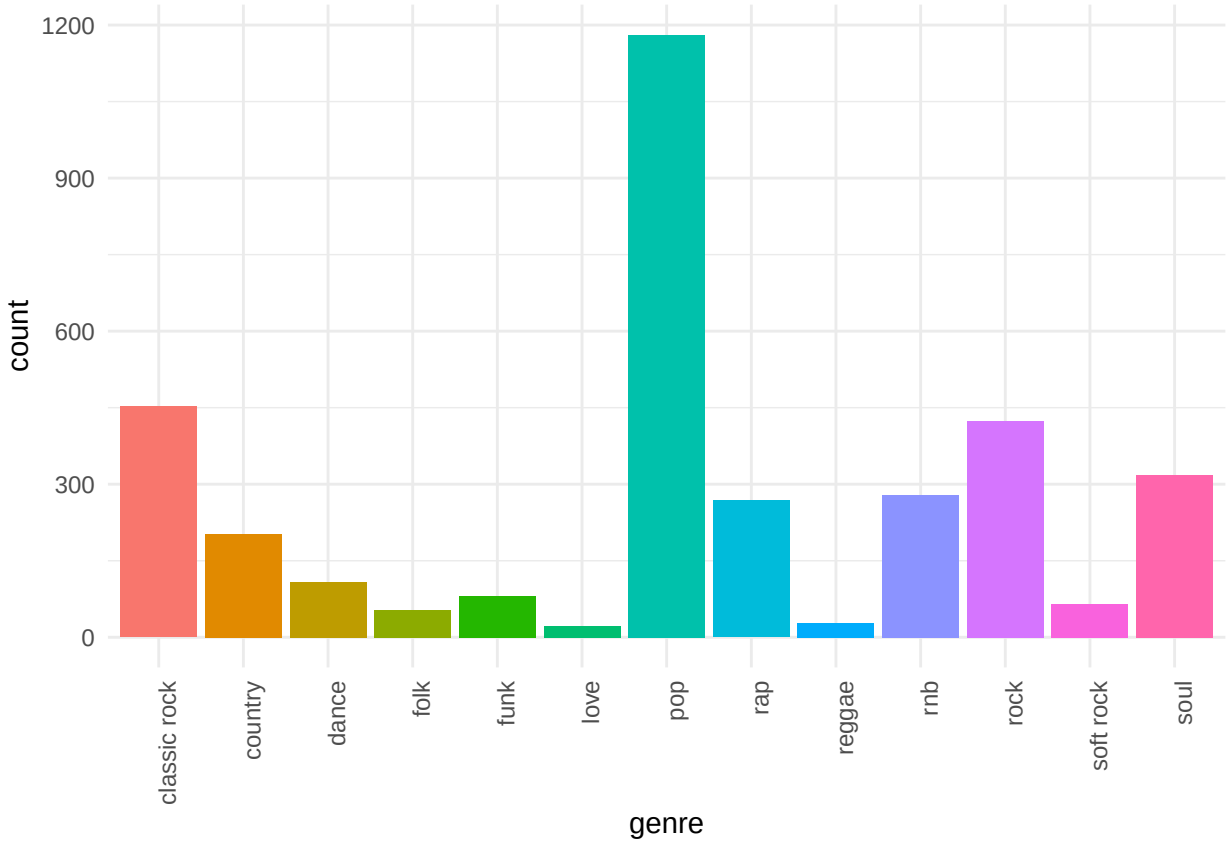
Using the Last.FM API, I was able to collect the top 5 user tags corresponding to each song. I collected 5 tags due to the fact that tags do not always contain the genre, but most songs have the specific genre in one of the 5 tags. I will explain my process for narrowing the tags down further in the report.

Logistic Regressions

Since none of my lyrics columns are good predictors of Rank, I think that using a classification model to predict genres from lyrics might prove more viable than a regression on rank.

Due to the nature of the API, I ended up dropping around 187 songs, but that leaves us with 3931 rows of song data, genre included. Lets look at the spread of the genres.

```
num_genre = clean_full %>% group_by(genre) %>% summarise(count=n()) %>% select(genre,count)

genreplot = num_genre[num_genre$count > 20,] %>%
  ggplot(., aes(x=genre, y=count, fill=genre)) +
  geom_bar(stat="identity")+theme_minimal()+
  theme(axis.text.x = element_text(angle = 90, hjust = 1),
        legend.position = 'none')

genreplot
```



There are about 1200 pop songs, but pop is such a wide ranging genre that this should be somewhat expected. Other than pop, it seems that these 17 genres are pretty all-representing.

This cleans the genres with under 20 songs and creates binary columns representing the 17 genres.

```
genre_list = num_genre[num_genre$count > 20,] %>% mutate(X = row_number())

clean_cut = clean_full[clean_full$genre %in% genre_list$genre,] %>%
  mutate(X = row_number())


binary = as.data.frame(clean_cut %>% select(X,genre) %>% model.matrix(~.,data=.))

final_data = right_join(clean_cut,binary,by="X",keep=F)
colnames(final_data)
```

```
##  [1] "X.2"              "X.1"              "Unnamed..0"
##  [4] "X"                "Rank"             "Song"
##  [7] "Artist"           "Year"             "Lyrics"
## [10] "Source"           "AverageWordLength" "AverageRhymeLength"
## [13] "NumberWords"      "UniqueWords"      "Sentiment"
## [16] "Loudness"         "Tempo"            "Key"
## [19] "Mode"             "time_signature"   "feature"
## [22] "genre"            "track"            "(Intercept)"
## [25] "genrecountry"     "genredance"       "genrefolk"
## [28] "genrefunk"        "genrelove"        "genrepop"
## [31] "genrerap"         "genrereggae"      "genrernb"
```

```
## [34] "genrerock"          "genresoft rock"     "genresoul"
```

```
head(final_data$Loudness)
```

```
## [1] -11.769 -21.836 -16.492 -11.763 -12.992 -26.048
```

Now that we have our genre columns, we can try to predict the genre from the different independent variables.

First lets try just the lyric data.

```
pop = final_data %>% select(genrepop,
                            Rank,
                            AverageWordLength,
                            AverageRhymeLength,
                            NumberWords,
                            UniqueWords,
                            Sentiment)

fit = glm(genrepop ~ Rank +
            AverageWordLength +
            AverageRhymeLength +
            NumberWords +
            UniqueWords +
            Sentiment,data=pop,family = "binomial")

summary(fit)
```

```
##
## Call:
## glm(formula = genrepop ~ Rank + AverageWordLength + AverageRhymeLength +
##     NumberWords + UniqueWords + Sentiment, family = "binomial",
##     data = pop)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.4538  -0.9439  -0.7841   1.3365   2.2356
##
## Coefficients:
##                       Estimate Std. Error z value Pr(>|z|)
## (Intercept)        -0.0754469  0.5206643  -0.145 0.884786
## Rank               -0.0074870  0.0012676  -5.907 3.49e-09 ***
## AverageWordLength   0.0972550  0.1289539   0.754 0.450739
## AverageRhymeLength  0.1621574  0.1500988   1.080 0.279992
## NumberWords         0.0016199  0.0004381   3.698 0.000217 ***
## UniqueWords        -0.0117302  0.0014862  -7.893 2.96e-15 ***
## Sentiment          -0.0254475  0.3454554  -0.074 0.941278
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 4459.5  on 3480  degrees of freedom
## Residual deviance: 4325.9  on 3474  degrees of freedom
## AIC: 4339.9
##
## Number of Fisher Scoring iterations: 4
```

From this it seems like Number of Words, Unique Words, and Rank are all significant. Let's drop the insignificant variables.

```
pop = pop %>% select(Rank,genrepop,NumberWords,UniqueWords)

fit = glm(genrepop ~ Rank + NumberWords + UniqueWords,data=pop,family=binomial)
summary(fit)
```

```
##
## Call:
## glm(formula = genrepop ~ Rank + NumberWords + UniqueWords, family = binomial,
##     data = pop)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.4130  -0.9461  -0.7843   1.3358   2.2175
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.4355607  0.1172066   3.716 0.000202 ***
## Rank        -0.0075186  0.0012669  -5.935 2.94e-09 ***
## NumberWords  0.0014610  0.0004119   3.547 0.000390 ***
## UniqueWords -0.0111974  0.0014109  -7.936 2.09e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 4459.5  on 3480  degrees of freedom
## Residual deviance: 4327.8  on 3477  degrees of freedom
## AIC: 4335.8
##
## Number of Fisher Scoring iterations: 4
```

And let's then create a confusion matrix.

```
probs = predict(fit,type="response")
prediction = predict(fit)
prediction = rep("Not Pop",nrow(pop))
prediction[probs > .5]="Pop"
real = recode(pop$genrepop,"1"="Pop","0"="Not Pop")
table(prediction,real)
```

```
##           real
## prediction Not Pop  Pop
##    Not Pop    2245 1160
##    Pop          55   21
```

The confusion matrix tells us that the model predicts that a song is not pop very accuratly, but seems unable to predict when a song is pop. Let's see if this is the case for other genres too.

Using the same logic from above, here is the fit summary for predicting whether a song is rock or not.

```
rock = final_data %>% select(genrerock,
                             Rank,
                             AverageWordLength,
                             AverageRhymeLength,
```

```
                            NumberWords,
                            UniqueWords,
                            Sentiment)

fit = glm(genrerock ~ Rank +
            AverageWordLength +
            AverageRhymeLength +
            NumberWords +
            UniqueWords +
            Sentiment,data=rock,family = "binomial")

summary(fit)
```

```
##
## Call:
## glm(formula = genrerock ~ Rank + AverageWordLength + AverageRhymeLength +
##     NumberWords + UniqueWords + Sentiment, family = "binomial",
##     data = rock)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.9798  -0.5598  -0.4763  -0.3716   2.5276
##
## Coefficients:
##                       Estimate Std. Error z value Pr(>|z|)
## (Intercept)        -2.9967693  0.7120522  -4.209 2.57e-05 ***
## Rank                0.0053326  0.0018302   2.914 0.003571 **
## AverageWordLength   0.3888353  0.1738979   2.236 0.025352 *
## AverageRhymeLength  0.1827987  0.2137877   0.855 0.392525
## NumberWords        -0.0026835  0.0007184  -3.735 0.000188 ***
## UniqueWords        -0.0001691  0.0021890  -0.077 0.938413
## Sentiment          -1.6573110  0.5038994  -3.289 0.001006 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2579.4  on 3480  degrees of freedom
## Residual deviance: 2498.9  on 3474  degrees of freedom
## AIC: 2512.9
##
## Number of Fisher Scoring iterations: 5
```

Interestingly, there are different significant variables. Rank and Number of Words are the same, but this time sentiment is significant. What is interesting is that the coefficient for sentiment is much larger than any other variable from rock or pop.

```
rock = rock %>% select(Rank,genrerock,NumberWords,Sentiment)

fit = glm(genrerock ~ Rank + NumberWords + Sentiment,data=rock,family=binomial)
summary(fit)
```

```
##
## Call:
## glm(formula = genrerock ~ Rank + NumberWords + Sentiment, family = binomial,
```

```
##     data = rock)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.9267  -0.5587  -0.4820  -0.3770   2.4828
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.2754856  0.1677819  -7.602 2.91e-14 ***
## Rank         0.0052733  0.0018224   2.894 0.003810 **
## NumberWords -0.0028971  0.0004252  -6.814 9.51e-12 ***
## Sentiment   -1.7587670  0.4975411  -3.535 0.000408 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2579.4  on 3480  degrees of freedom
## Residual deviance: 2505.3  on 3477  degrees of freedom
## AIC: 2513.3
##
## Number of Fisher Scoring iterations: 5
```

And let's then create a confusion matrix.

```
probs = predict(fit,type="response")
prediction = predict(fit)
prediction = rep("Not Rock",nrow(rock))
prediction[probs > .5]="Rock"
real = recode(rock$genrerock,"1"="Rock","0"="Not Rock")
table(prediction,real)
```

```
##           real
## prediction Not Rock Rock
##   Not Rock     3057  424
```

```
unique(prediction)
```

```
## [1] "Not Rock"
```

The last function I call is to prove that the model only predicts "Not Rock" (0) for any input in the data set. Due to that, the model can't be a good predictor if it can't predict that a rock song is a rock song. I'm going to remove pop songs from the data set to see if it helps.

```
notPop = final_data[final_data$genrepop == 0,]

rock = notPop %>% select(genrerock,
                         Rank,
                         AverageWordLength,
                         AverageRhymeLength,
                         NumberWords,
                         UniqueWords,
                         Sentiment)

fit = glm(genrerock ~ Rank + NumberWords + Sentiment, data=rock,family = binomial)
summary(fit)
```

```
##
## Call:
## glm(formula = genrerock ~ Rank + NumberWords + Sentiment, family = binomial,
##     data = rock)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.0083  -0.6984  -0.5881  -0.3856   2.2856
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.5589695  0.1693548  -3.301 0.000965 ***
## Rank         0.0026495  0.0018920   1.400 0.161416
## NumberWords -0.0031395  0.0004103  -7.651 1.99e-14 ***
## Sentiment   -1.8289528  0.5265700  -3.473 0.000514 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2198.4  on 2299  degrees of freedom
## Residual deviance: 2117.8  on 2296  degrees of freedom
## AIC: 2125.8
##
## Number of Fisher Scoring iterations: 5
```

```
probs = predict(fit,type="response")
prediction = predict(fit)
prediction = rep("Not Rock",nrow(rock))
prediction[probs > .5]="Rock"
real = recode(rock$genrerock,"1"="Rock","0"="Not Rock")
table(prediction,real)
```

```
##           real
## prediction Not Rock Rock
##   Not Rock     1876  424
```

```
unique(prediction)
```

```
## [1] "Not Rock"
```

This time, the model predicted "Rock" (1) one time, which was a false positive. I'll consider this model bust.

Now I am going to incorporate results from my Spotify API into these logisitc regressions. Hip-Hop is my favorite type of music, and tends to have pretty unique musical traits, so I'll use rap as a prediction variable out of non-pop songs.

```
aa = final_data[(final_data$Loudness != 9999) && (final_data$Tempo != 9999),]

fit = glm(genrepop ~ UniqueWords + Loudness:Tempo,data=aa,family = binomial)
summary(fit)
```

```
##
## Call:
## glm(formula = genrepop ~ UniqueWords + Loudness:Tempo, family = binomial,
##     data = aa)
##
```

```
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.3431  -0.9501  -0.8161   1.3586   2.0870
##
## Coefficients:
##                  Estimate Std. Error z value Pr(>|z|)
## (Intercept)     9.611e-02  9.746e-02   0.986    0.324
## UniqueWords    -9.067e-03  9.554e-04  -9.490  < 2e-16 ***
## Loudness:Tempo  3.943e-09  7.872e-10   5.009 5.47e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 4459.5  on 3480  degrees of freedom
## Residual deviance: 4353.9  on 3478  degrees of freedom
## AIC: 4359.9
##
## Number of Fisher Scoring iterations: 4
```

Both variables are signficantly different from 0 but their coefficients are so small that any change in the variable has negligible effect. It is interesting that unique words has a negetive coefficient, though. Maybe because Pop songs often repeat the same phrase or word often.

And the confusion table.

```
probs = predict(fit,type="response")
prediction = predict(fit)
prediction = rep("Not Pop",
                 nrow(aa[(aa$Loudness != 9999)&&(aa$Tempo != 9999),]))

prediction[probs > .5]="Pop"
real = recode(aa$genrepop,"1"="Pop","0"="Not Pop")
table(prediction,real)
```

```
##           real
## prediction Not Pop  Pop
##    Not Pop    2276 1172
##    Pop          24    9
```

```
# Percent Correct
mean(prediction==real)
```

```
## [1] 0.6564206
```

```
# False Positive Rate
20/23
```

```
## [1] 0.8695652
```

```
# True Positive Rate
2540/2560
```

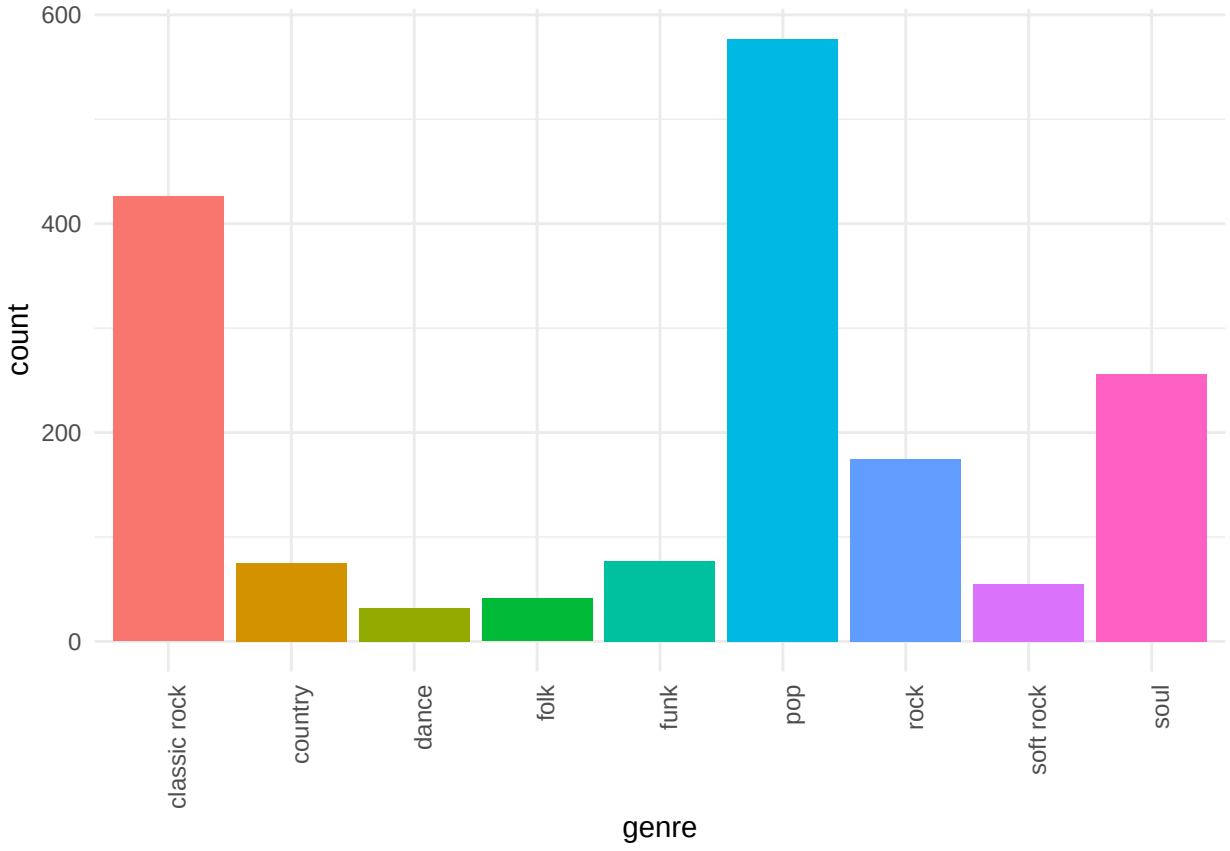
```
## [1] 0.9921875
```

For the sake of shortness, I ran a bunch of different combinations and this is the best I could predict. The model rarely predicts that a song is "Pop", but has a high false positive rate. Because the model mostly outputs "Not Pop" it has a high true positive rate - 66% of the data was "Not Pop".

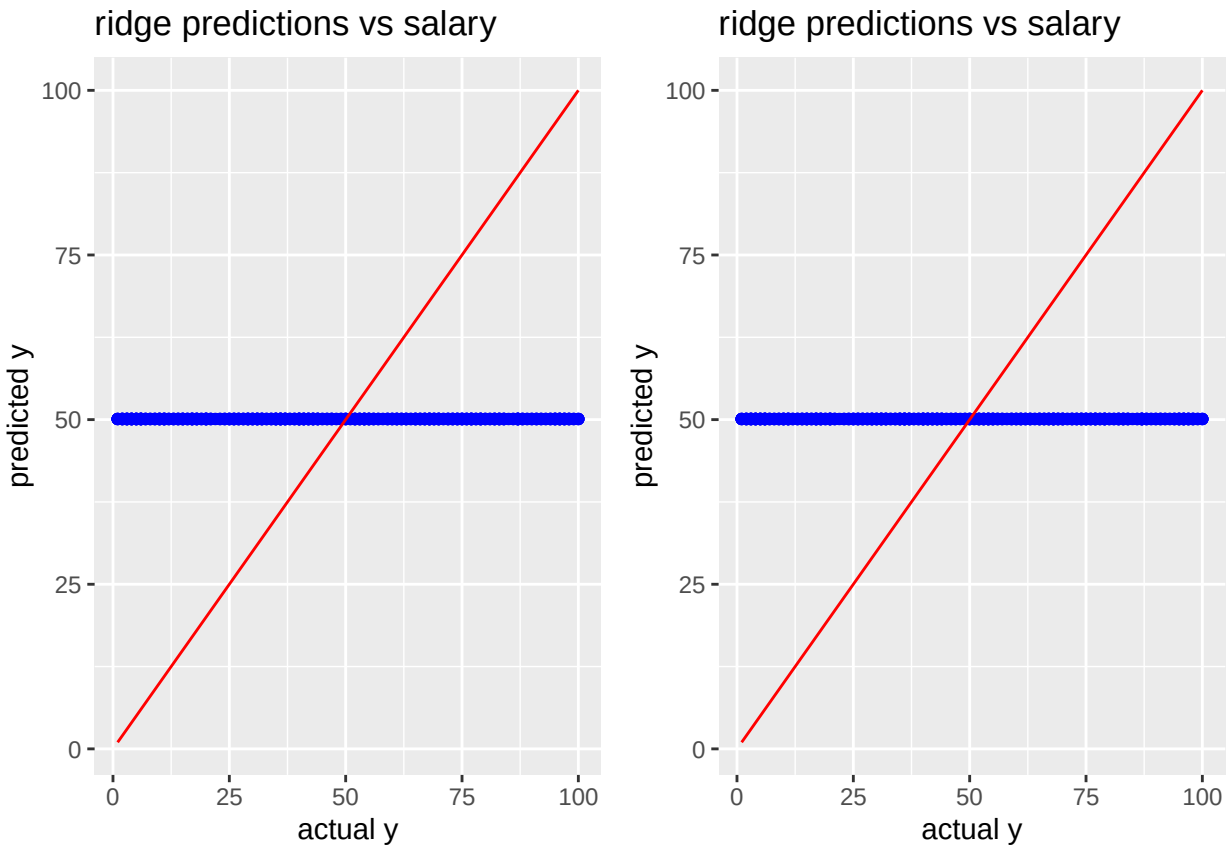
Ridge

This data has less than half of the observations as the complete dataset, but looking at the histogram of genres, we should have enough to perform classification of pop songs.

```
num_genre = current_data %>% group_by(genre) %>%
  summarise(count=n()) %>%
  select(genre,count)

genreplot = num_genre[num_genre$count > 20,] %>%
    ggplot(., aes(x=genre, y=count, fill=genre)) +
    geom_bar(stat="identity")+theme_minimal()+
    theme(axis.text.x = element_text(angle = 90, hjust = 1),
          legend.position = 'none')

genreplot
```



I already ran ridge and lasso models on a linear regression trying to predict Rank. Here's the prediction results plotted against the actual for both.



Obviously, Rank is not going to work for any kind of regression. I think it's due the to fact that Rank is uniformly distributed. The models can minimize their RSS and objective functions by setting the intercept to 50 and the coefficients to such small numbers that the prediction will always be around 50. It's unfortunate, hopefully non-linear models will work better.

Either way, I used a classification variable for the ridge and lasso models.

I let my training data include the majority of the data. Here's the distribution of genres among the two sets.

```
train_ = current_data %>%
  select(genre,Rank,c(8:18)) %>%
  sample_frac(0.6)
test_ = current_data %>%
  select(genre,Rank,c(8:18)) %>%
  setdiff(train_)
par(mfrow=c(2,1))

train_genres = train_ %>% group_by(genre) %>%
  summarise(count=n()) %>%
  filter(count > 20) %>%
    ggplot(., aes(x=genre, y=count, fill=genre)) +
    geom_bar(stat="identity")+theme_minimal()+
    theme(axis.text.x = element_text(angle = 90, hjust = 1),
          legend.position = 'none') +
    ggtitle("Distribution of Genres in Train Data")

test_genres = test_ %>% group_by(genre) %>%
  summarise(count=n()) %>%
```

```
  filter(count > 20) %>%
    ggplot(., aes(x=genre, y=count, fill=genre)) +
    geom_bar(stat="identity")+theme_minimal()+
    theme(axis.text.x = element_text(angle = 90, hjust = 1),
          legend.position = 'none') +
    ggtitle("Distribution of Genres in Test Data")

grid.arrange(train_genres, test_genres, ncol=2)
```

This following chunk creates a binary column for whether a song is of genre "Pop" or not. I use "-1" instead of "0" for better results.

```
train_$IsPop = ifelse(train_$genre == "pop",1,-1)
test_$IsPop = ifelse(test_$genre == "pop",1,-1)
```

Given that Rank proves to be unpredictable through linear models, I will preform ridge and lasso regressions on the binary variable, IsPop.

```
x = model.matrix(IsPop~., train)[,-1]
y = train$IsPop
grid = 10^seq(10, -2, length = 100)

ridge.cv = cv.glmnet(x,y,alpha=0)
```

After running the cross-validated model, the λ that minimizes the RSS/objective function is:

```
## [1] 0.1341999
```

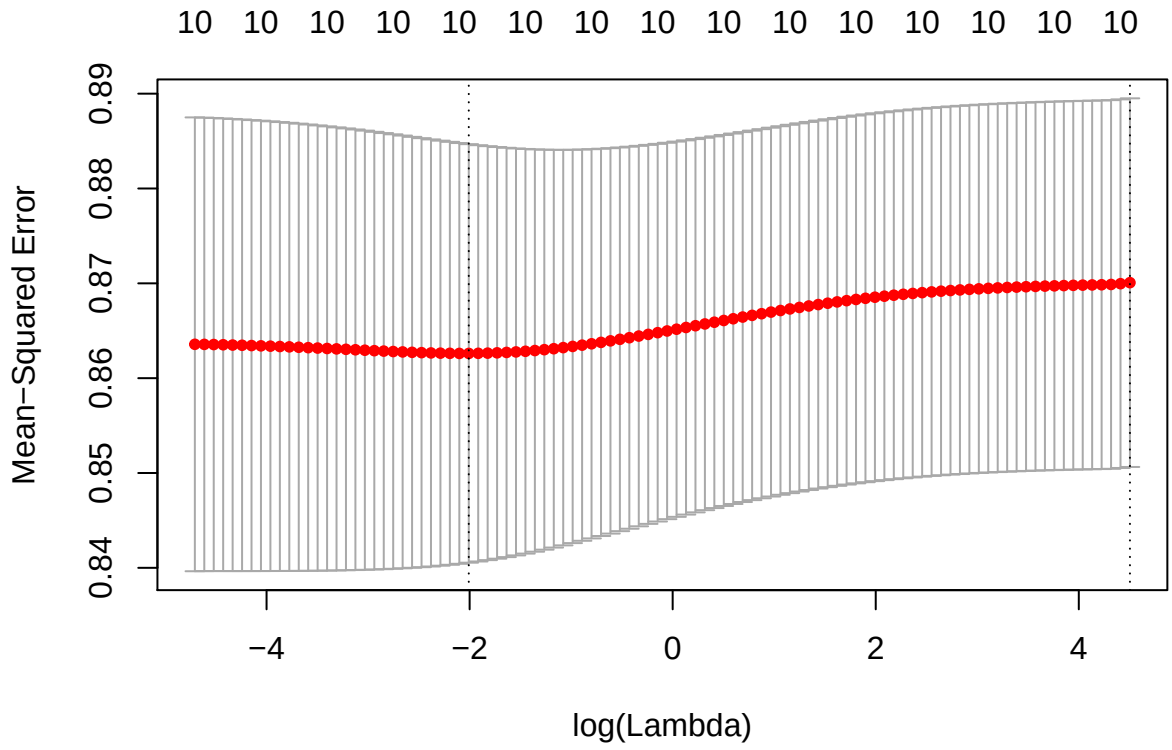
In practice, we use the best lambda 1 standard error away from the minimum λ.

```
bestlam = ridge.cv$lambda.1se
bestlam
```

```
## [1] 90.37225
```

Honestly, it strikes me as odd that the two λ's are so far apart. I wonder why.

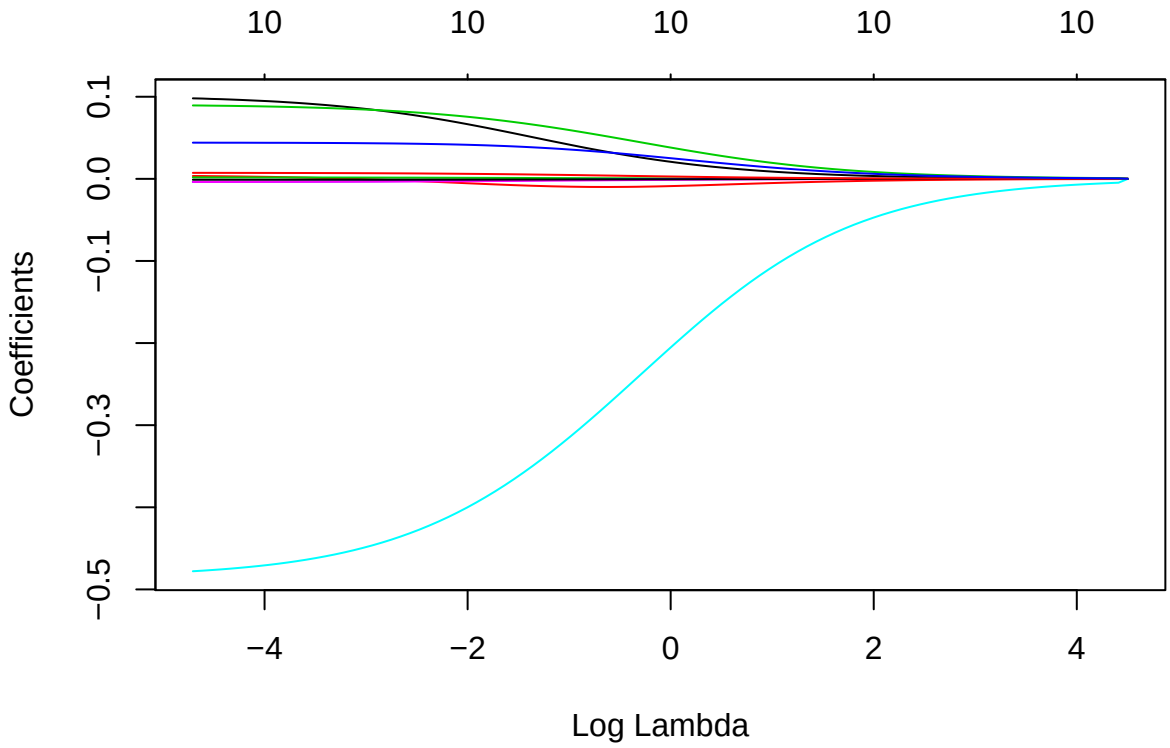
```
plot(ridge.cv)
```



Well that would be why.

Running a normal Ridge Regression, here's how the coefficients vary as λ increases:

```
ridge = glmnet(x,y,alpha=0)
plot(ridge,xvar="lambda")
```



And here are the coefficients for the λ one SE from the minimum λ:

```
predict(ridge.cv, type = "coefficients", s = bestlam)
```

```
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                                 1
## (Intercept)         -3.625349e-01
## AverageWordLength    2.882727e-38
## AverageRhymeLength -2.024538e-38
## NumberWords          9.566828e-40
## UniqueWords          3.575618e-40
## Sentiment           -4.263566e-37
## Loudness            -2.057485e-39
## Tempo               -1.188563e-39
## Key                  4.788017e-39
## Mode                 7.456503e-38
## time_signature       5.621368e-38
## feature              .
```

Using the Test subset to generate predictions from the model, we can generate the truth table of the model.

```
x_train = model.matrix(IsPop~., train)[,-1]
x_test = model.matrix(IsPop~., test)[,-1]
y_train = train$IsPop
y_test = test$IsPop


ridge_pred = predict(ridge.cv, s = bestlam, newx = x_test)
```

```
prediction = rep("Not Pop",nrow(ridge_pred))
prediction[ridge_pred > 0]="Pop"
real = recode(test$IsPop,"1"="Pop","-1"="Not Pop")
table(prediction,real)
```

```
##           real
## prediction Not Pop Pop
##    Not Pop     481 235
```

If you remember the second report, I ran into this issue, where the model only predicts "Not Pop". Hopefully Lasso might prove more helpful.

In comparison to my Logit regressions, I get the same outcome, so comparing them seems uneccessary.

Lasso

Using the same binary column, IsPop, let's see if a Lasso model proves more useful.

```
x = model.matrix(IsPop~., train)[,-1]
y = train$IsPop
grid = 10^seq(10, -2, length = 100)

lasso.cv = cv.glmnet(x,y,alpha=1)
```

After running the cross-validated model, the λ that minimizes the RSS/objective function is:

```
## [1] 0.001992826
```

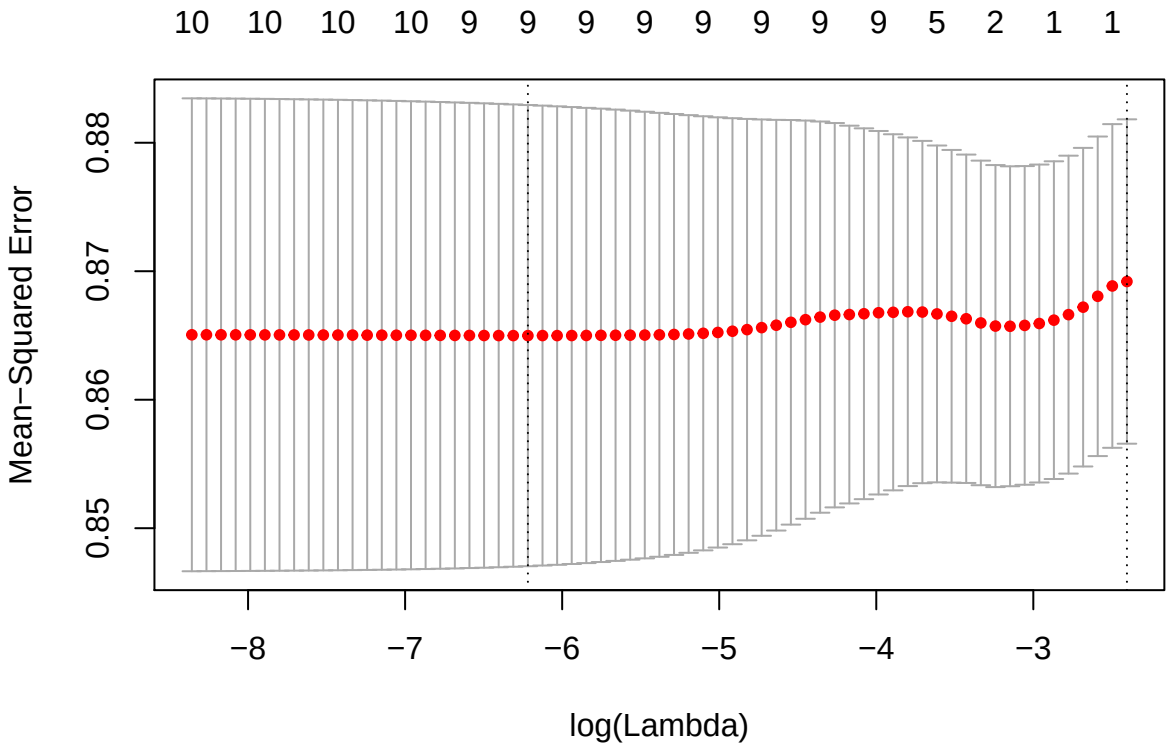
In practice, we use the best lambda 1 standard error away from the minimum λ.

```
bestlam = lasso.cv$lambda.1se
bestlam
```

```
## [1] 0.09037225
```

Again, these λs are strange, this time they are incredibly close together, but very small.

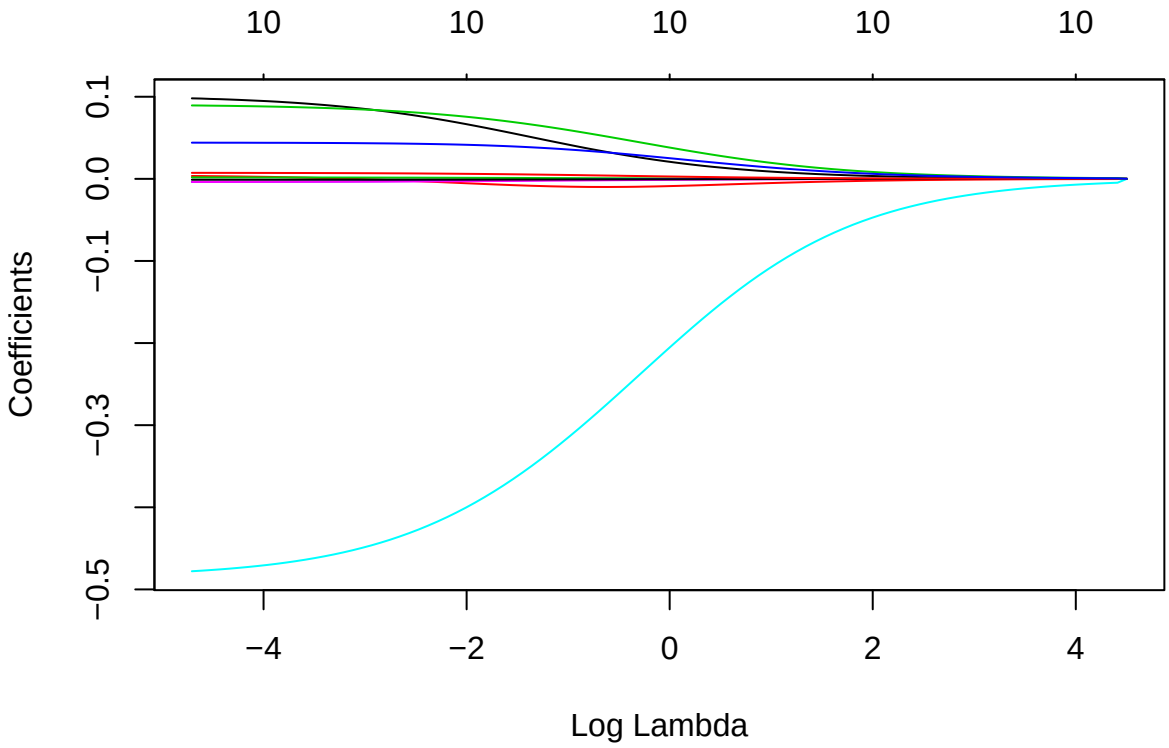
```
plot(lasso.cv)
```



I'm curious why there is a bump just before the minimum λ, but this plot does explain why the λs are so small, the minimizing λ is $\sim e^{-3}$ and the λ 1 SE from the minimum is not much larger.

Running a normal Ridge Regression, here's how the coefficients vary as λ increases:

```
lasso = glmnet(x,y,alpha=1)
plot(ridge,xvar="lambda")
```



And here are the coefficients for the λ one SE from the minimum λ:

```
predict(lasso.cv, type = "coefficients", s = bestlam)
```

```
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                              1
## (Intercept)        -0.3625349
## AverageWordLength   .
## AverageRhymeLength  .
## NumberWords         .
## UniqueWords         .
## Sentiment           .
## Loudness            .
## Tempo               .
## Key                 .
## Mode                .
## time_signature      .
## feature             .
```

Well. That's... I'm not sure what I expected, but this isn't to far from it. I'll finish the rest of the section, but everything else I do from here will be meaningless (because the model only includes the intercept).

Using the Test subset to generate predictions from the model, we can generate the truth table for the model.

```
x_train = model.matrix(IsPop~., train)[,-1]
x_test = model.matrix(IsPop~., test)[,-1]
y_train = train$IsPop
y_test = test$IsPop
```

```
lasso_pred = predict(lasso.cv, s = bestlam, newx = x_test)
lasso_prediction = rep("Not Pop",nrow(lasso_pred))
lasso_prediction[lasso_pred > 0]="Pop"
real = recode(test$IsPop,"1"="Pop","-1"="Not Pop")
table(lasso_prediction,real)
```

```
##                 real
## lasso_prediction Not Pop Pop
##          Not Pop     481 235
```

This is actually funny. The coefficients of the ridge model are so negligible that if you didn't factor them in, the predictions wouldn't change. I know this because we can run the following code to show the predictions are the same for the intercept-only lasso model and the ridge model.

```
unique(prediction == lasso_prediction)
```

```
## [1] TRUE
```

So linear models prove useless for predicting Rank and IsPop. Next, our first non-linear model, binary trees!

Decision Tree

I've run both of the following trees many times. Sometimes the algorithm never splits and leaves me with an empty tree, other times it splits by 5 or 6 variables. I'll write the following assuming the trees are non-empty.

First to create our tree. I'm going to try to predict IsPop.

```
tree_train = train_ %>% select(-c(Rank,genre))
tree_train$IsPop = as.factor(ifelse(tree_train$IsPop >= 0, "Pop","Not Pop"))
tree_test = test_ %>% select(-c(Rank,genre))
tree_test$IsPop = as.factor(ifelse(tree_test$IsPop >= 0, "Pop","Not Pop"))

t2 = tree(IsPop ~ .,data = tree_train)
summary(t2)
```

```
##
## Classification tree:
## tree(formula = IsPop ~ ., data = tree_train)
## Variables actually used in tree construction:
## [1] "NumberWords" "Tempo"       "UniqueWords"
## Number of terminal nodes:  4
## Residual mean deviance:  1.213 = 1296 / 1069
## Misclassification error rate: 0.302 = 324 / 1073
```

Hey this model has a split! Let's see if it can predict anything!

```
tree_pred = predict(t2, tree_test, type = "class")
table(tree_pred, tree_test$IsPop)
```

```
##
## tree_pred Not Pop Pop
##   Not Pop     433 204
##   Pop          48  31
```

Not terrible, I think.

```
## <simpleError in tree.prune(structure(list(frame = structure(list(var = structure(c(4L, 1L, 8L, 5L, 1
```

```
## [1] "Tree is illigitimate, whatever that means"
```

I'm not sure, but I keep getting an error saying that the tree is not a "legitimate" tree. I'll use the if statement to catch it, hopefully the tree is legitimate.

I've been getting different splits on the second tree everytime I run, so I'll still do the extra credit, even if it's seemingly pointless. Hopefully reading this has been more fun than writing it haha. :(.

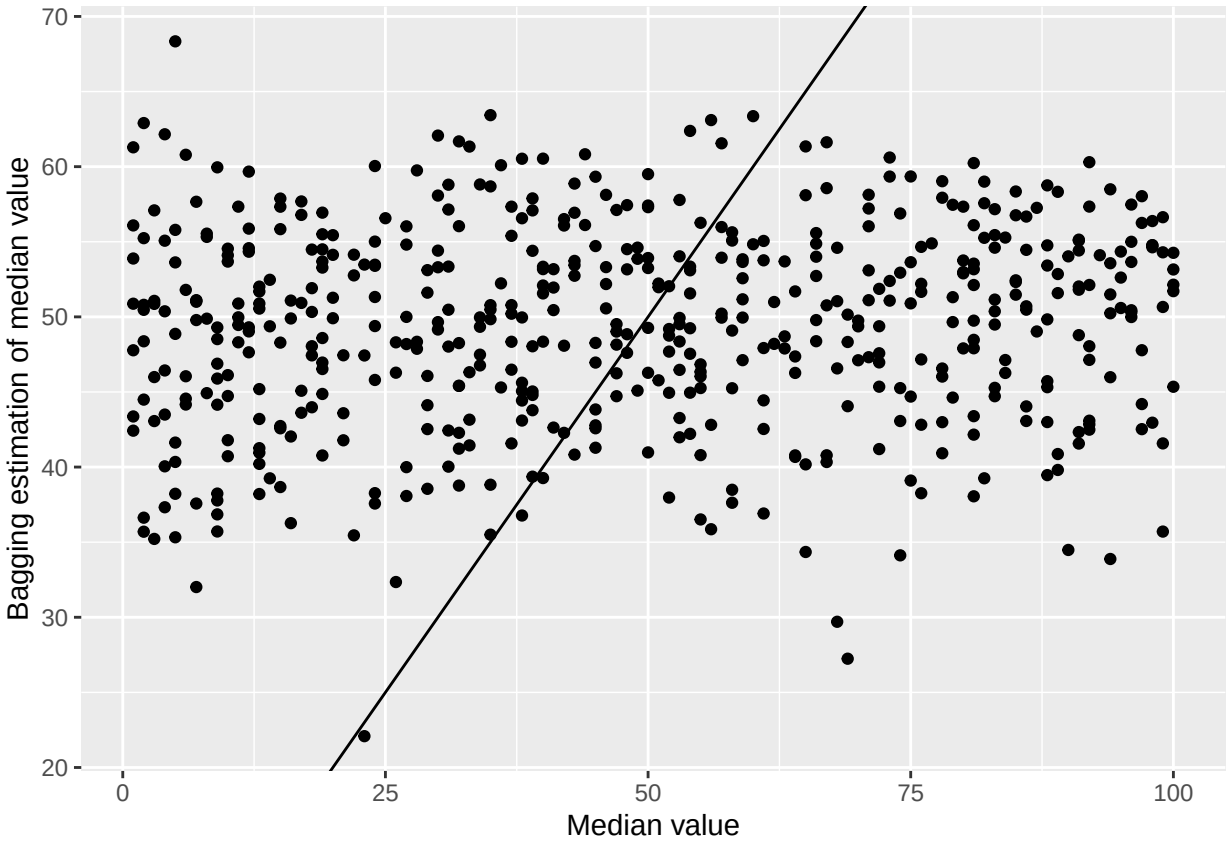
Bagging

I use the train and test subsets to help in validation.

```
set.seed(1)
train = data_topartist %>%
  select(-c(1:4,6:10,track,isTopArtist,genre)) %>%
  sample_frac(.7)
test = data_topartist %>%
  select(-c(1:4,6:10,track,isTopArtist,genre)) %>%
  setdiff(train)
```

```
bag = randomForest(Rank ~., data=train,
                           mtry=ncol(train)-1,
                           importance=TRUE,
                           ntree=400)
```

```
pred = predict(bag,newdata = test)

ggplot() +
    geom_point(aes(x = test$Rank, y = pred)) +
    geom_abline() +
    labs(x="Median value", y="Bagging estimation of median value")
```



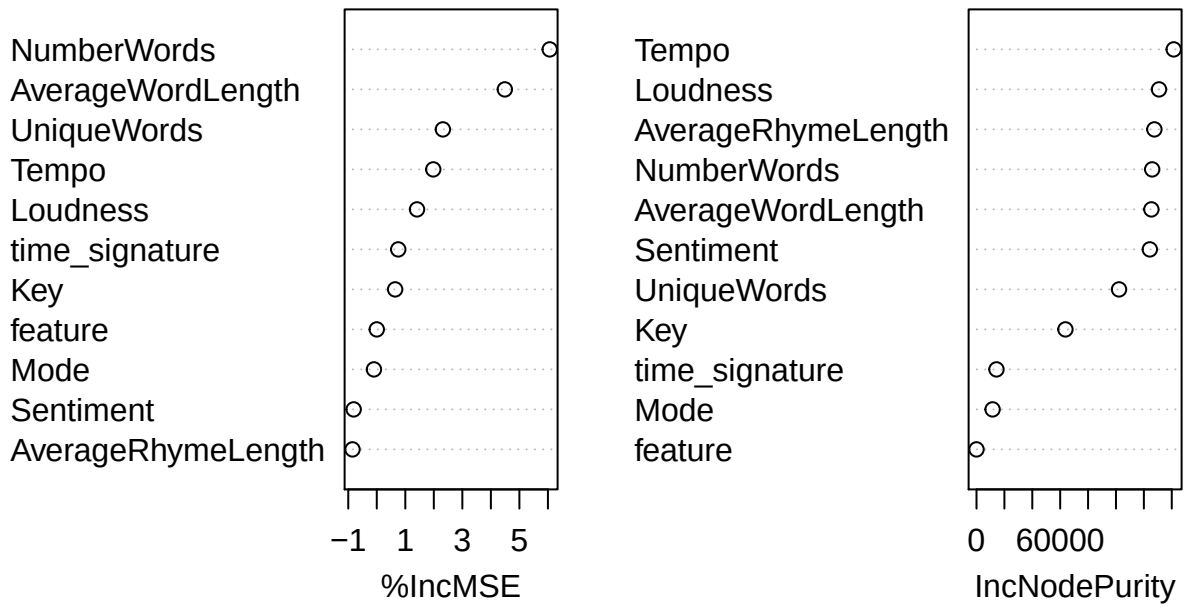
Yikes, it's not great.

```
mse = round(mean((pred-test$Rank)^2),2)
MSEs = data.frame("MSE"= mse,row.names = "Bagging")
mse
```

```
## [1] 874.05
```

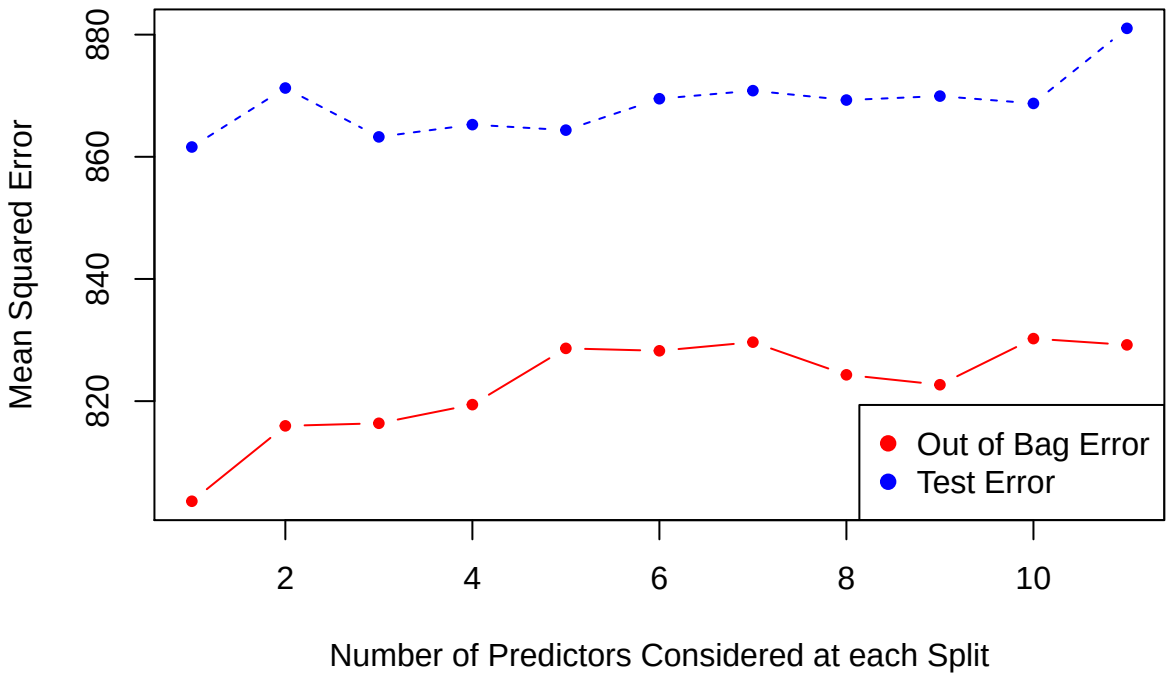
I used a classification model for the Lasso/Ridge models above, but after trying a classification for this report, I found that my results were the same, where the models only predict one side of the classification. From the above graph, I can safely guess that the results of this bagging model and the Lasso/Ridge models are all equally garbage.

```
varImpPlot(bag)
```



For as garbage the model is, it is pretty interesting to see which variables are deemed more "important" by the algorithm. It seems odd to me that Tempo is the bottom of the left-hand graph and yet the top of the right-hand side.

```
oob.err<-double(ncol(train)-1)
test.err<-double(ncol(train)-1)
for(mtry in 1:(ncol(train)-1))
{
  rf=randomForest(Rank ~ . , data = train, mtry=mtry, ntree=400)
  oob.err[mtry] = rf$mse[400]

  pred<-predict(rf,test)
  test.err[mtry]= with(test, mean( (Rank - pred)^2))
}

matplot(1:mtry , cbind(oob.err,test.err), pch=20 , col=c("red","blue"),type="b",ylab="Mean Squared Erro
legend("bottomright",legend=c("Out of Bag Error","Test Error"),pch=19, col=c("red","blue"))
```

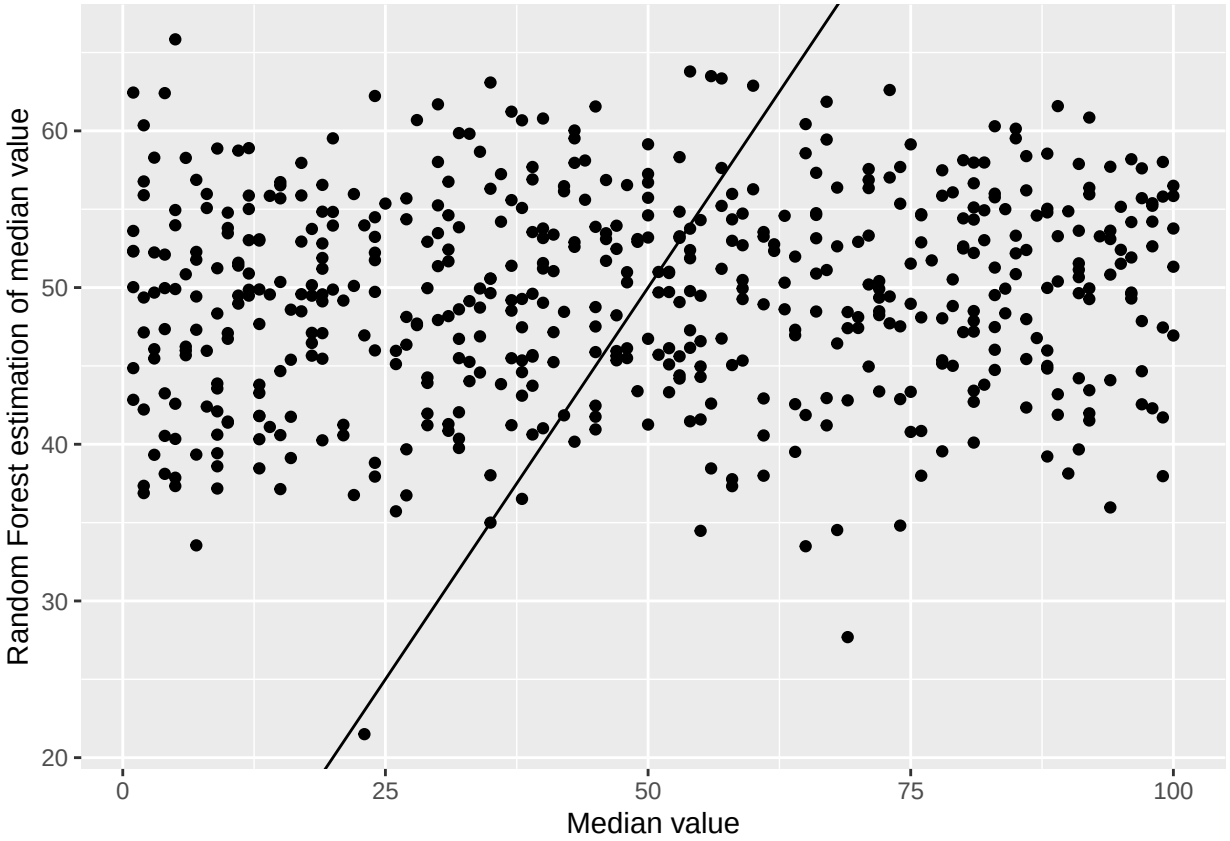


Uhhhhh, I'm fairly sure that they're not meant to increase, but I guess this follows the trend of my reports haha. I'll use mtry= 7 as my best model because I don't think 1 is my best option...

Random Forest

```
rf=randomForest(Rank ~ . , data = train, mtry=7, ntree=400)
```

```
pred<-predict(rf,test)

plot = ggplot() +
    geom_point(aes(x = test$Rank, y = pred)) +
    geom_abline() +
    labs(x="Median value", y="Random Forest estimation of median value")
plot
```



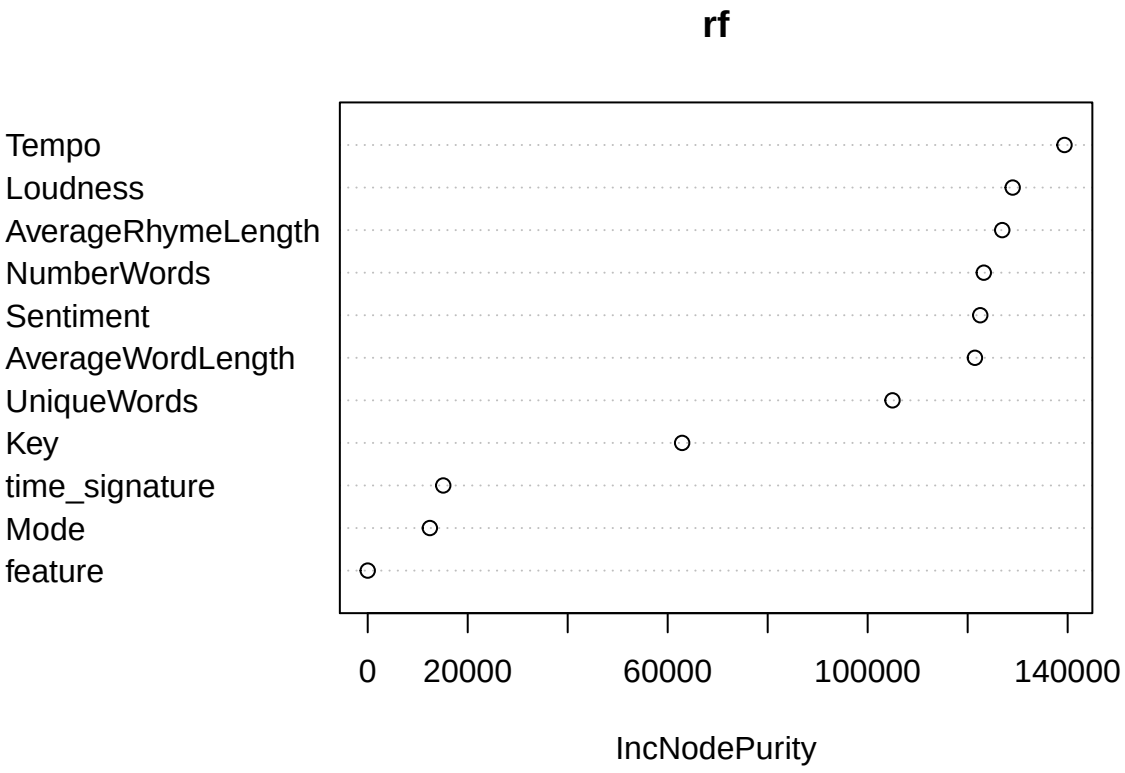
Very similar to the bagging result, though perhaps that shouldn't be a surprise.

```
mse = round(mean((test$Rank-pred)^2),2)
MSEs["Random Forest","MSE"] = mse
mse
```

```
## [1] 868.76
```

Much like the above graph and bagging results, this model sucks.

```
varImpPlot(rf)
```



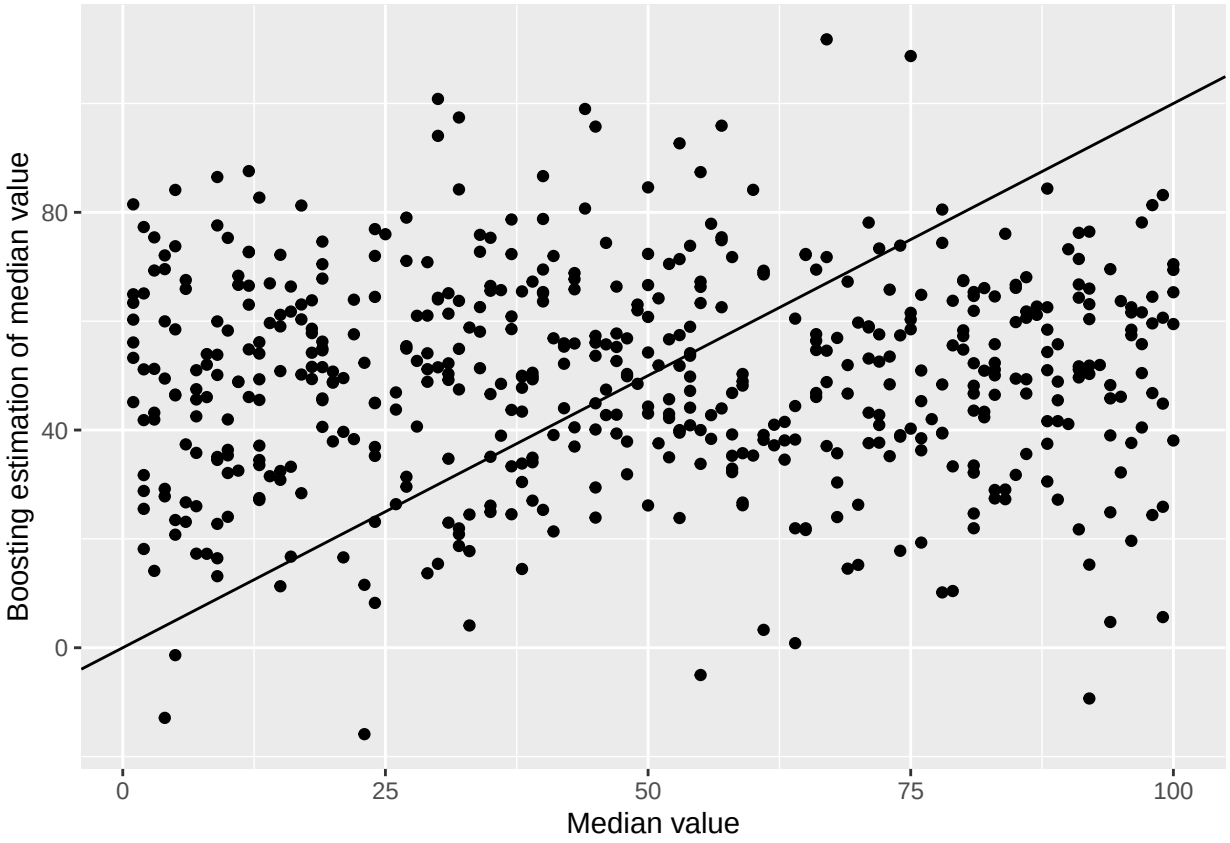
It seems that the important variables in the randomForest are similar to the important variables in the bagging results, which, I guess, shouldn't be too surprising either.

Boosting

```
boost = gbm(Rank~.,
                    data = train,
                    distribution = "gaussian",
                    n.trees = 5000,
                    interaction.depth = 4)
```

```
## Warning in gbm.fit(x = x, y = y, offset = offset, distribution =
## distribution, : variable 11: feature has no variation.
```

```
pred<-predict(boost,test,n.trees=5000)

ggplot() +
    geom_point(aes(x = test$Rank, y = pred)) +
    geom_abline() +
    labs(x="Median value", y="Boosting estimation of median value")
```



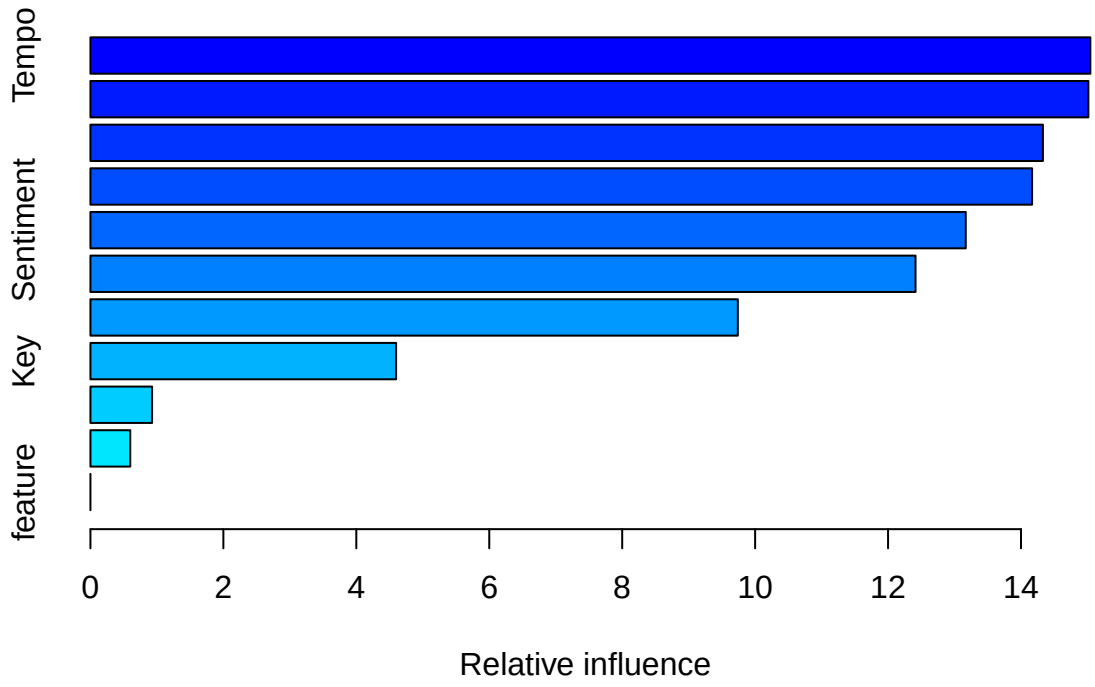
oof.

```
mse = round(mean((test$Rank-pred)^2),2)
MSEs["Boosting","MSE"] = mse
mse
```

```
## [1] 1197.84
```

Wow it's actually worse than the other models. Unless I'm mistaken, I thought boosting was supposed to be the best model out of all the previous tree models. Was it because we didn't tune it?

```
summary(boost)
```




```
##                                    var    rel.inf
## Tempo                            Tempo 15.0433463
## Loudness                      Loudness 15.0135538
## AverageRhymeLength AverageRhymeLength 14.3296057
## AverageWordLength   AverageWordLength 14.1669171
## Sentiment                    Sentiment 13.1686641
## NumberWords                NumberWords 12.4132247
## UniqueWords                UniqueWords  9.7396093
## Key                                Key  4.5982627
## Mode                              Mode  0.9276763
## time_signature         time_signature  0.5991402
## feature                       feature  0.0000000
```

It's interesting that Sentiment is the most important variable, as opposed to Tempo.

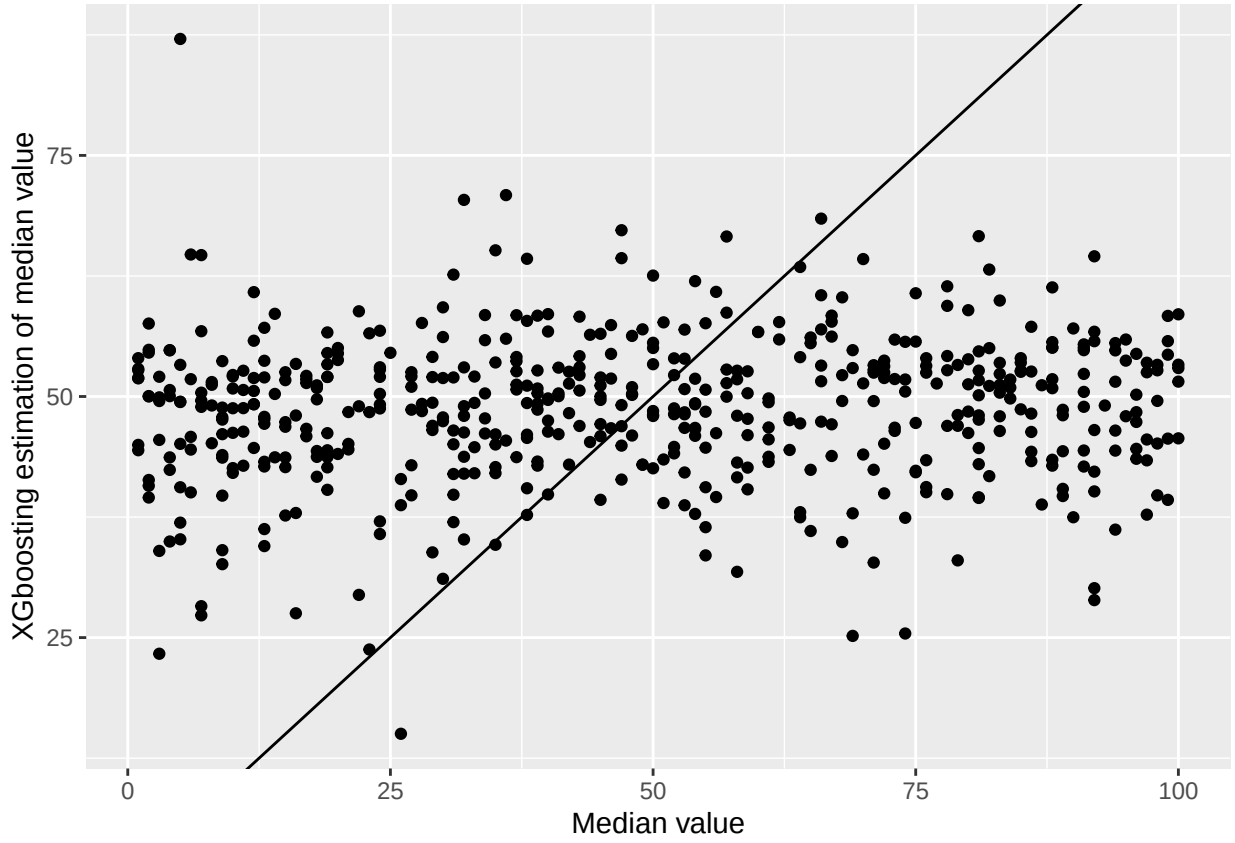
XGBoost

```
Y_train <- as.matrix(train[,"Rank"])
X_train <- as.matrix(train[!names(train) %in% c("Rank")])
dtrain <- xgb.DMatrix(data = X_train, label = Y_train)

X_test <- as.matrix(test[!names(train) %in% c("Rank")])

xgboost = xgboost(data=dtrain,
                  max_depth=5,
                  eta = 0.1,
```

```
                      nrounds=40,
                      lambda=0,
                      print_every_n = 10,
                      objective="reg:linear")
```

```
## [1]  train-rmse:52.702873
## [11] train-rmse:29.735592
## [21] train-rmse:24.265808
## [31] train-rmse:22.423471
## [40] train-rmse:21.383595
```

```
pred = predict(xgboost ,X_test)

ggplot() +
    geom_point(aes(x = test$Rank, y = pred)) +
    geom_abline() +
    labs(x="Median value", y="XGboosting estimation of median value")
```



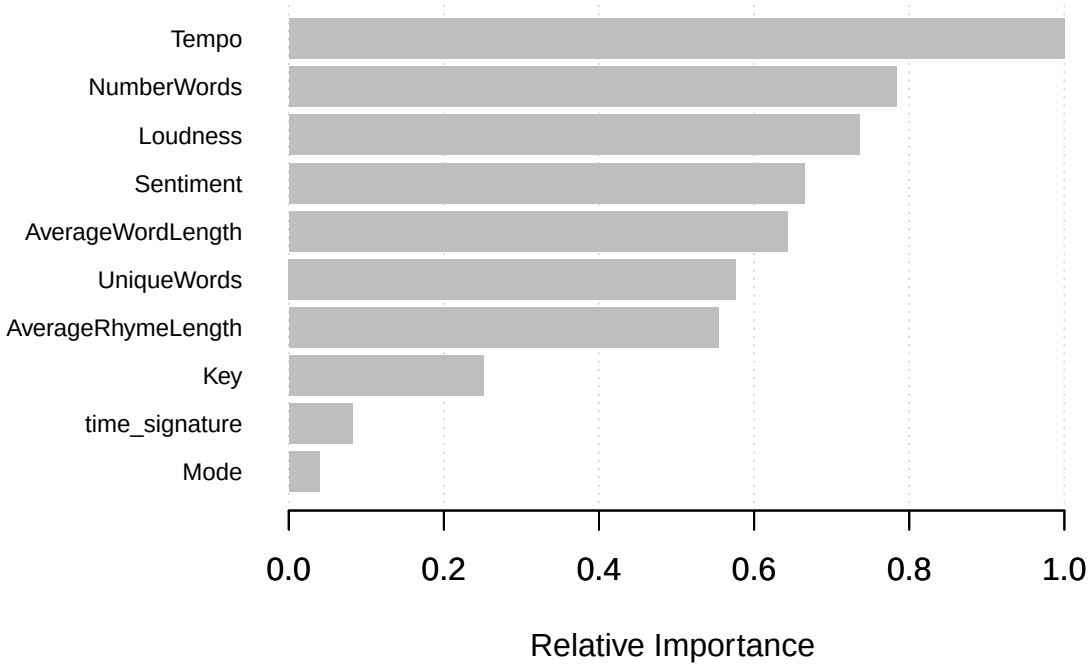
Hey wait that looks somewhat better than the previous graphs, though still underwhelming.

```
mse = round(mean((pred - test$Rank)^2),2)
MSEs["xgboost","MSE"] = mse
mse
```

```
## [1] 886.04
```

Well, it's lower than the regular boosting MSE, but still not significantly less than any of the other models.

```r
importance <- xgb.importance(colnames(X_train),model=xgboost)
xgb.plot.importance(importance, rel_to_first=TRUE, xlab="Relative Importance")
```



Oh that's actually interesting that AverageWordLength takes the most important variable spot from Tempo/Sentiment. My sneaking suspicion is that the models are so inconsistent of predictors that the true difference in importance of each variable is so slim that the most important variable is not much more important than the least important.

Neural Nets

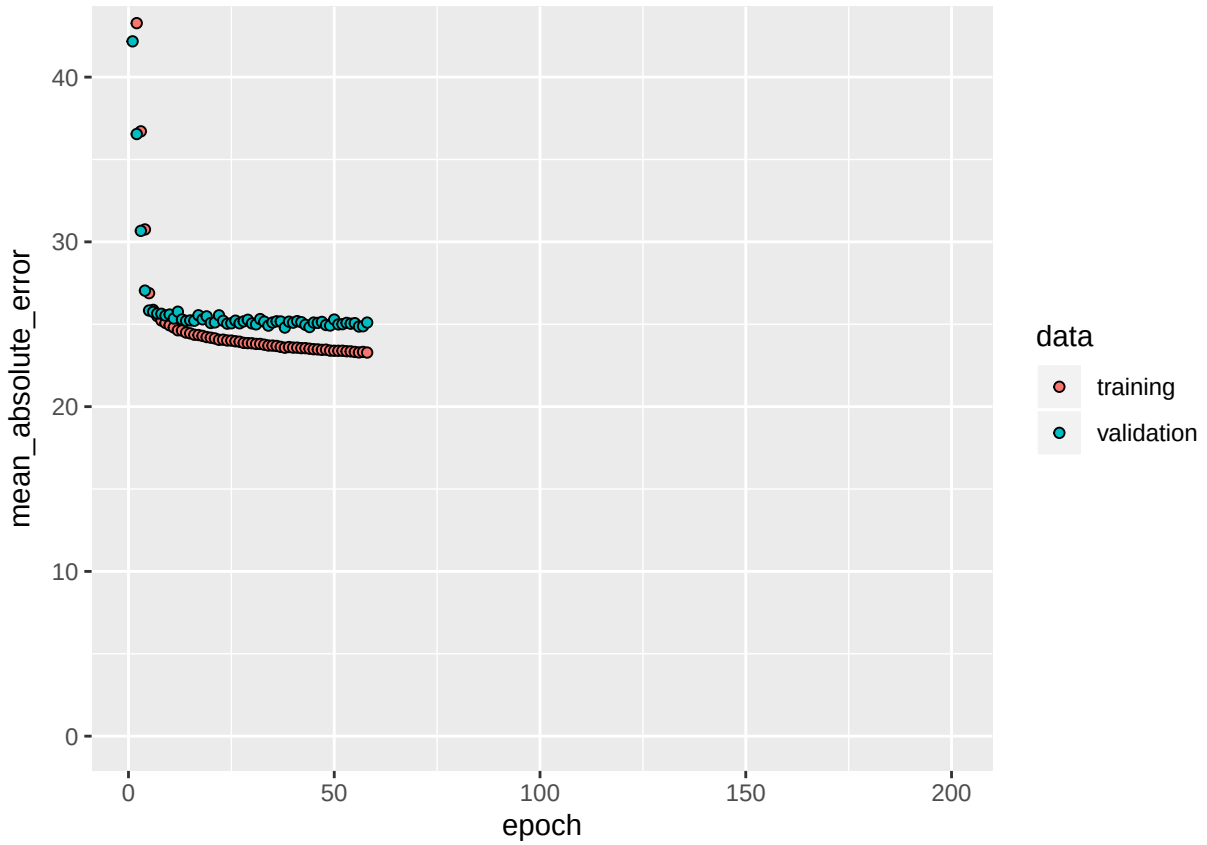
```r
set.seed(1)

train_labels <- as.matrix(train[,"Rank"])
train_data <- as.matrix(train[!names(train) %in% c("Rank","feature")])
test_data <- as.matrix(test[!names(train) %in% c("Rank","feature")])
test_labels <- as.matrix(test[,"Rank"])

train_data <- scale(train_data)
test_data <- scale(test_data)

model <- keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu",
              input_shape = dim(train_data)[2]) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 1)
```

```
model %>% compile(
  loss = "mse",
  optimizer = optimizer_rmsprop(clipnorm = 1),
  metrics = list("mean_absolute_error")
)

summary(model)
```

```
## Model: "sequential"
## _________________________________________________________________________
## Layer (type)                    Output Shape                  Param #
## =========================================================================
## dense (Dense)                   (None, 64)                    704
## _________________________________________________________________________
## dense_1 (Dense)                 (None, 64)                    4160
## _________________________________________________________________________
## dense_2 (Dense)                 (None, 1)                     65
## =========================================================================
## Total params: 4,929
## Trainable params: 4,929
## Non-trainable params: 0
## _________________________________________________________________________
```

```
epochs <- 200
early_stop <- callback_early_stopping(monitor = "val_loss", patience = 20)

history <- model %>% fit(
  train_data,
  train_labels,
  epochs = epochs,
  validation_split = 0.2,
  callbacks = list(early_stop)
)
```

```
plot(history, metrics = "mean_absolute_error", smooth = FALSE) +
  coord_cartesian(ylim = c(0, max(history$metrics$val_mean_absolute_error)))
```



The model seems to stop improving after the 87th epoch, which is probably not the best - though not entirely surprising given every other model attempted.

```
test_predictions <- model %>% predict(test_data)
mse = mean((test_labels - test_predictions)^2)
MSEs["Neural Net","MSE"] = mse
mse
```

```
## [1] 939.7132
```

This MSE is not great and is actually worse than some of the previous models. I wonder if there are improvements I could make to the Neural Net so as to bring the MSE down, but I'm assuming that, due to the nature of the dataset, the MSE won't go much further below 800 or so.

Comparing Models

```
MSEs
```

```
##                     MSE
## Bagging        874.0500
## Random Forest  868.7600
## Boosting      1197.8400
## xgboost        886.0400
## Neural Net     939.7132
```

From the above table, I can conclude that no model is that successful at predicting Rank. I used a classification model for the logistic, lasso, and ridge regressions, as well as the decision tree, but the abysmal performance of every model here makes it pointless to include them in the comparison. The table above does tell me that

XGBoost does the best at trying to predict Rank, so that is probably the most interesting observation I can make.

Conclusion

I greatly enjoyed this project - despite the unfortunate results of my dataset. In retrospect, I wish I can chosen a more economic-driven dataset as I think supervised learning that the class focuses on lends itself better to data that has apparent trends. Ultimately, the most important statement I can make on lyric and song data is that music popularity is more dependent on the people listening than the contents of the song. I think that - especially in modern music - there are trends that cause many of the columns I included to diverge to a common distribution, meaning that there are not many differences apparent between genres, which I'm guessing makes classification of genre very difficult.

At the end of the semester, I think that I am able to say that I walk away with a good understanding of what machine learning is and the many models and methods it entails. Though my project is rather defunct, I'm glad I was able to take the semester to learn a subject that I am very interested in. I look forward to learning more about machine learning - and maybe doing a few projects that produce real results.