A Comparison of Statistical Learning Models in Predicting Song Genre Using Lyrics

Completed for the Certificate in Scientific Computation
Spring 2023

Soomin Cho
Bachelor of Science in Mathematics Candidate
Mathematics Department
College of Natural Science

Joel Nibert
Assistant Professor of Instruction
Department of Mathematics

Abstract

The following research, A Comparison of Statistical Learning Models in Predicting Song Genre Using Lyrics, looked at different kernels of Support Vector Machines - linear, radial basis function, polynomial, and sigmoid. This was done through the scikit-learn library of Python in which multiclass statistical learning classifier models were trained on a set of songs extracted from the API of genius.com, a website that hosts contemporary song information and promotes collaboration of users in correcting, discussing, and interpreting song lyrics. Lyrics of each song were divided into words and their frequencies and both were used as the predictor variable. The genres in question were Rap, Pop, Country, R&B, and Rock.

This research explored data manipulation, big data memory limitations, and training models using word frequency. The confusion matrices and scores, often referred to as accuracy, of each multi-classification model are displayed in the Results section of this paper. The classifier that performed the best was the Support Vector Machine with a sigmoid kernel and the worst was the Support Vector Machine with a polynomial kernel. Rap and Rock songs were predicted correctly relatively well compared to the other three genres.

Introduction

Recent advancements in personalization, artificial intelligence, and data science have allowed software to simulate and predict nearly anything and everything. Music is a prime example of newly developing software transforming an industry. With AI generated songs [4], personalized DJs [17], and recommendation algorithms being improved every day, music has so much data to be explored. One critical component of music is genre - a method of categorizing mainstream media by content, rhythm, style, tempo, and much more.

For my research, I developed a statistical learning model to predict a song's genre using its lyrics. Observed genres in this research are Rap, Pop, Country, R&B, and Rock. These five main genres of contemporary music have distinct sounds and styles.

Are the words used to convey the artists' emotion, story, and/or culture distinct enough in usage and count to categorize a song into its proper genre? Are humans able to categorize music without sound and texture? Simple generalizations could be made with stereotypes, such as Country songs being about love or Pop songs making people dance; but, how distinct are the lyrics in comparison?

In statistical learning, a key component of modeling is supervised versus unsupervised learning. Supervised learning uses training samples with labeled response variables, whereas unsupervised learning does not have labels [7]. Thankfully, most songs today have a genre already associated with the song by the artist or fans. Thus, we will have labeled response variables for every observation and use supervised learning.

There are two types of supervised learning: regression and classification [7]. Regression is used when the data and response are quantitative. Classification is used when the response is qualitative - as in, the predicted response can be classified into categories. Since genres are categorical, I considered different models of classification, such as K-nearest neighbors (KNN), discriminant analysis, Naive Bayes classifier, decision trees, and more. However, the models selected would need a tremendous amount of flexibility and be equipped to handle multiple classes.

In order to categorize songs to its genre using lyrics with a statistical learning model, I planned to use support vector machines (SVM), an autonomous system of support vector classifiers (SVC), and compared the effects of changing kernels when training the model [7]. The support vector classifier is a type of classifier developed in the 1990s intended for binary classification using hyperplanes [7]. A "hyperplane in $\mathbb{R}^n$ is an $(n-1)$-dimensional affine subspace of $\mathbb{R}^n$" [15] which, in classifiers, separates the categories based on predictor variables. SVC was a method to soften the maximal margin classifier, which often over-fit due to its nature of maximizing the distance between training observations and the hyperplane and not permitting incorrect classification. Instead, the SVC allows support vectors, observations which may be on the decision boundary, on the wrong side of the margin, or the wrong side of the hyperplane all together. A support vector classifier is inherently a linear classifier with linear hyperplanes and a linear decision boundary. When we attempt to observe non-linear boundaries, we begin using different kernels, resulting in a variety of support vector machines.

Simply put, a "kernel is a function that quantifies the similarity of two observations" involving the inner products and/or the Euclidean distance of the observations [7]. "An inner product is a generalization of the dot product" which is denoted by $\langle x_i, x_{i'} \rangle$ with two vectors x_i and $x_{i'}$ in $\mathbb{R}^n$[6]. The Euclidean distance between two points in $\mathbb{R}^n$ space is defined as $||x_i, y_i|| = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2}$ [2]. Different kernels, and, thus, different functions of observations, of the support vector machines make for an interesting comparison. Table 1 describes the differences in some well-known kernels for SVMs.

Kernel	Function	Parameters	Notes
Linear	$K\langle x_i, x_{i'} \rangle$	i = 1, 2, …, *n*	where *n* = number of features
Polynomial	$K((\gamma\langle x_i, x_{i'} \rangle + r)^d)$	γ = scale, auto, or non-negative float d = degree of polynomial r = coef0 (default is 0)	scale (default): $\gamma = \frac{1}{n * var(X)}$ auto: $\gamma = \frac{1}{n}$
Radial Basis Function (RBF)	$K(exp(-\gamma\|\|x_i, x_{i'}\|\|^2))$	γ = scale, auto, or non-negative float || || = Euclidean distance	scale (default): $\gamma = \frac{1}{n * var(X)}$ auto: $\gamma = \frac{1}{n}$
Gaussian	$K(exp(-\frac{1}{2\sigma^2}\|\|x_i, x_{i'}\|\|^2))$	$\gamma = \frac{1}{2\sigma^2}$ || || = Euclidean distance	Example of RBF but uses Gaussian distribution where σ^2 is like variance
Sigmoid	$K(tanh(\gamma\langle x_i, x_{i'} \rangle + r))$	γ = scale, auto, or non-negative float r = coef0 (default is 0)	scale (default): $\gamma = \frac{1}{n * var(X)}$ auto: $\gamma = \frac{1}{n}$

Table 1: Different Kernels and Their Properties [5, 7, 18, 20]

However, with five genres being tested, we stretch the classifier to account for more than two classes. SVMs are able to accommodate more than two classes by either the One-vs-One approach, in which it creates combinations (in this case, $\binom{5}{2} = \frac{5!}{(5-2)!2!} = 10$ combinations) and compares each pair of classes, or the One-vs-All approach, in which the classifier compares one class to the remaining classes [7].

Ultimately, the statistical learning and modeling was performed via code using the Python library scikit-learn, often shortened to sklearn. Scikit-learn is a tool for machine learning and predictive data analysis, perfect for developing a statistical learning model. The library includes an SVM package that allows analysis using Linear SVC as well as SVMs with different kernels [20]. It is important to note that all sklearn SVC models for multi-classification problems have a parameter called *decision_function_shape* for SVC and *multi_class* for LinearSVC, in which you can choose the One-vs-One or One-vs-All approach. The default decision function shape "ovr", which stands for "one vs. rest", was used throughout the project. Below, Table 2 shows the result of using different kernels from sklearn's documentation on SVMs.

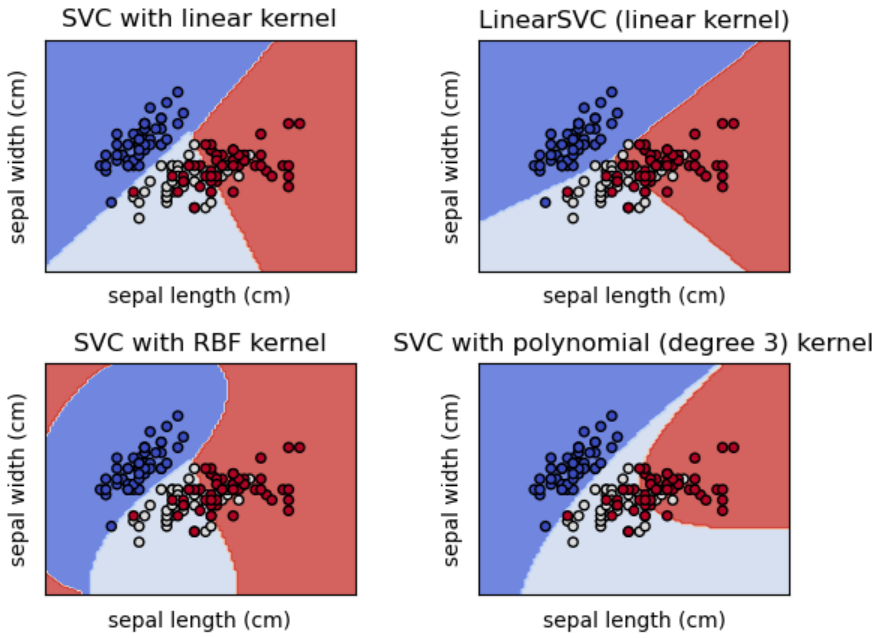


Table 2: Graphs of the effects of different kernels from sklearn's page on SVM [20]

Similar research has already been performed that used a Multinomial Naive Bayes model as well as the Linear SVC model using sklearn on Python [11]. These two functions are some of the simplest multiclass classifiers in sklearn. However, I will explore how different kernels of the support vector machines - radial basis function, polynomial, and sigmoid - possibly produce different predictions and results.

Materials and Methods

Materials

The majority of the research project was done through coding. Thus, I mainly used Visual Studio Code (VSCode) as my text editor. Additionally, I used a Jupyter Notebook to organize various files and connected it to VSCode so that cells could be run separately and repeatedly without affecting the rest of the code and data. To run a Jupyter Notebook on VSCode, I connected to the Python 3.8.0 kernel on my laptop's Python environment.

Later on in the project, I had to utilize a much more powerful machine. I discovered that the Mathematics Department's computing resource page was outdated with many pages directing me to https://www.ma.utexas.edu/404, indicating a 404 error where pages cannot be found. I tried to create an account with CS Labs, the Computer Science Department's machines, as I have done before, but my request was denied since I am no longer enrolled in a CS course. As a last resort, I contacted a friend who is an Electrical and Computer Engineering student to help run my Python files on the Electrical and Computer Engineering department's Linux Application Servers (ECE-LRC) using his account. Although under normal circumstances this is not allowed, given the time constraint and the massive amount of memory I needed as soon as possible, this was the best solution. Details of the server and connection are discussed further in Discussion and Acknowledgements.

Finding the Perfect Dataset

Much of the previous research I read about lyrics to genre classification used the musiXmatch dataset from Million Song Dataset [19] and the genre annotations from tagtraum industries [12]. However, upon downloading and examining the musiXmatch dataset, I deemed it challenging to manipulate due to its bag-of-words and stemming format as well as its difficult interpretability. This began an exploration of lyrics and genre datasets.

One of the first datasets I found was a dataset from Kaggle called "Song lyrics from 79 musical genres" [8], which was made by data scraping vagalume.com.br. The dataset included two CSVs, artists-data.csv and lyrics-data.csv, which was cleaned via the code from Appendix A & B. The cleaning process used pandas and regex to discard whitespace, symbols other than apostrophes, and lowered any capitalized letters for consistency in lyrics. The code then counted word frequency, unique word count, and total word count per song. The word frequency was stored as a dictionary. Finally, the code took the genres section of the original CSV from Kaggle and split on ';' and stored genres as a list. The resulting data frame was a set of English songs with columns *title, artists, lyrics, word freq, unique word count, total word count*, and *genres* which was exported to a CSV, *cleaned_kaggle.csv*, to be used in other Python files.

However, I wanted to look at other datasets since I saw that more similar research has been performed using this dataset. An idea I explored was to use the Spotify API to accumulate a list of songs per genre using a playlist. Sadly, this exploration was fruitless. Given the time constraints, making a playlist per genre was neither feasible nor within good statistical practice since the data would have been selected by me and the recommendations of my Spotify account.

However, playing with APIs gave me an idea to explore genius.com, “the world's biggest collection of song lyrics and musical knowledge” [16].

As seen in Appendix C, I was able to web-scrape using Genius’s API and extract song title, artists’ names, lyrics, and genres, labeled Tags, from each song webpage’s HTML using the BeautifulSoup Python library [10]. The code iterated through IDs set by genius.com and used multiprocessing to parallelize the process since it took a very long time per song. Multiprocessing and Pooling allows code to be run on multiple cores of the computer’s CPU [1], and, although accumulating enough data still took multiple days, was a much faster process. Comments in Appendix C will notate that some IDs were skipped. This was a decision made due to the fact that genius.com was originally used to gather and interpret Rap song lyrics, thus, many of the beginning songs were Rap songs. Skipping IDs allowed the dataset to reach songs of genres other than Rap quickly and without taking up unnecessary space in the dataset. Although *i* and *stop*, in thousands, are written as 120 and 125, respectively, the code was run multiple times with multiple values for *i* and *stop* to cover all the necessary IDs. Code in Appendix C follows the code in Appendix B with cleaning lyrics and storing genres as a list. The resulting pandas dataframe was saved as a CSV named *final.csv*.

It’s worth mentioning that there are some inaccuracies when accumulating song data. Genius.com is a collaborative space for users to write, edit, and discuss song lyrics. Thus, naturally, human mistakes will be made and transferred into my dataset. Additionally, although commented out in the final code, in the beginning iterations of the program, the first line was excluded when storing lyrics since the first couple of songs’ first line was the song title. However, after generating thousands of songs and checking the *lyrics* column of the dataframe, I discovered the inaccuracy much later and it was too late to regenerate data and much too difficult to differentiate between if the first line is the song title or part of the lyrics. Since the first line is usually a small percentage in the overall count of words, I considered this inaccuracy negligible given the time constraints.

Analyzing Datasets

After cleaning both Kaggle and Genius data, the final datasets were 191,814 and 331,315 songs, respectively. For the sake of analysis, we must generate some graphs and visual aids to help us comprehend such massive amounts of data. Within the Kaggle dataset, there were 258,178 unique words. 534 of them had word frequency of over 10,000 in the entire dataset; 139 of them had word frequency of over 50,000. To visualize word frequencies, Graphs A, B, C, and D were generated and are shown in Appendix I. Graphs A & B show the word frequency of some of the most frequent words for the Kaggle dataset. There is quite a dramatic decay in word frequency and the range is enormous to the point that the graphs’ fonts are much too small to read. Hence, there is a sample table of the words in the graphs.

Similar graphs were generated for the Genius data. However, with more than 100,000 songs than Kaggle, the Genius dataset had many more words and a much higher frequency. Graph C shows Genius words that have frequencies between 10,000 and 100,000. Graph D

shows Genius words with frequencies over 100,000. Although Graph D shows rapidly decaying patterns like Graphs A & B, Graph C does not have such a drastic drop off since it's a section of the overall decay picture. Interestingly, the top 5 words for the Kaggle and Genius datasets are the same, although not in the same order: the, you, i, to, and.

Another analysis was done via word clouds for each dataset's genres as seen in Appendix J. If we are expecting a model to distinguish between genres with a song's lyrics, what do those words look like? Images A & B show word clouds per genre per dataset. You can see in both Kaggle and Genius datasets that the word "the" is most frequent in Rap and Country and "you" is frequent in R&B songs from both datasets. Additionally, for both datasets, Rock did not have as many words that were significantly larger than the others. Despite being different datasets, these similar patterns in word count indicate that there are distinguishing differences between genres.

Developing Model

After analyzing the datasets, the next step was to develop the model. To use sklearn's functions, the ultimate goal was to have a matrix in the form of a pivot table or data frame to be used as the predictor variable in which the columns are words and the rows are each song's index. The data in the matrix is the word frequency for the particular song of that row. Since the data is supervised and a set of response variables are present, there will also be another matrix in the form of a pivot table or data frame that will contain song indices matched with the song's genre.

However, many decisions had to be made along the way. The first was to truncate the list of genres to the first genre and then to eliminate any songs if the specified genre was not one of Rap, Pop, Country, R&B, or Rock. This will allow the response variable matrix to contain one genre and train the model on one specific genre per song. This process shrunk the datasets such that the Kaggle data ended up being 61,211 songs, almost a third of the original data, and genius ended up being 331,313 songs, which was only 2 songs less than the original. Later on, the genres Rap, Pop, Country, R&B, and Rock were mapped to corresponding integer values 0, 1, 2, 3, and 4 using a function and *apply()* as seen in Appendix H.

Genre	Integer
Rap	0
Pop	1
Country	2
R&B	3
Rock	4

Table 3: Genre to Integer

The second decision was to not include very rare words when listing words for the columns of the predictor variable. In the beginning, only words with a frequency of more than 10 in the entire dataframe were counted. This narrowed kaggle's 258,178 unique words down to 43,942 words and genius's 604,531 unique words narrowed down to 82,952 words. However, this means that the predictor variables for Kaggle will be a matrix of 61,211 by 42,942 and Genius will be a matrix of 82,952 by 331,313. These are extremely large dataframes. Consequently, I spent the next few weeks dealing with various computer memory issues with pandas dataframes, elaborated further in the Discussion section.

Due to the aforementioned memory problem, this began another few weeks of data cutting, shrinking, and cleaning. The third decision was to stop using the Kaggle dataset all together. This decision was due to the fact that, upon exploring the contents of lyrics from the Kaggle data when cleaning, it was discovered that some lyrics had blanks represented by a series of underscores. Examples are shown in Image C of Appendix J to justify the elimination. Hence, the appendices after B will only show code that dealt with the Genius dataset.

The fourth decision was to discard any songs whose word count was over 3,500 and to also go through the Genius dataset and remove any songs that had a word whose frequency was greater than 255. When populating the genius dataset, some snippets of audio books that were part of genius.com seeped through. Although receiving audio books or text that weren't songs was minimized by returning out of the *get_df()* function if the genre tags included 'Non-Music' as seen in Appendix C, this was not enough to catch any non-song text if it was tagged with another genre without 'Non-Music' (many of these chunks of text were tagged with Rap). The latter process was performed due to the fact that when storing numbering in a pandas dataframe, data types play a large role in how much memory the data frame uses. The amount of memory of a dataframe with data type 'uint32', which can store up to 2^{32} bits, is approximately 4 times as large compared to the same dataframe with data type 'uint8', which can store up to 2^{8} bits. Keeping word frequency below 255 allows dataframes to be stored with the 'uint8' data type, in which 'u' indicates using the positive number, 0-255. After both cleaning processes, the Genius dataset had 331,204 songs.

After many failed attempts to make the predictor variable matrix, in which, after every failed attempt, I incrementally increased the word frequency threshold up to 250, I made the final decision to reduce the Genius dataset. In the end, the Genius dataset had to be reduced twice because the dataframe after removing 50% of Rap and Pop songs was still too large and crashed the Jupyter Notebook kernel. I settled on removing 75% of Rap and Pop songs, resulting in 136,099 songs, which was saved as *genius_cut_again.csv* as seen in Appendix G. Table 4 details the number of songs for each genre during this process.

Original = 331,204 songs		**First cut = 201,134**		**Second cut = 136,099**	
Rap	130,762	**Rap**	65,381	**Rap**	32,690
Pop	129,377	**Pop**	64,688	**Pop**	32,344
Rock	50,434	**Rock**	50,434	**Rock**	50,434
R&B	10,708	**R&B**	10,708	**R&B**	10,708
Country	9,923	**Country**	9,923	**Country**	9,923

Table 4: Genius Data Removal Counts Per Genre

Out of the 136,099 songs, 6,635 words had a frequency over 250 in the entire data frame. The predictor variable matrix will now be 136,099 rows and 6,635 columns which is 903,016,865 elements. Although this was small enough to not produce a performance warning, which happened before the reducing measures, it was still too large and crashed the kernel on my personal device. However, when the code was tested on the ECE-LRC machine, it successfully ran and produced the predictor variable matrix. From this moment onwards, I changed to only running on the ECE-LRC machines and converted my Jupyter Notebooks to Python files. The resulting python file is shown in Appendix H which includes reading the Genius data CSV, creating *gte_lst* which is the list of words with frequency greater than or equal to 250, and creating *x* and *y*, the predictor variable matrix and the response variable matrix, respectively. Using sklearn's model selection package and its *train_test_split()* function, the code was able to take *x* and *y* and allocate 20% of the dataset to be the test features and labels. The other 80% was used to train the model.

After multiple attempts to run the program, which took days at a time, I began to research how to minimize runtime. A solution I found was to scale the input data using *sklearn.preprocessing.StandardScaler()* [13]. After scaling the data, I ran the program one last time on all variety of the kernels with a random state set to 0 in order to control any random number generation when shuffling data:

LinearSVC

The support vector classifier draws a linear hyperplane in the (*n-1*) dimensional space as described in the Introduction. In this case, with 6,635 words, the Linear SVC made a linear hyperplane of the 6,634th dimension. Let p = 6,635. Then equation of the hyperplane can be written as

$$f(X) = f(X_1, X_2, \ldots, X_p) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p$$

where β_i was calculated using the training data [7]. Since we used to one-vs-rest default decision function shape, the model detects if the point $X_i = (X_1, X_2, \ldots, X_p)$ per observation/song *i* is on the correct or incorrect side the hyperplane $f(X)$.

SVC(kernel='rbf)

When the kernel is set to the radial basis function (RBF), data is mapped to a higher-dimension [14] using the function *K* as defined for the RBF kernel in Table 1. This allows a linear hyperplane to be fit to the newly transformed data. However, when reverting back to the correct dimension, the hyperplanes will look much more flexible than the linear SVC model, seen in Table 4. A correct gamma is crucial to the accuracy of the model; alas, with little time, the default gamma of 'scale' was used.

SVC(kernel='poly')

The polynomial kernel behaves similarly to the RBF function in which data is mapped to a higher-dimension by a polynomial function and then a hyperplane is constructed. When the hyperplane is shown in the original dimension, the shape of it will look like a polynomial function with degree *d*, as seen in Table 4.

SVC(kernel='sigmoid')

The sigmoid kernel behaves similarly to the RBF function in which data is mapped to a higher-dimension by a sigmoid function and then a hyperplane is constructed. When the hyperplane is shown in the original dimension, the shape of it will look like a sigmoid function, as depicted in Table 4.

Multinomial Naive Bayes

The distribution of the multinomial data is represented by vectors $\theta_j = (\theta_{j1}, \theta_{j2}, \dots, \theta_{ji})$ where *j* = number of genres and *i* = number of words. Each $\theta_{ji} = P(x_n|j)$ of the *n*th word belonging to genre *j*. Each θ_j is calculated using Laplace smoothing ($\alpha = 1$, the default) and maximum likelihood function in which , $\theta_{jn} = \frac{N_{jn}+\alpha}{N_j+\alpha i}$, where N_{jn} = number of times *i*th word appears in genre *j* in the training data and N_j = number of words in genre *j*. The model then uses test data words to predict the genre to be the class in which the posterior probability is the highest. [21]

Results

With supervised data, we can map correct and incorrect predictions using a confusion matrix which highlights how the model performed when predicting genres using test data. The following images are the confusion matrices of different models trained on 80% of the genius data set and then tested on the remaining 20%. The confusion matrices are shown in counts out of 27,220 songs, which is 20% of 136,099 - the total number of songs used in the final creation of the predictor and response variables. The score, also known as accuracy, was calculated using $score=\frac{TruePositives}{TotalCount}$, in which True Positives are along the main diagonal, the top-left-to-bottom-right line, where the model correctly predicted the genre. The specific outputs from the code are detailed in Appendix H as well.

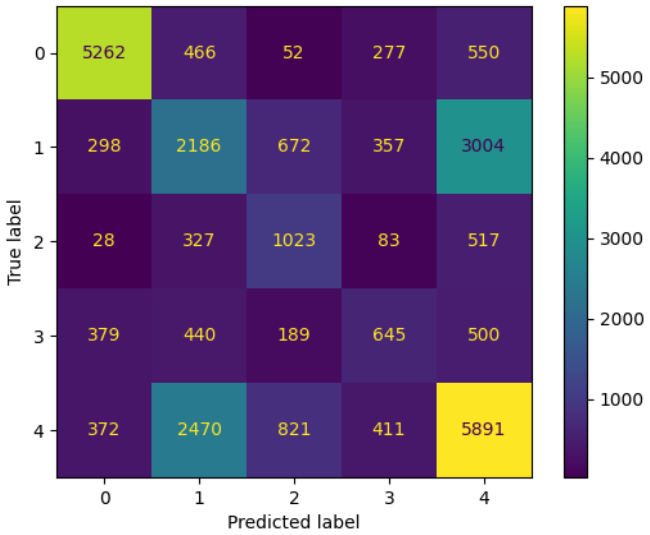
It must be noted that, in order to produce results in the given time frame, I had to run the models on different servers. Linear SVC and SVM with RBF kernel were run on one machine while SVM with Polynomial and Sigmoid kernels and the Multinomial Naive Bayes model were run on a different machine. Thus, when we run the line *train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2)* with no random state specified, the train data and test data selections will be different due to the random process the function uses. This was an oversight I did not foresee until after the code was run and did not have sufficient time to rerun the code since each model takes days to train and predict. However, the random sampling resulted in similar distribution of genres between the two machines. A table of each model's specific counts and percentages are given next to each confusion matrix.

I trained a Linear SVC and a Multinomial Naive Bayes model, similar to Rutter's research [11]. Out of the Support Vector Machines, Linear SVC ran the fastest. This may have been due to the fact that, since default values of the models were used except random state, the default *max_iter* value in Linear SVC is 1000. A Convergence Warning appears in the output of Appendix H as a result of the default limiting iterations before the model was fully fitted with the

training data. Since we are calculating probability and not looking at hyperplanes, the Multinomial Naive Bayes model ran in less than a minute while the others took a day or more.

Linear SVC

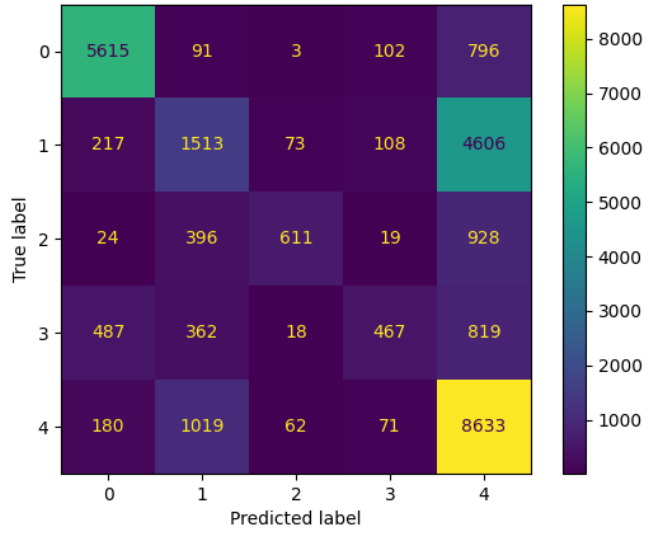
Score: 0.5513



Linear SVC				
Label	**Genre**	**Count**	**Correct**	**Correct%**
0	Rap	6607	5262	79.64%
1	Pop	6517	2186	33.54%
2	Country	1978	1023	51.72%
3	R&B	2153	645	29.96%
4	Rock	9965	5891	59.12%

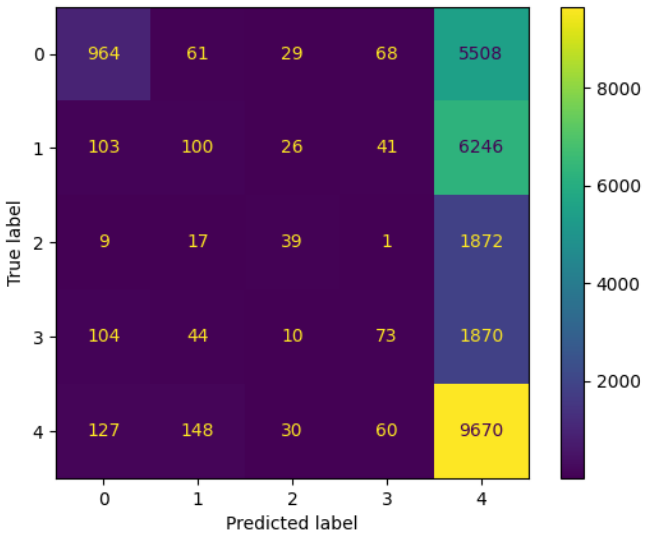
SVM with Radial Basis Function kernel

Score: 0.6186



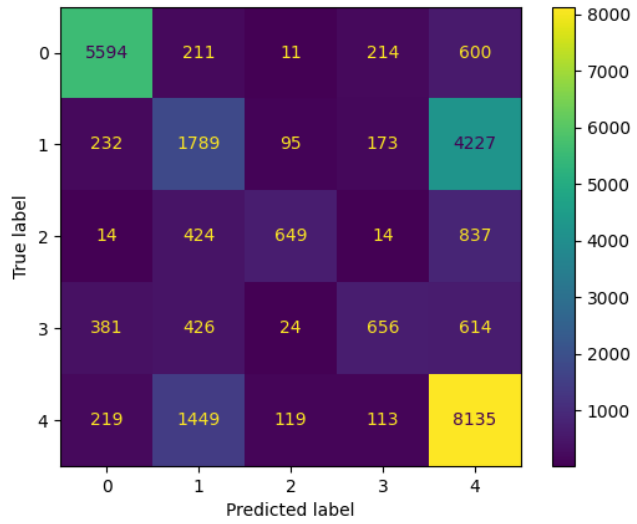
SVM with Radial Basis Function kernel				
Label	**Genre**	**Count**	**Correct**	**Correct%**
0	Rap	6607	5615	84.99%
1	Pop	6517	1513	23.22%
2	Country	1978	611	30.89%
3	R&B	2153	467	21.69%
4	Rock	9965	8633	86.63%

SVM with Polynomial kernel
Score: 0.3985



SVM with Polynomial kernel				
Label	**Genre**	**Count**	**Correct**	**Correct%**
0	Rap	6630	964	14.54%
1	Pop	6516	100	1.53%
2	Country	1938	39	2.01%
3	R&B	2101	73	3.47%
4	Rock	10035	9670	96.36%

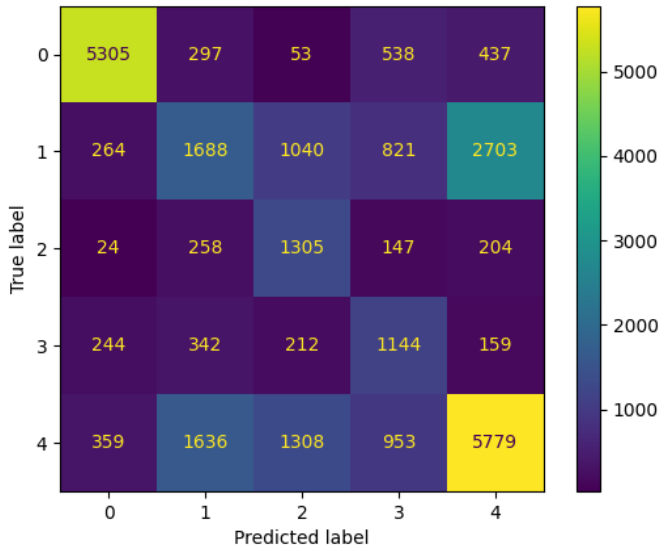
SVM with Sigmoid kernel
Score: 0.6180



SVM with Sigmoid kernel				
Label	**Genre**	**Count**	**Correct**	**Correct%**
0	Rap	6630	5594	84.37%
1	Pop	6516	1789	27.46%
2	Country	1938	649	33.49%
3	R&B	2101	656	31.22%
4	Rock	10035	8135	81.07%

Multinomial Naive Bayes

Score: 0.5592



Multinomial Naïve Bayes				
Label	**Genre**	**Count**	**Correct**	**Correct%**
0	Rap	6630	5305	80.02%
1	Pop	6516	1688	25.91%
2	Country	1938	1305	67.34%
3	R&B	2101	1144	54.45%
4	Rock	10035	5779	57.59%

The polynomial kernel had the worst outcome of all the models. The runtime was also substantially greater than all the other models, taking three days. I do not think the default degree of 3 was the right choice for such a complex multivariate problem, nor was a polynomial function as flexible as the radial basis function or sigmoid function; thus, the predictions were greatly misclassified. With more time, I would have loved to explore how different degrees of the polynomial affect the results.

In all the other models, Rock and Pop are often misclassified against each other. More specifically, although Rock had a high percentage of correctness, Pop was misclassified to Rock more often than correctly classified. This may be due to the fact that Pop and Rock are genres that cater to the audience compared to the more narrative, story-like nature of Rap, R&B, and Country. This similarity may confuse models that look at different usage of words, such as our Support Vector Machines.

Furthermore, R&B had quite low correct classifications across all models. Between Country and R&B, two genres that had a smaller set of data than the others, R&B performed worse than Country songs every time, especially in LinearSVC. One possibility is that there was not sufficient data for these genres and, thus, the models were not fitted with enough data. Another possibility is that R&B is not as distinguishable in wording choices as the other genres due to its smooth, storytelling, narrative format.

On the contrary, Rock, the most abundant genre, had the most songs correctly classified. Although the number of Rock songs in the test data was very large compared to the other genres, when looking at the percentages of correct classification, Rock and Rap had similar relatively high true positive predictions with non-linear kernels. Given more data and time, I foresee the models becoming more accurate, especially between very different genres such as Rock and Rap.

Discussion

At the beginning of the semester, I was discouraged from proceeding with my research project. Similar research has been done previously. Specifically, Rutter looks at using lyrics to be categorized into four genres - Rock, Pop, Hip-Hop, and Country - and compares the results of using a Multinomial Naive Bayes model and a Linear Support Vector Classifier model [11]. The datasets used were the musiXmatch dataset from Million Song Dataset [19], the same dataset I was originally planning on using in my proposal, and the genre annotations from tagtraum industries [12]. However, I was encouraged to find small differences to improve and/or add to already existing research. Thus, I settled on doing similar research using similar methods, but on a different dataset and with different kernels of SVMs.

However, I underestimated the memory and computing power needed to run non-linear SVM models. Unfortunately, this problem took up most of my time. During the course of the project, the kernel for the Jupyter Notebook on my laptop failed several times because the computer ran out of memory, which forced the entire notebook to be restarted. Repeating this process over and over again, hoping that small changes would help, took weeks.

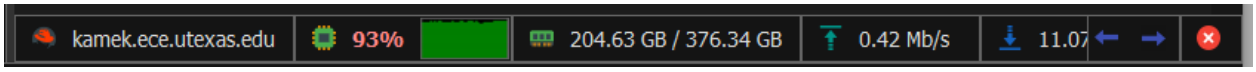
Thankfully, I was able to run my code on ECE Linux Application Servers through a friend's account. Although this is not within their terms of using the machines for only Electrical Engineering purposes, as mentioned before, this was the last resort. The friend, who wishes to remain anonymous, and I were able to share a Google Drive folder in which I uploaded the necessary CSV and Python files and they were downloaded and immediately ran. Any and all changes to the code were written by me when errors arose or I had to change the specifications because the code was running for too long. When the confusion matrices were generated, the PNGs were uploaded to the Google Drive folder as well so I could retrieve them.

The ECE-LRC servers have extremely powerful computing power. The details of each server (there are nine servers total, but I limited usage to two) are outlined in Table 3.

CPU	CPUs	Cores/CPU	RAM (GB)	OS	Architecture
Intel Xeon 5218 2.3GHz, 32T	2	16	384	CentOS 7	64-bit

Table 5: ECE-LRC specifications [3]

However, even on these powerful machines, my code was stretching them to its limits. When testing how to make the predictor variable matrix with over a billion elements, the RAM indicator increased to over 200GB. When training nonlinear support vector machines, the computing power was over 90%, almost using the server's entire computing power. An example of the servers moderating panel is shown below:



Fortunately, I was able to use other methods, such as scaling, to bring down necessary computing power. However, the runtime is still extremely long for such a large dataset.

If allotted more time and computing power, I would love to have explored custom kernels as well as . Using different functions besides provided kernels was my ultimate goal at the inception of this research project. Nevertheless, I was admittedly underprepared to use scikit-learn's powerful library and underestimated the complexity of the models/code. Before jumping right into analysis, I should have researched more about the Python library and all that is necessary to use it in addition to the mathematical readings and discoveries. However, this semester-long research allowed me to experience a plethora of real-life situations and difficulties of the growing industry of data science as well as learn various skills to be used through my future endeavors.

Acknowledgements

First and foremost, I would like to thank my faculty supervisor, Dr. Joel Nibert, for his tremendous help throughout the course of this project. From the brainstorming sessions of last semester, providing guidance in statistical learning, all the way up to his persistent encouragement for me to find solutions when I encountered a problem exemplifies how integral Dr. Nibert has been to the success of this research project. With this being my first big research project and paper, I had various questions in decision making and writing. Dr. Nibert always promoted discussion and allowed me to reach conclusions myself, while still pointing me in the right direction. Through our weekly virtual meetings, I was able to provide updates on the status of my project and Dr. Nibert continuously aided in further analysis and sprouting new considerations to be made.

I would also like to extend my gratitude to Sarah Mendoza, a fellow student also under the guidance of Dr. Nibert for her research project for the certificate in scientific computing and data science. Sarah was an essential part of being able to discuss any questions or concerns, to bounce ideas off of, and, since she was always on top of her project, a source of inspiration to continue my research. If it wasn't for Sarah's timely work, I would not have remembered the benefits of Jupyter Notebooks, have a detailed calendar, nor have been as motivated to stay on track.

Additionally, I would like to thank the Electrical and Computer Engineering student, who wishes to remain anonymous. However, without their help and their access to the ECE-LRC computers, this project would have been barren and incomplete. I truly hope there will be no consequences to a student who helped out a fellow student struggling due to computing limitations. Without this student's constant availability and openness to help, the extensive runtime of my program would be even greater and possibly infinite. This gratitude also extends to the Electrical and Computer Engineering Department for being readily prepared for such intensive computing projects.

I would also like to acknowledge genius.com and its API as well as kaggle.com, despite not being used in the final analysis. These datasets allowed my research to be unique and hold on

its own compared to past research. Lastly, I would like to acknowledge the various Python libraries that were used in this project along with their documentation. Countless hours were spent on the documentation pages of pandas [9], BeautifulSoup [10], and sklearn. Additionally, I would like to acknowledge the hundreds of StackOverflow pages I deciphered through to find various solutions to errors and/or advice on coding. Without the ability to lookup warnings, errors, and different solutions, I would have been unable to proceed with my code several times throughout the semester.

References

1. Bonannella, L. A Guide to Python Multiprocessing and Parallel Programming https://www.sitepoint.com/python-multiprocessing-parallel-programming/#:~:text=One%20way%20to%20achieve%20parallelism,multiprocessing%20accomplishes%20process%2Dbased%20parallelism (accessed Feb 6, 2023).
2. Distance https://mathworld.wolfram.com/Distance.html (accessed Apr 13, 2023).
3. ECE Linux Application Servers https://wikis.utexas.edu/display/eceit/ECE+Linux+Application+Servers (accessed Mar 27, 2023).
4. Hochberg, B. Spotify wants to hook users on AI Music Creation Tools https://www.forbes.com/sites/williamhochberg/2022/06/29/spotify-is-developing-ai-tools-to-hook-users-on-music-creation/?sh=1d35136d4834 (accessed Apr 13, 2023).
5. Hofmann, M. Support Vector Machines — Kernels and the Kernel Trick. *Hauptseminar "Reading Club: Support Vector Machines."* https://cogsys.uni-bamberg.de/teaching/ss06/hs_svm/slides/SVM_Seminarbericht_Hofmann.pdf (accessed Apr 6, 2023).
6. Inner Product https://mathworld.wolfram.com/InnerProduct.html (accessed Apr 13, 2023).
7. James, G.; Witten, D.; Hastie, T.; Tibshirani, R. *An Introduction to Statistical Learning: with Applications in R*; 2nd ed.; Springer: New York, 2021.
8. Neisse, A. Song lyrics from 79 musical genres. Kaggle https://www.kaggle.com/datasets/neisse/scrapped-lyrics-from-6-genres?resource=download&select=artists-data.csv and select=lyrics-data.csv (accessed Jan 25, 2023).
9. pandas documentation https://pandas.pydata.org/docs/.
10. Richardson, L. Beautiful Soup Documentation¶ https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (accessed Feb 1, 2023).
11. Rutter, B. Naive Bayes, Song Lyrics and Genre: I built a naive Bayes model to predict genre from song lyrics and it went OK-ish https://towardsdatascience.com/i-built-a-naive-bayes-model-to-predict-genre-from-song-lyrics-and-it-went-ok-ish-639af0b0a078 (accessed Jan 30, 2023).
12. Schreiber, H. Improving Genre Annotations for the Million Song Dataset. In Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR). 2015, 241-247 https://www.tagtraum.com/msd_genre_datasets.html (accessed Jan 24, 2023).
13. Sklearn.preprocessing.StandardScaler https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html#sklearn.preprocessing.StandardScaler (accessed Apr 10, 2023).
14. Sidharth. The RBF kernel in SVM: A complete guide https://www.pycodemates.com/2022/10/the-rbf-kernel-in-svm-complete-guide.html (accessed Apr 11, 2023).

15. Sloughter, D. 1.4: Lines, Planes, and Hyperplanes. In *The Calculus of Functions of Several Variables*; Furman University; LibreTexts, 2021. https://math.libretexts.org/Bookshelves/Calculus/The_Calculus_of_Functions_of_Several_Variables_(Sloughter) (accessed Apr 10, 2023).
16. Song Lyrics & Knowledge https://genius.com/ (accessed Jan 27, 2023).
17. Spotify debuts a new AI DJ, right in your pocket https://newsroom.spotify.com/2023-02-22/spotify-debuts-a-new-ai-dj-right-in-your-pocket/ (accessed Apr 13, 2023).
18. Sreenivasa, S. Radial Basis Function (RBF) Kernel: The Go-To Kernel https://towardsdatascience.com/radial-basis-function-rbf-kernel-the-go-to-kernel-acf0d22c798a (accessed Apr 10, 2023).
19. The musiXmatch Dataset | Million Song Dataset http://millionsongdataset.com/musixmatch/ (accessed 2022 Dec 3).
20. 1.4. Support Vector Machines https://scikit-learn.org/stable/modules/svm.html (accessed Apr 13, 2023).
21. 1.9. Naive Bayes https://scikit-learn.org/stable/modules/naive_bayes.html#multinomial-naive-bayes (accessed 2023 Mar 20).

Appendix A - *Kaggle_combine.py*

After downloading CSVs from kaggle.com, load and combine artists and lyrics on artists' link and eliminate any non-English songs

```
import pandas as pd
kg_lyrics = pd.read_csv("lyrics-data.csv")
kg_artists = pd.read_csv("artists-data.csv")
kg_lyrics_en = kg_lyrics[kg_lyrics['language'] == 'en']
kg_lyrics_en = kg_lyrics_en.drop(columns=['SLink', 'language'], axis=1)
kg_artists = kg_artists.drop(columns=['Songs', 'Popularity'])

combined = kg_lyrics_en.merge(kg_artists, left_on='ALink', right_on='Link', how='left')
combined = combined.drop(columns=['ALink', 'Link'], axis=1)
combined.to_csv('kaggle_combined.csv')
```

Appendix B - *Kaggle_clean.py*

Clean whitespace, symbols, and lower any and all capitalized letters

```
import pandas as pd
import re
import collections

df = pd.read_csv('kaggle_combined.csv', index_col=0)
columns = ['title', 'artists', 'lyrics', 'word freq', 'unique word count', 'total word count', 'genres']
cleaned = pd.DataFrame(columns=columns)

for index, row in df.iterrows():
   cleaned.loc[index, 'title'] = row['SName']
   cleaned.loc[index, 'artists'] = row['Artist']

   lyrics = row['Lyric']
   lyrics = " ".join(lyrics.split("\n")[:])
   symbols_removed = re.sub(r'((\[.*?\])*)|([^A-z\s\'])', '', lyrics)
   whitespace_removed = re.sub(r'[ ]{2,}', ' ', symbols_removed)
   final_lyrics= whitespace_removed.lower()

   cleaned.loc[index, 'lyrics'] = final_lyrics

   genres = str(row['Genres']).split(";")
   cleaned.loc[index, 'genres'] = genres


   word_lst = final_lyrics.split(" ")
   word_freq = dict(collections.Counter(word_lst))

   # print(word_freq)

   cleaned.loc[index, 'word freq'] = str(word_freq)
   cleaned.loc[index, 'unique word count'] = len(word_freq)
   cleaned.loc[index, 'total word count'] = len(word_lst)

cleaned.to_csv("cleaned_kaggle.csv")
```

Appendix C - *Genius_Parallelized.py*

Web-scrape songs from genius.com and clean whitespace, symbols, and lowercase letters

```
import requests
import pandas as pd
from bs4 import BeautifulSoup
import re
import multiprocessing
import collections
import time

# make api_df
columns = ['title', 'artists', 'lyrics', 'word freq', 'unique word count', 'total word count', 'genres']
api_df = pd.DataFrame(columns=columns)


access_token = "f7upmRHup3nrB3HlLc4qh7jeP8dK3dSeW2m-mWbzPrQjWvE79xag0N4_M4RQ4Msl"

token = 'Bearer {}'.format(access_token)
headers = {'Authorization': token}

def get_df(id):
   try:
       r = requests.get('https://api.genius.com/songs/' + str(id), headers=headers)

       if r.status_code != 200:
           return

       song = r.json()["response"]["song"]

       if song["language"] != 'en':
           return

       if song["lyrics_state"] != "complete":
           return

       url = song["url"]

       url_request = requests.get(url)

       bs = BeautifulSoup(url_request.text.replace('<br/>', '\n'), "html.parser")
       div = bs.find("div", class_=re.compile("^lyrics$|Lyrics__Root"))
       tags = bs.find_all("a", class_=re.compile("SongTags__Tag"))

       if (div is None) or (len(tags) == 0):
           return

       genres = [tag.get_text() for tag in tags]
       if 'Non-Music' in genres:
           return

       # cleaning lyrics
       # get rid of lines with "Verse", "Chorus", band member name, etc
       # erase any special characters except ' and replace new lines and/or multiple spaces with one space
       # use all lowercase letters
       lyrics = div.get_text()
       lyrics = " ".join(lyrics.split("\n")[:])
       # lyrics = " ".join(lyrics.split("\n")[1:])
       lyrics = lyrics[:-5]
       lyrics = re.sub(r'((\[.*?\])*)|([^A-z\s\'])', '', lyrics)
       lyrics = re.sub(r'[ ]{2,}', ' ', lyrics)
       lyrics = lyrics.lower()

       word_lst = lyrics.split(" ")
       word_freq = dict(collections.Counter(word_lst))

       api_df.loc[id] = song["title"], song["artist_names"], lyrics, word_freq, len(word_freq), len(word_lst), genres
   except:
       print("Song information for ID: " + str(id) + " not found\n")
   return api_df


if __name__ == '__main__':
   # skip 200001 - 400000
   # skip 600001 - 800000
   # skip 1000001 - 1200000?
   i = 120
   stop = 125
   pool = multiprocessing.Pool()
   while i < stop:
       results = pool.map(get_df, range((i * 1000) + 1,  ((i + 1) * 1000) + 1))
       combined = pd.concat(results)
       final = combined.drop_duplicates(subset=['title', 'artists', 'lyrics', 'unique word count', 'total word count'])
       final.to_csv('final.csv', mode='a', index=False, header=False)
       print("Done. Added IDs " + str((i * 1000) + 1) + " to " + str((i + 1) * 1000) + " to CSV")
       time.sleep(200)
       i += 1
   pool.close()
```

Results in *final.csv* with columns title, artists, lyrics, word freq, unique word count, total word count, genres

Appendix D - Total word count per observed genres

```
genius = pd.read_csv('final.csv', header=0, index_col=0)

g_rap_songs = genius[genius['genres'].str.contains("'Rap'")]
g_rap_songs.reset_index(drop=True, inplace=True)

g_pop_songs = genius[genius['genres'].str.contains("'Pop'")]
g_pop_songs.reset_index(drop=True, inplace=True)

g_country_songs = genius[genius['genres'].str.contains("'Country'")]
g_country_songs.reset_index(drop=True, inplace=True)

g_rb_songs = genius[genius['genres'].str.contains("'R&B'")]
g_rb_songs.reset_index(drop=True, inplace=True)

g_rock_songs = genius[genius['genres'].str.contains("'Rock'")]
g_rock_songs.reset_index(drop=True, inplace=True)

# twc = total word count
g_rap_twc = dict()
g_pop_twc = dict()
g_country_twc = dict()
g_rb_twc = dict()
g_rock_twc = dict()

for index, row in g_rap_songs.iterrows():
    d = ast.literal_eval(g_rap_songs.loc[index, 'word freq'])
    for word in d:
        if word in g_rap_twc:
            g_rap_twc[word] += d[word]
        else:
            g_rap_twc[word] = d[word]

g_rap_twc_df = pd.DataFrame(list(g_rap_twc.items()), columns=['word', 'total count'])
g_rap_twc_df = g_rap_twc_df.sort_values(by='total count', ascending=False)
g_rap_twc_df.reset_index(inplace=True, drop=True)

for index, row in g_pop_songs.iterrows():
    d = ast.literal_eval(g_pop_songs.loc[index, 'word freq'])
    for word in d:
        if word in g_pop_twc:
            g_pop_twc[word] += d[word]
        else:
            g_pop_twc[word] = d[word]

g_pop_twc_df = pd.DataFrame(list(g_pop_twc.items()), columns=['word', 'total count'])
g_pop_twc_df = g_pop_twc_df.sort_values(by='total count', ascending=False)
g_pop_twc_df.reset_index(inplace=True, drop=True)

for index, row in g_country_songs.iterrows():
    d = ast.literal_eval(g_country_songs.loc[index, 'word freq'])
    for word in d:
        if word in g_country_twc:
            g_country_twc[word] += d[word]
        else:
            g_country_twc[word] = d[word]

g_country_twc_df = pd.DataFrame(list(g_country_twc.items()), columns=['word', 'total count'])
g_country_twc_df = g_country_twc_df.sort_values(by='total count', ascending=False)
g_country_twc_df.reset_index(inplace=True, drop=True)

for index, row in g_rb_songs.iterrows():
    d = ast.literal_eval(g_rb_songs.loc[index, 'word freq'])
```

```
    for word in d:
        if word in g_rb_twc:
            g_rb_twc[word] += d[word]
        else:
            g_rb_twc[word] = d[word]

g_rb_twc_df = pd.DataFrame(list(g_rb_twc.items()), columns=['word', 'total count'])
g_rb_twc_df = g_rb_twc_df.sort_values(by='total count', ascending=False)
g_rb_twc_df.reset_index(inplace=True, drop=True)

for index, row in g_rock_songs.iterrows():
    d = ast.literal_eval(g_rock_songs.loc[index, 'word freq'])
    for word in d:
        if word in g_rock_twc:
            g_rock_twc[word] += d[word]
        else:
            g_rock_twc[word] = d[word]

g_rock_twc_df = pd.DataFrame(list(g_rock_twc.items()), columns=['word', 'total count'])
g_rock_twc_df = g_rock_twc_df.sort_values(by='total count', ascending=False)
g_rock_twc_df.reset_index(inplace=True, drop=True)

g_rap_twc_df.to_csv('genius_rap_twc.csv')
g_pop_twc_df.to_csv('genius_pop_twc.csv')
g_country_twc_df.to_csv('genius_country_twc.csv')
g_rb_twc_df.to_csv('genius_rb_twc.csv')
g_rock_twc_df.to_csv('genius_rock_twc.csv')
```

Produces CSVs to feed into wordcloud generating Python file (Appendix E)
The same process was used on the kaggle dataset to generate kaggle total word counts..

Appendix E - *genius_wordcloud.py*

```
import pandas as pd
import matplotlib.pyplot as plt
from wordcloud import WordCloud

g_rap_twc_df = pd.read_csv('genius_rap_twc.csv', index_col=0)
g_pop_twc_df= pd.read_csv('genius_pop_twc.csv', index_col=0)
g_country_twc_df = pd.read_csv('genius_country_twc.csv', index_col=0)
g_rb_twc_df = pd.read_csv('genius_rb_twc.csv', index_col=0)
g_rock_twc_df = pd.read_csv('genius_rock_twc.csv', index_col=0)

g_rap_text_freq = dict()
g_pop_text_freq = dict()
g_country_text_freq = dict()
g_rb_text_freq = dict()
g_rock_text_freq = dict()

for w, f in g_rap_twc_df.values:
   g_rap_text_freq[w] = f

for w, f in g_pop_twc_df.values:
   g_pop_text_freq[w] = f

for w, f in g_country_twc_df.values:
   g_country_text_freq[w] = f

for w, f in g_rb_twc_df.values:
   g_rb_text_freq[w] = f

for w, f in g_rock_twc_df.values:
   g_rock_text_freq[w] = f

wordcloud = WordCloud().generate_from_frequencies(g_rap_text_freq)
plt.imshow(wordcloud)
plt.axis("off")
plt.title("Word cloud for Genius rap songs")
plt.show()

wordcloud = WordCloud().generate_from_frequencies(g_pop_text_freq)
plt.imshow(wordcloud)
plt.axis("off")
plt.title("Word cloud for Genius pop songs")
plt.show()

wordcloud = WordCloud().generate_from_frequencies(g_country_text_freq)
plt.imshow(wordcloud)
plt.axis("off")
plt.title("Word cloud for Genius country songs")
plt.show()

wordcloud = WordCloud().generate_from_frequencies(g_rb_text_freq)
plt.imshow(wordcloud)
plt.axis("off")
plt.title("Word cloud for Genius R&B songs")
plt.show()

wordcloud = WordCloud().generate_from_frequencies(g_rock_text_freq)
plt.imshow(wordcloud)
plt.axis("off")
plt.title("Word cloud for Genius rock songs")
plt.show()
```

The same process was used on the kaggle dataset to generate kaggle word clouds.

Appendix F - Get first genre from ‘genres’ list. Remove row if the first genre is not of Rap, Pop, Country, R&B, Rock

```
genius_one_genre = genius.copy()

for index, row in genius_one_genre.iterrows():
    lst = ast.literal_eval(genius_one_genre.loc[index, 'genres'])
    genius_one_genre.loc[index, 'genres'] = lst[0]

obs_genres = ['Rap', 'Pop', 'Country', 'R&B', 'Rock']
genius_one_genre_copy = genius_one_genre.copy()

for index, row in genius_one_genre_copy.iterrows():
    genre = str(genius_one_genre_copy.loc[index, 'genres'])
    if genre not in obs_genres:
        genius_one_genre.drop(index, inplace=True)

genius_one_genre.reset_index(inplace=True, drop=True)

genius_one_genre.to_csv('genius_first_genre.csv')
```

Appendix G - *Cut_Genius.py*

Original data frame of 331,204 rows was too large to control. Removed songs of Rap and Pop genres, which were much larger than the other genres, by 75%

```
og_genius = pd.read_csv('genius_first_genre.csv', index_col=0)
print("Shape of original df: ", str(og_genius.shape[0]))

genre_counts = og_genius['genres'].value_counts()
print(genre_counts)

rap = og_genius[og_genius['genres'] == 'Rap']
pop = og_genius[og_genius['genres'] == 'Pop']
rest_lst = ['Rock', 'R&B', 'Country']
rest = og_genius[og_genius['genres'].isin(rest_lst)]
rap_cut = rap.sample(frac = 0.25)
pop_cut = pop.sample(frac = 0.25)
# took out 195105 rows
df = pd.concat([rap_cut, pop_cut, rest])
df.reset_index(drop=True, inplace=True)

print("Shape of new df: " + str(df.shape[0]))
df.to_csv('genius_cut_again.csv')
```

Output:

```
Shape of original df:  331204
Rap        130762
Pop        129377
Rock        50434
R&B         10708
Country      9923
Name: genres, dtype: int64
Shape of new df: 136099
```

Resulted in creating *genius_cut_again.csv* (I ended up doing it twice, but only kept the second result) with 136099 rows

Appendix H - *scaled_research.py* and its outputs

```
import ast
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import LinearSVC, SVC, NuSVC
from sklearn.metrics import classification_report, confusion_matrix , ConfusionMatrixDisplay, accuracy_score
from sklearn import preprocessing

genius = pd.read_csv('genius_cut_again.csv', header=0, index_col=0)

twc = dict()
for index, row in genius.iterrows():
   d = ast.literal_eval(row['word freq'])
   for word in d:
       if word in twc:
           twc[word] += d[word]
       else:
           twc[word] = d[word]

twc_df = pd.DataFrame(list(twc.items()), columns=['word', 'total count'])
twc_df = twc_df.sort_values(by='total count', ascending=False).reset_index(drop=True)

n = 250
twc_df_gte_n = twc_df[twc_df['total count'] >= n]
twc_df_gte_n = twc_df_gte_n.sort_values(by='total count', ascending=False)
twc_df_gte_n.reset_index(inplace=True, drop=True)
print("Number of words with frequency equal to or greater than " + str(n) + " is " + str(twc_df_gte_n.shape[0]))

def func(g):
   if g == 'Rap':
       return 0
   elif g == 'Pop':
       return 1
   elif g == 'Country':
       return 2
   elif g == 'R&B':
       return 3
   elif g == 'Rock':
       return 4

gte_lst = twc_df_gte_n['word'].to_list()

genius['genres'] = genius['genres'].apply(func)

print("Creating new lst")
new_lst = []

for i, row in genius.iterrows():
   for key, value in ast.literal_eval(row['word freq']).items():
       if key in gte_lst and key != "" and key != '\'':
           r = [i, row['genres']]
           r.append(key)
           r.append(value)
           new_lst.append(r)

print("making new lst DF")
lst_df = pd.DataFrame(new_lst, columns=['ID', 'genre', 'word', 'count'])
lst_df['genre'] = lst_df['genre'].astype('uint8')
intermed_df1 = pd.DataFrame(lst_df[['ID', 'genre']].groupby(['ID', 'genre']).size().reset_index(name='drop'))
y = intermed_df1.drop(['ID', 'drop'], axis=1)
y['genre'] = y['genre'].astype('uint8')
print("y made")

del intermed_df1

lst_df['genre'] = lst_df['genre'].astype('category')
lst_genre_and_lyrics = pd.pivot_table(lst_df, values='count', index='ID', columns='word', fill_value=0)
x = lst_genre_and_lyrics.reset_index(drop=True)
print("x made")
print("splitting x and y into train and test data")
train_x, test_x, train_y, test_y = train_test_split(x, y, test_size=0.2,)
# new scaling technique
# only scale x-values
print("scaling training x data")
scaler = preprocessing.StandardScaler().fit(train_x)
x_scaled = scaler.transform(train_x)

print("scaling test x data")
test_scaler = preprocessing.StandardScaler().fit(test_x)
x_test_scaled = test_scaler.transform(test_x)
```

```
# linear SVC
print("Running Linear SVC")
linear = LinearSVC(random_state=0)
linear.fit(x_scaled, train_y.values.ravel())
print("Linear SVC predicting")
y_lin_predicted = linear.predict(x_test_scaled)
linear_score = accuracy_score(test_y, y_lin_predicted)
print("The score of the scaled Linear SVC is: ", linear_score)
cm_lin = confusion_matrix(test_y, y_lin_predicted)
disp_lin = ConfusionMatrixDisplay(cm_lin)
disp_lin.plot()
disp_lin.figure_.savefig("ScaledLinearSVC.png")
print("Confusion matrix of scaled linear SVC downloaded")

# SVC default - RBF
print("Running RBF SVC")
rbf = SVC(kernel='rbf', random_state=0)
rbf.fit(x_scaled, train_y.values.ravel())
print("RBF SVC predicting")
y_rbf_predicted = rbf.predict(x_test_scaled)
rbf_score = accuracy_score(test_y, y_rbf_predicted)
print("The score of the scaled RBF SVC is: ", rbf_score)
cm_rbf = confusion_matrix(test_y, y_rbf_predicted)
disp_rbf = ConfusionMatrixDisplay(cm_rbf)
disp_rbf.plot()
disp_rbf.figure_.savefig("ScaledRBFSVC.png")
print("Confusion matrix of scaled RBF SVC downloaded")
```

Output:

```
>>> exec(open('scaled_research.py').read())
Number of words with frequency equal to or greater than 250 is 6635
Creating new lst
making new lst DF
y made
x made
splitting x and y into train and test data
scaling training x data
scaling test x data
Running Linear SVC
/home/ecelrc/students/agarcia2/Research/venv/lib64/python3.6/site-packages/sklearn/svm/_base.py:986: ConvergenceWarnin
g: Liblinear failed to converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
Linear SVC predicting
The score of the scaled Linear SVC is:  0.5513225569434239
Confusion matrix of scaled linear SVC downloaded
Running RBF SVC
^[[A^[[ARBF SVC predicting
The score of the scaled RBF SVC is:  0.618626010286554
Confusion matrix of scaled RBF SVC downloaded
```

```
# poly SVC
print("Running poly SVC")
poly = SVC(kernel='poly', random_state=0)
poly.fit(x_scaled, train_y.values.ravel())
print("Polynomial SVC predicting")
y_poly_predicted = poly.predict(x_test_scaled)
poly_score = accuracy_score(test_y, y_poly_predicted)
print("The score of the scaled Polynomial SVC is: ", poly_score)
cm_poly = confusion_matrix(test_y, y_poly_predicted)
disp_poly = ConfusionMatrixDisplay(cm_poly)
disp_poly.plot()
disp_poly.figure_.savefig("ScaledPolynomia;SVC.png")
print("Confusion matrix of scaled polynomial SVC downloaded")
```

Output:

```
Running poly SVC
Polynomial SVC predicting
The score of the scaled Polynomial SVC is:  0.3984570168993387
Confusion matrix of scaled polynomial SVC downloaded
```

```
# sigmoid SVC
print("Running sigmoid SVC")
sig = SVC(kernel='sigmoid', random_state=0)
sig.fit(x_scaled, train_y.values.ravel())
print("Sigmoid SVC predicting")
y_sig_predicted = sig.predict(x_test_scaled)
sig_score = accuracy_score(test_y, y_sig_predicted)
print("The score of the scaled Sigmoid SVC is: ", sig_score)
cm_sig = confusion_matrix(test_y, y_sig_predicted)
disp_sig = ConfusionMatrixDisplay(cm_sig)
disp_sig.plot()
disp_sig.figure_.savefig("ScaledSigmoidSVC.png")
print("Confusion matrix of scaled sigmoid SVC downloaded")
```

Output:

```
Running sigmoid SVC
Sigmoid SVC predicting
The score of the scaled Sigmoid SVC is:  0.6180382072005878
Confusion matrix of scaled sigmoid SVC downloaded
```

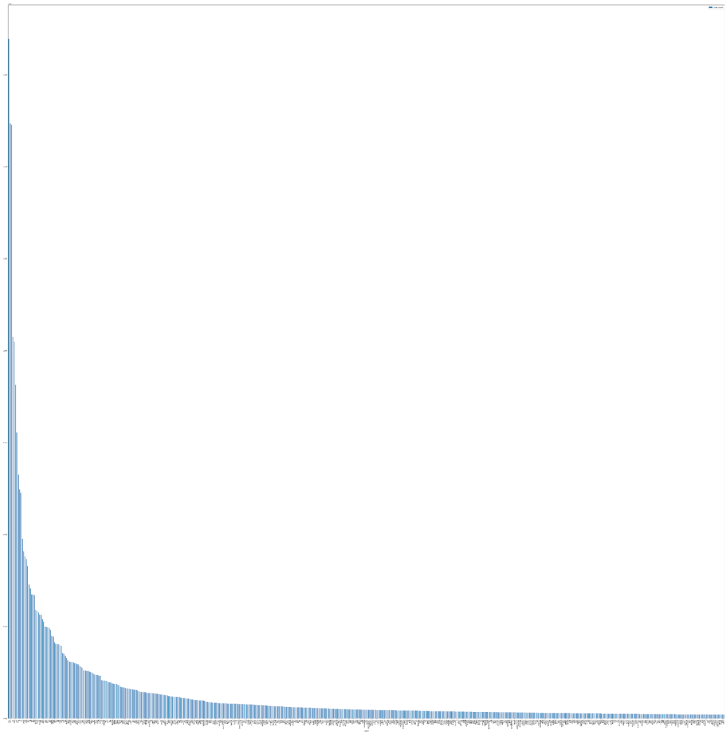
```
# MultinomialNB
print("Running Multinomial Naive Bayes. Did NOT use scaled input data.")
mult = MultinomialNB()
mult.fit(train_x, train_y.values.ravel())
print("Multinomial Naive Bayes predicting")
y_mult_predicted = mult.predict(test_x)
mult_score = accuracy_score(test_y, y_mult_predicted)
print("The score of the Multinomial Naive Bayes is: ", mult_score)
cm_mult = confusion_matrix(test_y, y_mult_predicted)
disp_mult = ConfusionMatrixDisplay(cm_mult)
disp_mult.plot()
disp_mult.figure_.savefig("MultinomialNB.png")
print("Confusion matrix of Multinomial Naive Bayes downloaded")
```

Output:

```
Running Multinomial Naive Bayes. Did NOT use scaled input data.
Multinomial Naive Bayes predicting
The score of the scaled Multinomial Naive Bayes is:  0.5591844232182219
Confusion matrix of Multinomial Naive Bayes downloaded
```

Appendix I: Graphs A, B, C, D

Graph A - Kaggle dataset: word frequency > 10,000 with a list of sample of words

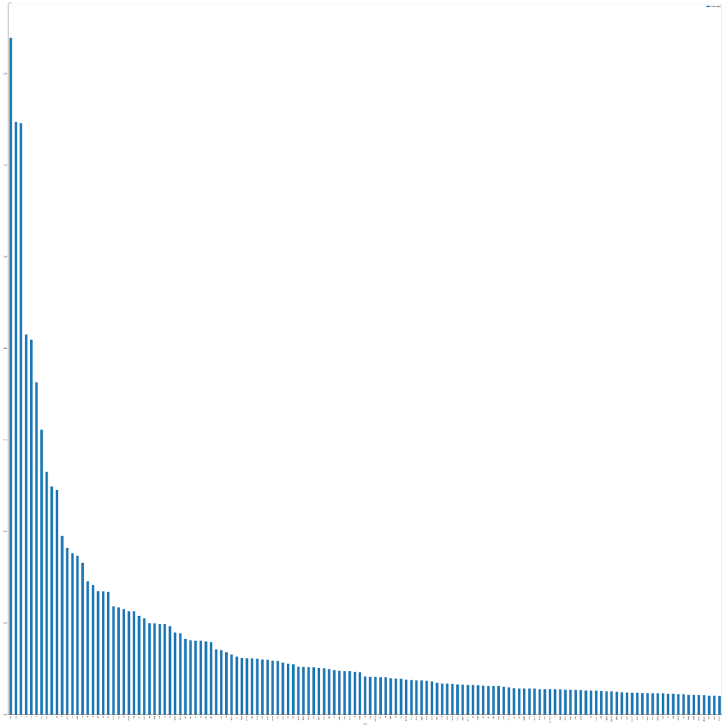


	word	total count
0	the	1847028
1	you	1617734
2	i	1614139
3	to	1036969
4	and	1023380
5	a	906959
6	me	777608
7	my	662673
8	it	622447
9	in	613054
10	of	487537
11	your	454512
12	on	440080
13	that	433043
14	i'm	414219
15	all	364228
16	is	353028
17	we	336816
18	be	336087
19	for	335005

...

	word	total count
514	peace	10448
515	honey	10428
516	car	10425
517	eye	10416
518	you'd	10402
519	doesn't	10398
520	e	10343
521	sea	10334
522	gets	10311
523	cut	10250
524	apart	10219
525	calling	10195
526	burning	10176
527	catch	10168
528	moment	10120
529	standing	10119
530	who's	10112
531	loved	10086
532	aint	10035
533	takes	10011

Graph B - Kaggle dataset: word frequency > 50,000 with a list of sample of words

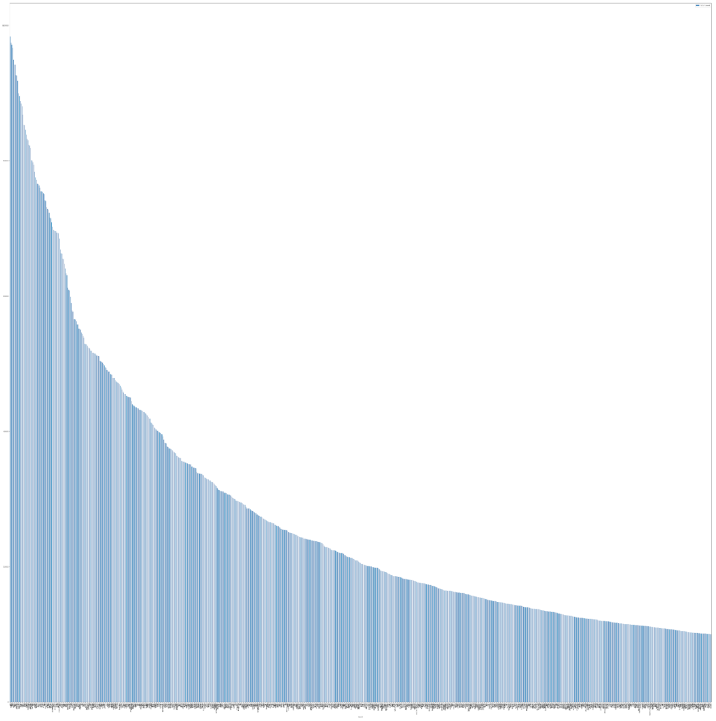


	word	total count
0	the	1847028
1	you	1617734
2	i	1614139
3	to	1036969
4	and	1023380
5	a	906959
6	me	777608
7	my	662673
8	it	622447
9	in	613054
10	of	487537
11	your	454512
12	on	440080
13	that	433043
14	i'm	414219
15	all	364228
16	is	353028
17	we	336816
18	be	336087
19	for	335005

...

	word	total count
119	said	60473
120	over	59888
121	around	59622
122	them	59103
123	look	58542
124	won't	58404
125	eyes	57882
126	every	57862
127	well	57854
128	his	56502
129	had	55978
130	we're	55551
131	am	55193
132	long	53847
133	mind	53660
134	would	53246
135	always	52768
136	find	51196
137	ever	51136
138	home	50264

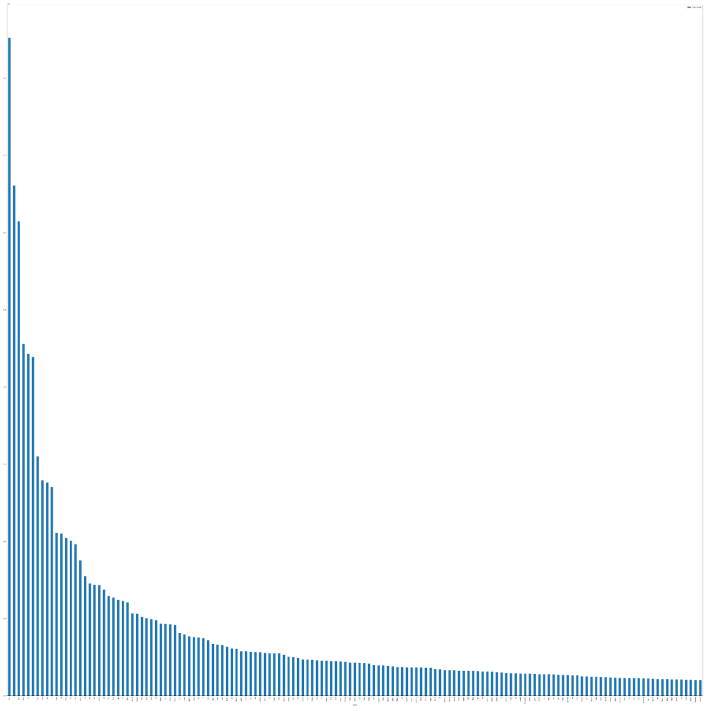
Graph C - Genius dataset: 10,000 < word frequency < 100,000 with a list of sample of words



	word	total count
0	am	98361
1	won't	97420
2	well	97152
3	call	96675
4	again	94915
5	eyes	94213
6	him	94183
7	really	92646
8	always	92559
9	new	91790
10	'cause	89940
11	there's	89553
12	stop	88825
13	ever	88435
14	'em	88042
15	long	86789
16	into	85309
17	live	85255
18	find	84562
19	hard	83847

	word	total count
807	hole	10234
808	insane	10233
809	magic	10217
810	loose	10202
811	somethin'	10177
812	'round	10162
813	answer	10160
814	momma	10132
815	raise	10131
816	goin	10124
817	longer	10108
818	breaking	10103
819	tryin	10101
820	week	10095
821	lil'	10088
822	stone	10064
823	whats	10063
824	test	10050
825	freak	10036
826	heads	10033

Graph D - Genius dataset: word frequency > 100,000 with a list of sample of words



	word	total count
0	the	4260424
1	i	3304195
2	you	3072624
3	and	2278348
4	to	2214509
5	a	2194231
6	my	1550519
7	me	1395444
8	in	1380540
9	it	1352490
10	like	1054218
11	of	1050216
12	that	1021694
13	on	1004498
14	i'm	981121
15	your	877907
16	is	773575
17	we	726644
18	for	718390
19	with	716218

	word	total count
127	every	117533
128	night	117323
129	money	115756
130	look	115295
131	an	114928
132	i've	114624
133	put	114563
134	around	113054
135	over	111352
136	gotta	110253
137	its	108427
138	heart	108189
139	little	107508
140	mind	106240
141	than	105887
142	yo	104515
143	real	103863
144	even	103297
145	better	101863
146	would	100640

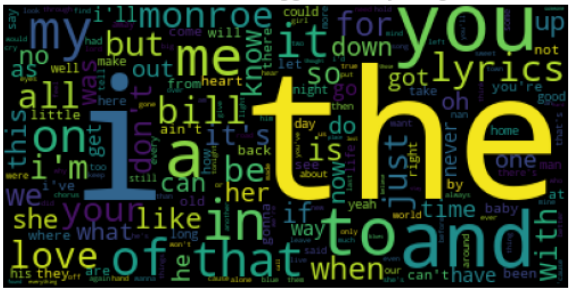
Appendix J - Images A, B, C

Image A - word clouds for kaggle

Word cloud for Kaggle rap songs

Word cloud for Kaggle pop songs

Word cloud for Kaggle country songs



Word cloud for Kaggle R&B songs

Word cloud for Kaggle rock songs

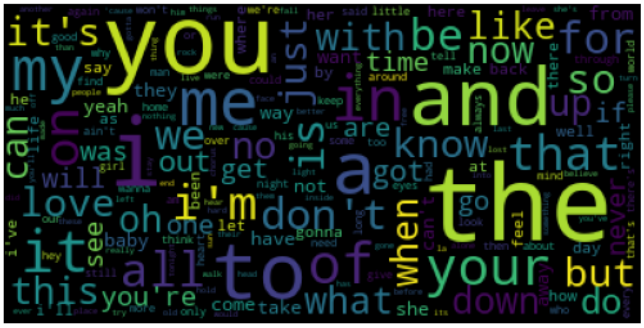


Image B - word clouds for genius

Word cloud for Genius rap songs

Word cloud for Genius pop songs

Word cloud for Genius country songs

Word cloud for Genius R&B songs

Word cloud for Genius rock songs

Image C - Kaggle dataset elimination

Bright Sunny Day (Unreleased)	Neil Young	if you wake up in the morning light something doesn't seem or know is right lay back and let your feeling show come on and let the people know oh that's the price that you pay oh that's the price that you pay just _____ money your baby _____ _____ now come on boys _____ oh that's the price that you pay oh that's the price that you pay oh that's the price that you pay oh that's the price that you pay bad news is just an excuse get down and let yourself keep loose go on and live for tomorrow so much spent today in sorrow oh that's the price that you pay	{'if': 1, 'you': 8, 'wake': 1, 'up': 1, 'in': 2, 'the': 9, 'morning': 1, 'light': 1, 'something': 1, "doesn't": 1, 'seem': 1, 'or': 1, 'know': 2, 'is': 2, 'right': 1, 'lay': 1, 'back': 1, 'and': 4, 'let': 3, 'your': 2, 'feeling': 1, 'show': 1, 'come': 2, 'on': 3, 'people': 1, 'oh': 7, "that's": 7, 'price': 7, 'that': 7, 'pay': 7, 'just': 2, '_____': 1, 'money': 1, 'baby': 1, '_____': 1, '_____': 1, 'now': 1, 'boys': 1, '_____': 1, 'bad': 1, 'news': 1, 'an': 1, 'excuse': 1, 'get': 1, 'down': 1, 'yourself': 1, 'keep': 1, 'loose': 1, 'go': 1, 'live': 1, 'for': 1, 'tomorrow': 1, 'so': 1, 'much': 1, 'spent': 1, 'today': 1, 'sorrow': 1, '': 1}
In a Woman's Life	Jewel	_____ _____ as sung in santa cruz february somewhere between right and wrong is a love song its tongue hits the target but its phone number is always wrong it speaks through calluses of gentler things it speaks of kindness but i don't think you know what that word means your love is a bitter seed that only the blind can see if this is what love is then i guess there's no love song left in me somewhere between right and wrong is reality justice and shame speak foolishly of impractical things your kisses are as kind as candles your hands are as giving as stone your mind is kind as hitler's until night leaves you feeling alone your love is something i must do but never be if this is what love is then i guess there's no love song left in me i didn't mean to fall into darkness it's just that i trip in the lights cause aw man it's hard to see that clearly and not put up a fight but i knew i wpould break and not bend just look at this trouble i'm in i need to realize this is how the unhappy half dies somewhere between good and bad is every lie i have ever been told they all come down but good deeds don't count it's what cards you hold but i'm tired of your grey laughter i am tired of your hungry eyes this love is so rotten it's startin' to attract flies your love is something that must do but never be its blade is made of jealousy and insecurity your love is a bitter seed that only the blind can see if this is what love is if this is what love is then i guess there's no love song left in me _____ _____ as sung in madison nj on april somewhere between right and wrong is a love song its tongue hits the target but its phone	{'_____': 1, 'as': 13, 'sung': 2, 'in': 12, 'santa': 1, 'cruz': 1, 'february': 1, 'somewhere': 6, 'between': 6, 'right': 4, 'and': 14, 'wrong': 6, 'is': 39, 'a': 8, 'love': 26, 'song': 8, 'its': 6, 'tongue': 2, 'hits': 2, 'the': 10, 'target': 2, 'but': 12, 'phone': 2, 'number': 2, 'always': 2, 'it': 5, 'speaks': 4, 'through': 2, 'calluses': 2, 'of': 12, 'gentler': 2, 'things': 4, 'kindness': 1, 'i': 23, "don't": 4, 'think': 2, 'you': 6, 'know': 2, 'what': 12, 'that': 11, 'word': 2, 'means': 2, 'your': 19, 'bitter': 4, 'seed': 4, 'only': 4, 'blind': 4, 'can': 4, 'see': 6, 'if': 8, 'this': 14, 'then': 7, 'guess': 6, "there's": 6, 'no': 6, 'left': 6, 'me': 6, 'reality': 2, 'justice': 2, 'shame': 2, 'speak': 2, 'foolishly': 2, 'impractical': 2, 'kisses': 2, 'are': 4, 'kind': 4, 'candles': 2, 'hands': 2, 'giving': 2, 'stone': 2, 'mind': 2, "hitler's": 2, 'until': 2, 'night': 2, 'leaves': 2, 'feeling': 2, 'alone': 2, 'something': 4, 'must': 4, 'do': 4, 'never': 4, 'be': 4, "didn't": 2, 'mean': 2, 'to': 8, 'fall': 2, 'into': 2, 'darkness': 2, "it's": 7, 'just': 4, 'trip': 2, 'lights': 2, 'cause': 2, 'aw': 2, 'man': 2, 'hard': 2, 'clearly': 2, 'not': 4, 'put': 2, 'up': 2, 'fight': 2, 'knew': 2, 'wpould': 1, 'break': 2, 'bend': 2, 'look': 2, 'at': 2, 'trouble': 2, "i'm": 4, 'need': 2, 'realize': 2, 'how': 2, 'unhappy': 2, 'half': 2, 'dies': 2, 'good': 4, 'bad': 2, 'every': 2, 'lie': 2, 'have': 1, 'ever': 2, 'been': 2, 'told': 2, 'they': 1, 'all': 2, 'come': 1, 'down': 1, 'deeds': 2, 'count': 2, 'cards': 2, 'hold': 2, 'tired': 4, 'grey': 2, 'laughter': 2, 'am': 2, 'hungry': 2, 'eyes': 2, 'so': 2, 'rotten': 2, "startin'": 1, 'attract': 2, 'flies': 2, 'blade': 2, 'made': 2, 'jealousy': 2, 'insecurity': 2, '_____': 1, 'madison': 1, 'nj': 1, 'on': 1, 'april': 1, 'charity': 1, 'well': 1, 'starting': 1, 'would': 1, "i've": 1, 'comes': 1, 'done': 1, 'yeah': 1, '': 1}