Music Genre Classification by Analyzing Song Lyrics

By Group 4: Brennan Casey, Hao Cheng, Shikuan Gao & Zimo Liu

Stevens Institute of Technology

Introduction

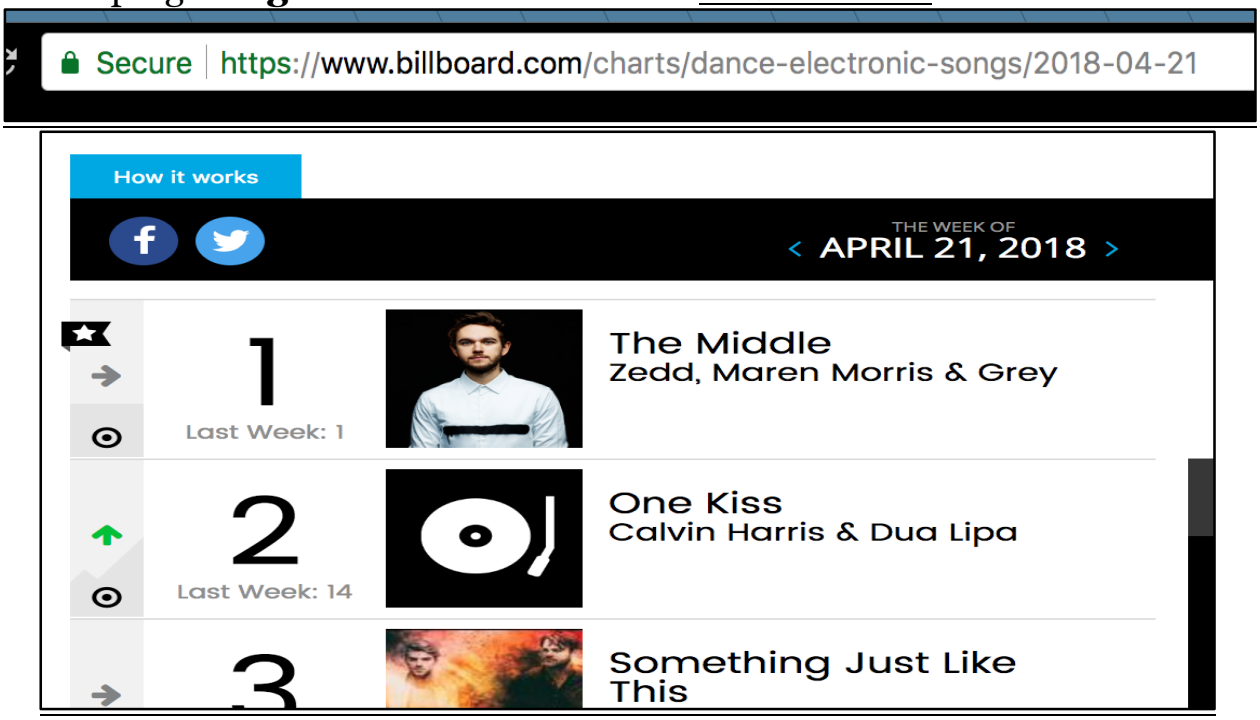
Music is a very interesting industry, because it has very few successes and many small artists trying to make a name for themselves. Could it be the sound or the lyrics that make a hit song? More research needs to be done. We decided to create a classifier which can predict the genre of a song based on the lyrics to determine if the lyrics are a significant factor of the genre. This model could help song recommendation softwares by incorporating the lyrics and their similarities to find songs people might like. Currently spotify mostly provides users with more songs from artists they already listen to, or artists that were featured in songs they already heard. Another site that could benefit from this model is SoundCloud. SoundCloud struggles to organize their user published songs into the correct categories. By taking lyrics from users, the songs can be accurately placed into the communities to which they belong.

Important to note: Word2Vec and Doc2Vec based classifiers and EDA are predicting values for 2 genres(Pop & Country) while the Multinomial Naive Bayes and Support Vector Machine classifiers which use tf-idf and CV are predicting all 4 genres (Pop, Country, Rock, & Rap). The data gathering for these two methods are the same but the other components will be explained separately.

Data Gathering and Preprocessing

Data Gathering
The songs were gathered using a two-step process:

1. Scraping **Song Name** and **Artist** from Billboard.com

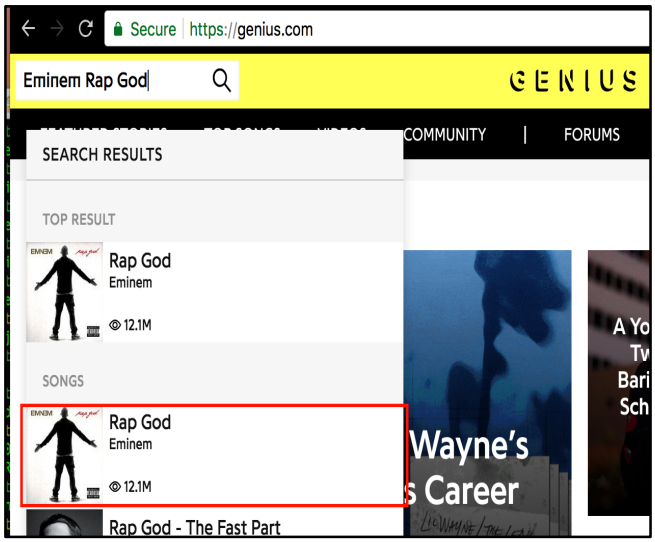


The way Billboard.com's URL is makes scraping very easy. We are able to loop through every month for the last several decades retrieving the top 50 songs for that month. Of course some songs are top of the charts for more than a month, so we only took the unique songs. At the end of scraping, we are left with a dataframe that looks like this:

2.Searching Genius.com using Billboard data and scraping the **Lyrics**

Getting the lyrics off of genius.com was a bit more challenging. The URL's were rarely recreatable like billboard was and if it had a slight error no page would be loaded. The best solution ended up being to search the song name and artist collected earlier and pick the top song result.

```
driver = webdriver.Chrome(executable_path='./chromedriver')
driver.get("https://genius.com/")

def getSongLyrics(songName, artist):
    lyrics = []

    try:
        searchBox = driver.find_element_by_css_selector("input[ng-m
        searchBox.send_keys(artist+' '+songName)
        searchBox.send_keys(Keys.ENTER)
        time.sleep(4)

        results = driver.find_elements_by_css_selector("a[class='mi
        results[1].click()
        time.sleep(4)


        try:
            element = driver.find_element_by_class_name('lyrics')
            element.find_element_by_tag_name('p')

            lyricElement = element.find_elements_by_tag_name('a')
            for line in lyricElement:
                lyrics.append(line.text)
            rockLyrics.append(lyrics)
        except:
            print('Lyrics Not Found')
```



Predicting 2 Genres with Word Vectors

Preprocessing

After importing the dataset, we have extracted the lyrics column. Then we used NLTK package to analyze text. We split the documents into tokens or segments. For the further analysis, we have to clean the text by removing the stop words that are useless. Some messy formatting should also be removed, such as the brackets, punctuations, white spaces, something like that. Then, we also lowered case lyrics and stemmed the text. All those procedures are prepared to better analyze the lyrics text.

During the preprocessing, we have found a package called “ftfy”, which can easily fix the unicode problem. Moreover, there are also some useless words needed to be removed, like “chorus”, “verse 1”, “verse 2” whose function is to separate each paragraph.

In addition, for the English lyrics, we have used tag name to screen only noun, adjective, pronoun and verb, which are meaningful comparing to other words. By doing this step can make the result accuracy much more accurate.

Exploratory Data Analysis

After preprocessing our lyrics dataset, we attempted to use EDA to generate some visualization to better illustrate the data information.

Artist
Firstly, we counted the number of each genres’ artists respectively. As it is shown below, Figure 1 stands for pop artists and Figure 2 stands for country artists. How often do the artists appear? It seems that some artists showed several times in pop songs, the most frequent times is 8. However, in country songs, the most frequent times reach to about 33. In general, most artist show only once. According to the result, we can see that there is almost no similarity between these two types of artists, so we decide to include this column to our features. According to the result, we can see that there is almost no similarity between these two types of artists, so we decide to include this column to our features.

```
pop_lyrics.Artist.value_counts()
```

```
Janelle Monáe          8
Ed Sheeran             8
The Weeknd             8
Hugh Jackman           7
XXXTENTACION           7
Lin-Manuel Miranda     7
Cardi B                6
BTS                    6
Camila Cabello         5
Beyoncé                5
Bruno Mars             5
Virgoun                5
```

Figure 1. Pop

```
country_lyrics.Artist.value_counts()
```

```
Taylor Swift          33
Kacey Musgraves       12
Johnny Cash           12
Elvis Presley          9
Chris Stapleton        8
Thomas Rhett           8
Carrie Underwood       7
Yelawolf               6
Eagles                 6
Keith Urban            6
Bob Dylan              6
Lady Gaga              6
```

Figure 2. Country

Lyrics Length

Secondly, we combined pop lyrics and country lyrics together to see the density distribution of the length of lyrics. It's obvious to see that the songs with about 1000 words have the largest density, which indicates that those songs take up the most percentage of all the songs we have scraped.

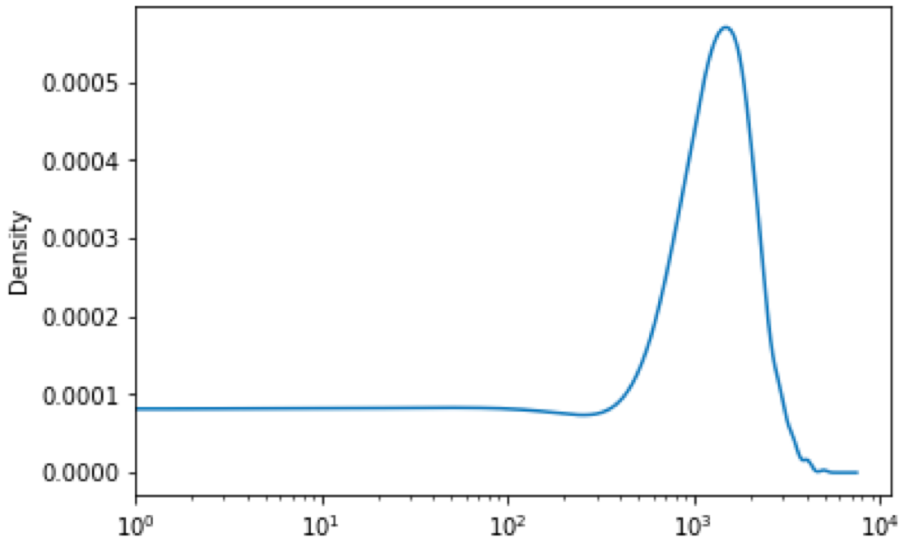


Figure 3.

The below graph compares the lyrics length distribution of two genres (pop and country). Comparing with the right genre, the length of left genre has more intensive distribution. As shown in the graph, the pop lyrics are usually longer than the country lyrics represented as 1.0, but the difference is not big enough to separate these two type clearly.

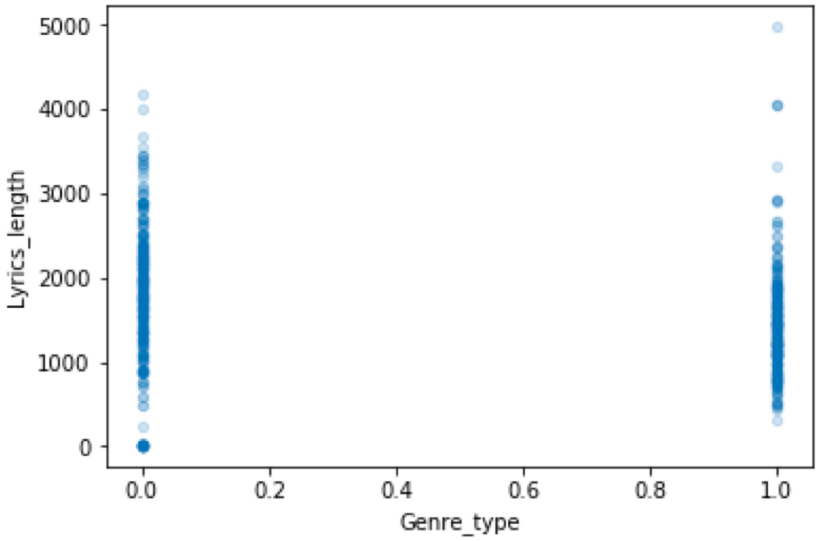


Figure 4.

Word Cloud

Word Cloud is a visualization method that displays how frequently words appear in a given body of text, by making the size of each word proportional

to its frequency. All the words are then
arranged in a cluster or cloud of words. The more a specific word appears, the bigger and bolder it shows in the word cloud.

Before doing this process, we have extracted each genres' lyrics separately and clean the text by removing the stop words and some messy format. Meanwhile, we have only kept the Noun, Adjective, Verb and Pronoun, which are meaningful for the later analysis.

At first, we just drew the word cloud directly, but found there are some words appearing so many times, like Figure 5 shows. To fix this problem, we extracted collocations, which we can disable using the collocations options and make collocations is "False".

Figure 5.

Then, we generated word cloud for pop lyrics and country lyrics as shown below (Figure 6, Figure 7). For the pop lyrics, words like "Know", "Love", "Say" stand out significantly which means these words were used the most frequently in the pop lyrics. When it comes to the country lyrics, words like "Know", "Love", "Say", "time" are the most apparent.

Figure 6. Pop lyrics

Figure 7. Country lyrics

From the word cloud, you can easily tell the similarities and differences between the two genres. For instance, there are some common words in both genres, which means maybe there are some similar types of songs in both genres. You can also find some differences if you see it clearly. In the pop lyrics, you can get some dirty words like “f*ck”, but the country lyrics seem to be much purer. But, it appears too many common words which makes us hardly tell the differences.

So, we decided to drop all the words that have been shown in both genres in the next step. We have set a list called common_word which consists of all the common words. Then, we dislodged those common words in both pop and country lyrics word bag. Finally, we came up with another word cloud of the two genres (Figure 8, Figure 9).

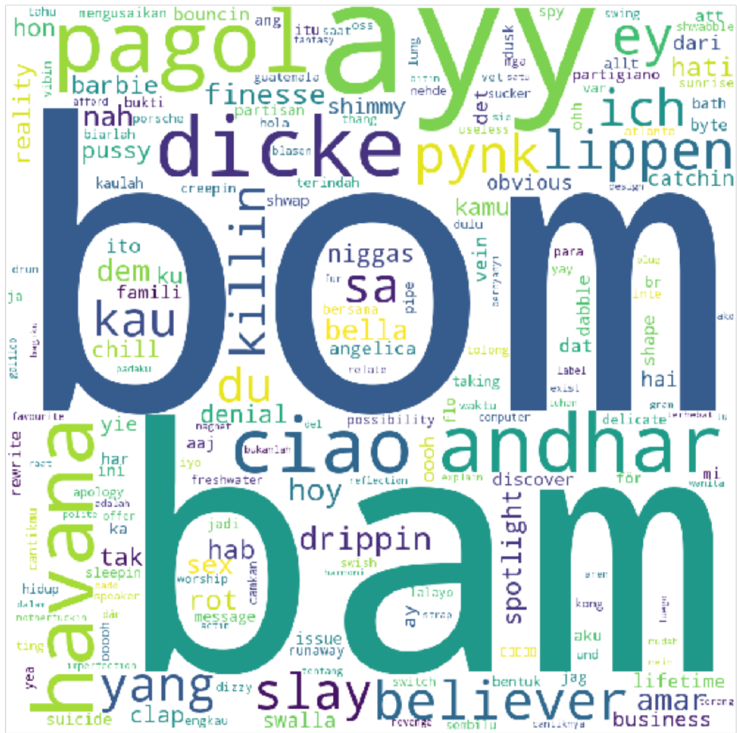


Figure 8. Pop_word_unique

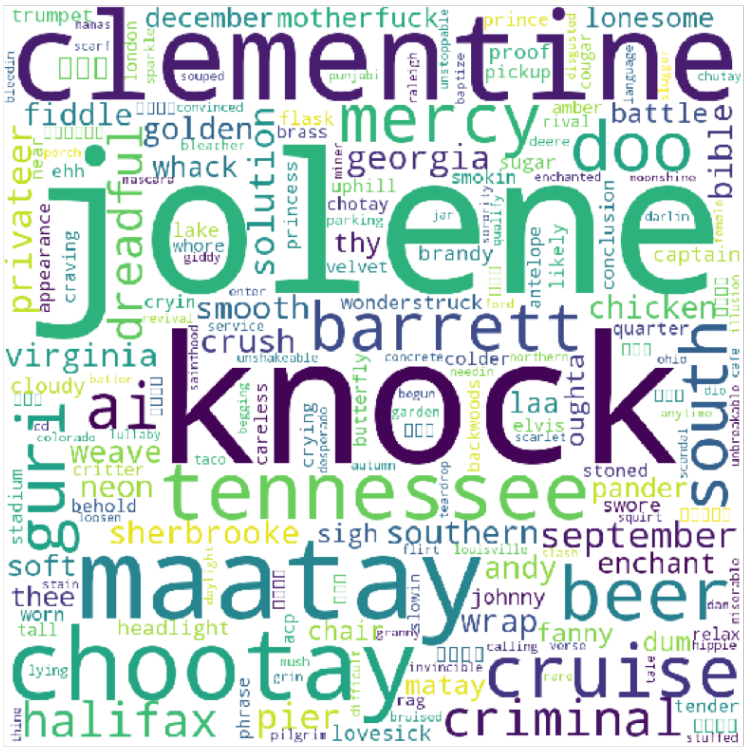


Figure 9. Country_word_unique

In all, we can easily tell the differences between the four genres. Here for pop lyrics, there appears so many oral words and some buzzwords. However, it seems there are many political and lyrical words. It make us clearly get the main idea of each genres.

Features and Word Vector Training

According to EDA part, we decide to select the lyrics text and the artists name as our features to build our model. And to convert the the text to some matrices the classifier model can processed, we try several word vector model and test them.

Word2Vec

We first try the Word2Vec to produce our word vector. Word2Vec has many parameters need to be tuned to fit our data. First one is the dimensionality of the output vectors ‘size’. When this feature get larger, the vectors will

have more dimensions which will make it easier to classify those words from different type. But at same time, the model will consume a lot resource to realize and if it over a threshold, the improvement is not obvious.

And the second one is the parameter called 'min_count' which is used to remove those words with low frequency appearance. Firstly, we want to use it to decrease the impact from some abnormal words like mimetic word in lyrics. But from the EDA portion, we find that actually some mimetic word can classify the genres more clearly. Because the mood in different genre are usually not same, the mimetic words they used are different. So we set this value very low just remove those words with very low frequency but not the mimetic words with little higher frequency.

The next one is the 'window'. It can decide the window size of adjacent words used to estimate the vector of the current word. Because the length of song lyrics sentence are usually short. So we set this value equals to the mean value of the lyrics sentence length. This will make the vector capture more information of the current topic or sentence.

The last one is the 'sample'. This one is for configuring which higher-frequency words are randomly downsampled. There some high frequency words like the EDA portion said. This one is used to punish them to highlight those more special but with lower frequency words.

Doc2Vec

After introduce to the Word2Vec. It is clearly to move on to the second word vectors method we used, Doc2Vec. Doc2vec is an unsupervised algorithm to generate vectors for sentence/paragraphs/documents. It is an algorithm that employs a neural network to map words to a vector space called word vectors. The vectors generated by doc2vec can be used for tasks like finding the most similar sentences/paragraphs/documents. In this project, the task is to find the most similar songs. For sentence similarity, doc2vec vectors may perform well. However, if the input corpus is one with lots of misspellings like tweets or Facebook comments, this algorithm may not be the best choice. Fortunately, in this project out data is the song lyrics we scrape from main music websites. After processing, the situation mentioned above may not happen.

To initialize the Doc2Vec model, we'll first do the following show as the figure below. It is important to understanding each parameter setting. Let's go over to explain each parameter. Alpha is the initial learning rate. Essentially, the learning rate is how quickly a network abandons old beliefs for new ones. In general, we want the learning rate that is low enough to do the useful modeling, but also high enough to shorten the time for training the model. Min_alpha is exactly what it sounds like, the minimum alpha can be, the learning rate will be reduced to the minimum learning rate as the training process. Workers is the number of threads used to train the model. Min_count specifies a term frequency that must be met for a word to be considered by the model. For example, in our model it means that if a word appears less than 2 which is 1, the word will not be considered in the model. Window is the maximum distance between the current and predicted word within a sentence. Size is the number of dimensions. Unlike most numerical datasets that have only 2 dimensions, text data can have hundreds or even more. In this case, we set up the size to be 300 because each genre of the song is 300.

```
from gensim.models.doc2vec import Doc2Vec

model = Doc2Vec(
    alpha=0.025,
    min_alpha=0.025,
    workers=15,
    min_count=2,
    window=10,
    size=300,
    iter=20,
    sample=0.001,
    negative=5
)
```

Figure.10

The second step is to build the vocabulary and train the model. To save the model so we can use it later without training it again, we'll use model.save and load it using Doc2Vec.load.

After building the Doc2Vec model. Each Word has its own vector representations. For example, the figure shown below is the world ‘happy’ own vector representation.

```
model.wv.word_vec('happy')
```

```
array([-8.4486394e-04, -6.2790845e-04,  9.9712610e-04, -9.3378412e-04,
        5.5724307e-04, -3.0082569e-04,  4.1908762e-04, -6.4023130e-04,
       -4.7016767e-04, -4.7470321e-04,  1.5864313e-03, -1.3716890e-03,
        9.5139036e-04,  1.1780156e-03,  6.7519024e-04, -3.0426827e-04,
        1.3228344e-04,  1.4654811e-03, -2.6862454e-04,  1.4381481e-03,
       -1.3017675e-03, -9.0168296e-06, -8.9505909e-04,  5.6609948e-04,
        3.5463987e-04,  1.3222456e-03, -1.5003211e-03, -1.3768022e-03,
        1.2367138e-03,  1.0941421e-03, -3.0891868e-05, -2.0405003e-04,
        4.4574792e-04,  4.8933667e-04,  1.6516810e-03, -1.1747207e-03,
        3.6451855e-04,  1.5079493e-03,  1.2469344e-03, -7.6224492e-04,
       -3.1475394e-04, -9.4337598e-04,  1.3131623e-03, -1.1894562e-03,
       -9.5697981e-04, -1.2363881e-04, -1.3411731e-03, -1.1639313e-03,
        4.4448019e-04,  1.3913091e-04,  1.6081205e-03, -7.0629205e-04,
       -7.6563761e-04,  5.6726881e-04,  7.4246107e-04, -1.8695786e-04,
        7.2708831e-04,  3.9750629e-04,  1.4232759e-03, -2.4419455e-04,
       -1.0720497e-03,  1.2227929e-03,  6.4352952e-04,  4.3648155e-04,
       -2.5523841e-04, -1.0727564e-03,  7.9821162e-05,  2.3714521e-04,
        1.5713818e-03,  4.6053238e-04,  1.4418486e-03,  1.2198903e-03,
        9.4975543e-04, -5.3390657e-04,  7.2610693e-04,  4.4422817e-05,
```

Figure.11

Third, we'll find the most similar words given a target word. Similar words refers to words that have similar vector representations. We start a test with the most similar method which finds the top 10 words most similar to the target word.

The goal of our project is not only finding the similar word, move on to the last step, we'll find the most similar songs given a target word. Using the function and model we build, we can find the top n most similar songs to a target word. From the figure show below, when given 'country' as our target word, it appears top 10 songs that that most related and similar to the word 'country'. It is clearly to see that some famous country musician such as 'Dolly Parton', 'Taylor Swift' dominate the results.

```
print_songs(
    model.docvecs.most_similar([model['country']], topn=10)
)
```

```
[['Dolly Parton', 'Jolene', 0.16160523891448975],
 ['Virgoun', 'Bukti', 0.15511712431907654],
 ['Larray', 'First Place (The Race - Remix)', 0.14716432988643646],
 ['Hugh Jackman', 'The Other Side', 0.13595302402973175],
 ['Demi Lovato', 'Sorry Not Sorry', 0.13429497182369232],
 ['Taylor Swift', 'Enchanted', 0.13356564939022064],
 ['Johnny Cash', 'You Are My Sunshine', 0.1256883293390274],
 ['5 Seconds of Summer', 'Youngblood', 0.11990809440612793],
 ['Cakra Khan', 'Kekasih Bayangan', 0.11939827352762222],
```

Figure.12

Classifier Modeling

After we get the corpus vectors, we need to calculate the feature vector for every song in our data set. We tried two methods for that. But regardless of which one, the first step are same, calculating the mean vector for every song lyrics by sum up all word vectors and divide by the number of words as shown in the figure.

```
def buildWordVector(text, size):
    vec = np.zeros(size).reshape((1,size))
    count = 0.
    for word in text:
        try:
            vec += imdb_w2v[word].reshape((1,size))
            count += 1.
        except KeyError:
            continue
    if count != 0 :
        vec /= count
    return vec
```

Figure 13. build vector for every song lyrics

Next step we use scale or normalize to process all lyrics vectors. Standardization is to scale all vectors to make them shrink to zero and the std equals to 1 (mostly use for Gradient Descent). Normalization is to normalize all vectors with same length but different direction.

```
from sklearn.preprocessing import scale

train_vecs = np.concatenate([buildWordVector(z,n_dim) for z in X_train])
train_vecs = scale(train_vecs)

test_vecs = np.concatenate([buildWordVector(z,n_dim) for z in X_test])
test_vecs = normalize(test_vecs)
```

Figure 14. scale and normalize

SVM

We first use SVM to build our model. The first reason is SVM is good for small data set which we only have. And it's helpful to classify types from

high dimensions vectors like what we build for each song lyrics. And here is the example using Word2Vec with standarzition (scale).

```
clf = svm.SVC(C=0.9, kernel='linear', gamma=20, decision_function_shape='ovo')
clf.fit(train_vecs,y_train)
clf.score(train_vecs,y_train)
clf.score(test_vecs,y_test)
```

```
0.82222222222222219
```

Figure 15. SVM model

RandomForest

We secondly try the RandomForest for our data. Although we only have two features, lyrics text and artist name, the vectors to represent songs have lots of dimensions which is just like those features for decision trees in RandomForest. And with this model, we can check which feature or dimension have highest influence on the result. Then the word with the vector has large value on that dimension is more useful to classify the genre.

This model has lots of parameters can be tuned. They just decide what the trees like in the forest, like how deep it is and how many leaves requires for each node. Because the features, the dimension of our vectors have no actual meaning, we can't figure out the relation between the values of parameters and the accuracy of the output. Then we use GridSearchCV to tune them automatically as the figure shown below.

```
param_test4 = {'max_depth':range(1,5), 'min_samples_split':range(2,10),'min_samples_leaf':range(1,5)}
gsearch4 = GridSearchCV(estimator = RandomForestClassifier(n_estimators=130,max_depth=1,min_samples_split=2,min_samples_leaf=42),
gsearch4.fit(train_vecs,y_train)
print(gsearch4.best_params_)
print(gsearch4.best_score_)
```

```
{'max_depth': 3, 'min_samples_leaf': 1, 'min_samples_split': 6}
0.83971291866
```

Figure 16. GridSearchCV

SGDClassifier

This model has a parameter called loss function which can be set to different function to decide which linear classifier will be used. We just use this to try other different model for our data like logistic regression. With SGD training, this classifier is more helpful for large-scale and sparse machine learning, and it mostly used for text classification and NLP. Here

is the example to use this model.

```
lr = SGDClassifier(loss = 'log', penalty='l1')
lr.fit(train_vecs, y_train)
print('test accuraccy: %.2f'%lr.score(test_vecs,y_test))
lr = SGDClassifier(loss = 'log', penalty='l2')
lr.fit(train_vecs, y_train)
print('test accuraccy: %.2f'%lr.score(test_vecs,y_test))
```

```
C:\ProgramData\Anaconda3\envs\scraping\lib\site-packages\skle
r and tol parameters have been added in <class 'sklearn.linea
eft unset, they default to max_iter=5 and tol=None. If tol is
max_iter will be 1000, and default tol will be 1e-3.
  "and default tol will be 1e-3." % type(self), FutureWarning
```

```
test accuraccy: 0.82
test accuraccy: 0.85
```

Figure 17. SGDClassifier

Results Table

	Word2Vec	Doc2Vec
SVM (accuracy)	0.82	0.59
RandomForest (accuracy)	0.81	0.82
SGDClassifier-Logistic Regression(accuracy)	0.85	0.52

Predicting All 4 Genres

B. Casey

Preprocessing

From the data scraped we had: 1300 Rap Songs, 1300 Rock Songs, 299 Pop Songs, & 300 Country Songs. The first step I took to clean this data is to remove rows which had no lyrics. If the scrapers had an error on some of the pages, something need to fill that index temporarily. The next step was to remove all special character characters and line breaks for obvious reasons.

```
def isEnglish(s):
    try:
        s.encode(encoding='utf-8').decode('ascii')
    except UnicodeDecodeError:
        return False
    else:
        return True
```

I used this function to remove as many songs that were non-english. This would improve the accuracy of the model when predicting using english lyrics which was the goal of the project. Some songs that were not completely english still made it into the final dataset which can be improved upon in the future along with more data.

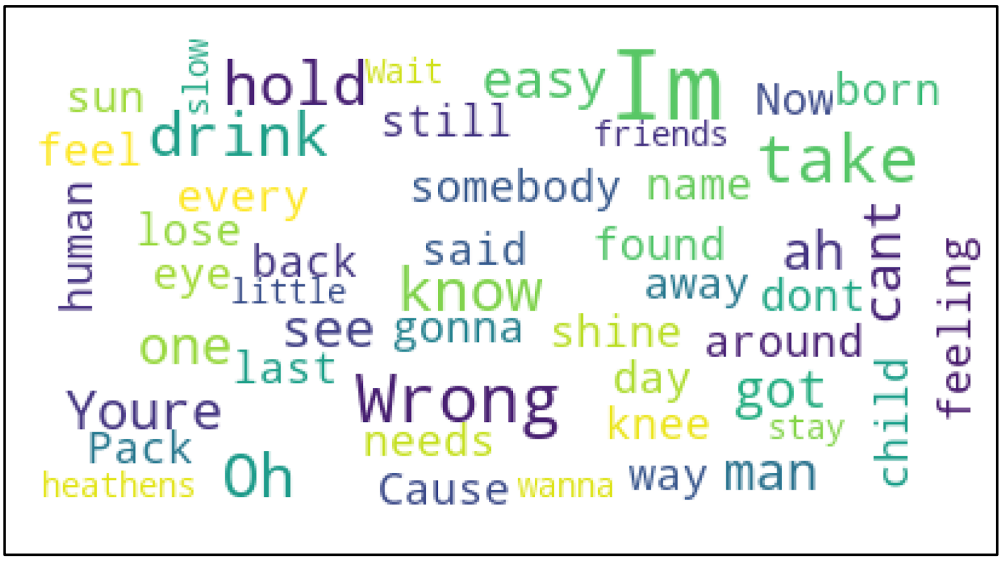
Genre	Artist	Lyrics	Song	Song_Name
Pop	197	197	0	197
Rap	1069	1069	1069	0
Rock	1112	1112	1112	0
country	240	240	0	240

After cleaning the data, this is the final dataset that was outputted for modeling. We can see that Pop and country could definitely use more lyric data and we will have to use RandomOverSampler to balance the data.

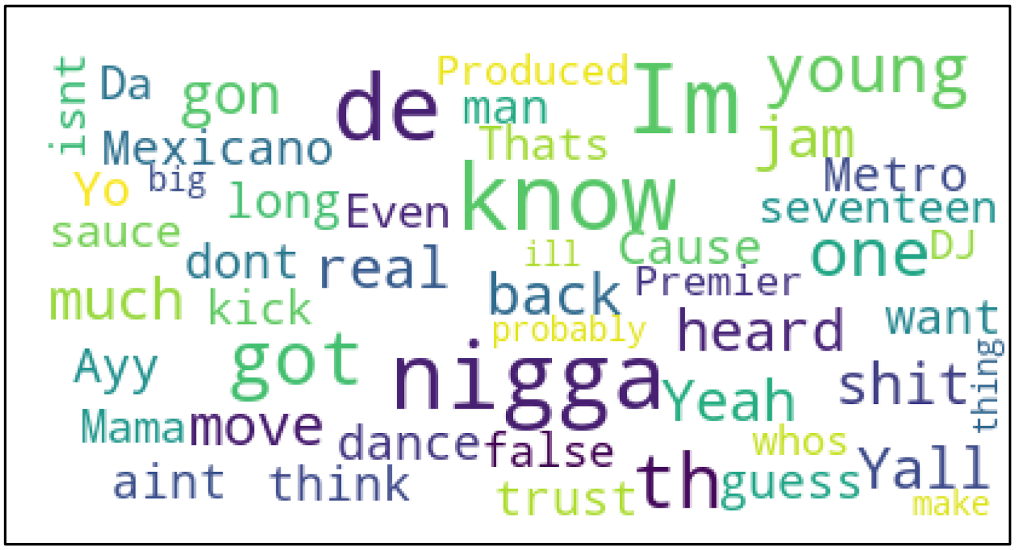
Exploratory Data Analysis

From the earlier EDA, there is not much different. Although understanding the frequencies of unique words for Rap and Rock is very important. I have developed word clouds for those.

Rock Unique Word Frequency



Rap Unique Word Frequency



If you are at familiar with rap or rock genres, you can easily tell the difference between the two and think of a lyric that features many of these words. Using CountVectorizer and TF-IDF should do the trick and output some decent results because the word frequencies seem to be a solid data point. If these Word Clouds ended up being rather vague in similarity, I would have to extract more features from the lyrics like word length or tri-grams.

Training

```
X_train, X_test, y_train, y_test = train_test_split(lyricData['Lyrics'].values,
                                                    lyricData['Genre'].values,
                                                    test_size=0.20, shuffle=True)
```

The first step for training was to load my data and split the data into train and test sets.

After that, I chose two pipelines- the first being MultinomialNB and the second being Support Vector Machine (SVM).

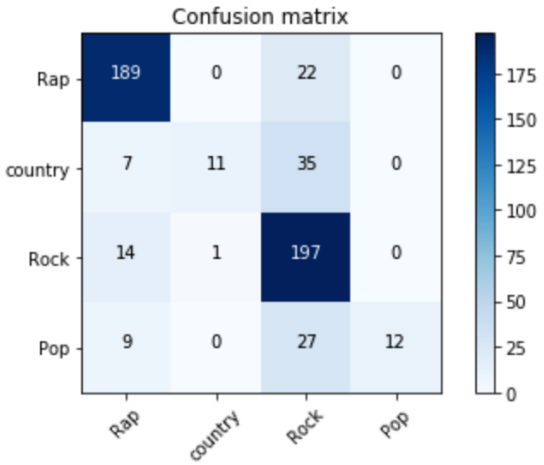
For SVM , I used CountVectorizer, TF-IDF, and for SGDClassifier all of the values are set to the defaults. I first checked the accuracy before using GridSearchCV to find the best parameters.

SVM

```
pipeline_svm = Pipeline([('vect', CountVectorizer()), ('tfidf', TfidfTransformer()),
                         ('svm', SGDClassifier(loss='hinge',
                                               alpha=1e-3,
                                               penalty='l2',
                                               max_iter=5,
                                               random_state=None))])

pipeline_svm = pipeline_svm.fit(X_train, y_train)
prediction_svm = pipeline_svm.predict(X_test)
np.mean(prediction_svm == y_test)
```

```
0.77480916030534353
```

The SVM on first test scored ~77.5% which was not too bad since it was not a binary classification having to distinguish between 4 classes. The next step was to improve this models accuracy by adding random over sampling to correct the imbalance in the classes.



As you can see from the confusion matrix above, there are many pop and country songs that were guessed to be other genres although all the pop songs that were actually pop were guessed correctly. I added random oversampling to have a better present of pop and country in the training set which may not improve the accuracy of this model but it should increase the precision.

```
pipeline_svm = Pipeline([('vect', CountVectorizer()), ('tfidf', TfidfTransformer()),
                         ('ros', RandomOverSampler(random_state=0)),
                          ('svm', SGDClassifier(loss='hinge',alpha=1e-3, penalty='l2', max_iter=5, random_state=None))])

pipeline_svm = pipeline_svm.fit(X_train, y_train)
prediction_svm = pipeline_svm.predict(X_test)
np.mean(prediction_svm == y_test)
```

```
0.80916030534351147
```

By just adding a ROS, the model improved to nearly 81%. The next step is to use grid search and try to find the best parameters.

```
parameters_svm = {'vect__ngram_range': [(1, 1), (1, 2)], 'tfidf__use_idf': (True, False),'svm__alpha': (1e-2, 1e-3)}
GridSearch = GridSearchCV(pipeline_svm, parameters_svm, n_jobs=-1)
GridSearch = GridSearch.fit(X_train, y_train)

prediction_svm = GridSearch.predict(X_test)
np.mean(prediction_svm == y_test)
```

```
0.85305343511450382
```

GridSearch allows me to create a range of parameters for which the model runs and returns the parameters that output the highest score. I found that the ngram range (1,2) which factors in collocated words highly increased the accuracy of this model by almost 5% to 85%.

The next model is Multinomial Naive Bayes using the same components, random oversampler, and GridSearchCV.

```
pipeline = Pipeline([('cv', CountVectorizer(stop_words='english')),
                     ('tfidf', TfidfTransformer()),
                     ('mnb', MultinomialNB(fit_prior=False)),
                    ])

parameters = {'cv__ngram_range': [(1, 1), (1, 2)], 'tfidf__use_idf': (True, False), 'mnb__alpha'
```

```
GridSearch = GridSearchCV(pipeline, parameters, n_jobs=-1)
GridSearch = GridSearch.fit(X_train, y_train)
```

```
GridSearch.best_score_
GridSearch.best_params_
results1 = GridSearch.predict(X_test)
np.mean(results1 == y_test)
```

```
0.85877862595419852
```

This pipeline uses the same parameters as the last and also uses GridSearch. This model was tested without random oversampler. The fit_prior parameter at false sets probability of classes to be uniform regardless of prior learning. This was also tested for “True” but there is no noticeable difference in accuracy.

```
pipeline = Pipeline([('cv', CountVectorizer(stop_words='english')),
                     ('tfidf', TfidfTransformer()),
                     ('ros', RandomOverSampler(random_state=0)),
                     ('mnb', MultinomialNB(fit_prior=False)),
                    ])

parameters = {'cv__ngram_range': [(1, 1), (1, 2)], 'tfidf__use_idf': (True, False),
              'mnb__alpha': (1e-2, 1e-3)}
GridSearch = GridSearchCV(pipeline, parameters, n_jobs=-1)
GridSearch = GridSearch.fit(X_train, y_train)

results2 = GridSearch.predict(X_test)
np.mean(results2 == y_test)
```

```
0.85687022900763354
```

The final pipeline used ROS with all of the same parameters. The result was surprising. In the SVM pipeline the ROS dramatically improved the model but it actually worsened the model for Naive Bayes.

Results

*ROS- RandomOverSampler	Support Vector Machine	Multinomial Naive Bayes

Accuracy Without ROS	80.92%	85.88%
Accuracy With ROS	85.31%	85.69%

In conclusion, the best model for predicting all 4 classes of genres is Multinomial Naive Bayes with RandomOverSampler. Although the accuracy is very slightly worse than without ROS, this prevents the data from guessing the much larger classes by chance and making it a more precise model for a more balanced test set.

References

What is doc2vec? -Quora Available:
https://www.quora.com/What-is-doc2vec
What is the learning rate in neural networks? -Quora Available:
https://www.quora.com/What-is-the-learning-rate-in-neural-networks
基于gensim的Doc2Vec简析 Available:
https://blog.csdn.net/lenbow/article/details/52120230
深度学习笔记——Word2vec和Doc2vec训练实例以及参数解读
Available:
https://blog.csdn.net/mpk_no1/article/details/72510655
手把手教你打造一个曲风分类机器人 Available:
http://baijiahao.baidu.com/s?id=1572252233830248&wfr=spider&for=pc
fxsjy/jieba: 结巴中文分词 Available:
https://github.com/fxsjy/jieba
Random Forests in Python:
https://community.alteryx.com/t5/Data-Science-Blog/Random-Forests-in-Python/ba-p/138437
Python 中的Scale 和Normalization(正则化):
https://blog.csdn.net/liluo9527/article/details/51028617

基于word2vec或doc2vec的情感分析：
http://datartisan.com/article/detail/48.html

sklearn-SGD:
http://blog.sina.com.cn/s/blog_7103b28a0102w08u.html

Sklearn api reference:
http://scikit-learn.org/stable/modules/classes.html

WordCloud Repetitive word fix:
https://stackoverflow.com/questions/43954114/python-wordcloud-repetitve-words

SVM Parameter tuning in SciKit Learn using GridSearchCV:
https://medium.com/@aneesha/svm-parameter-tuning-in-scikit-learn-using-gridsearchcv-2413c02125a0