Song Classification using Lyrics

Rachit Shah[*], Shreya Phadke[**]

*Computer Engineering Department, Institute of Technology Nirma University,
Ahmedabad 380009, India. Email: 14bce110@nirmauni.ac.in
**Computer Engineering Department, Institute of Technology Nirma University,
Ahmedabad 380009, India. Email: 14bce085@nirmauni.ac.in

Abstract— Text mining is frequently connected with processing of the English language to figure a general inference about a group of content. In this work, this idea is connected trying to use different models to categorize song's genre from its lyrics. Such models must have the capacity to analyze the words and classify songs as one of the six genres: rock, rap, country, Christian, R&B or pop. This work laid emphasis on classifiers like Multinomial Naive Bayes, Bernoulli Naive Bayes, Decision Trees, Random Forest, k-nearest-neighbor and K Means algorithms to classify every song. Similarly, it also talks about the importance of data collection and pre-processing keeping in mind the end goal to guarantee valid results.

Keywords— song, genre, classification, text, lyrics

I. INTRODUCTION

The lyrics of a song for the most part hold data about what that song is about. Performing text mining on lyrics of a song means to utilize data mining techniques to understand the meaning of the words in that song. In this work, we have used the results of data mining techniques to classify lyrics of a song into six genres: rock, rap, country, Christian, R&B or pop. This also denotes some assumptions: songs within same genre will have same lyrics; genre of a song is mutually exclusive. Sometimes it is true and sometimes not. For example: lyrics of rap and rock songs are different but the lyrics of rock and heavy metal songs are somewhat similar. Hence, we have assumed that the twelve genres do not overlap.

II. DATA COLLECTION AND PREPROCESSING

Data collection and preprocessing play a very important role in organizing the data for an input to the models.

A. Data Collection

The essential bit of information required in this work is the lyrics of various songs belonging to the twelve genres. These songs were acquired by using a python script which will query an online music database. For that we have imported BeautifulSoup in python. This database[1] gives tables containing artist, title, genre, and lyrics. The python script makes an association with the songlyrics.com website which contains the data above for every genre, extracts the table values from the HTML source, and generates a list of tuples of lyrics, genre and songinfo. Songlyrics.com website contains top 100 songs from rock, rap, country, Christian, R&B and pop genres. We have also used another dataset namely million subset database which contains approx. 22000 songs. Data is sent to a .csv file after going through all the links. It is also possible some data is not available at all. In such cases, we ignore those data and continue with the next tuple.

B. Preprocessing

Preprocessing[2] is a very important component of this project. The classifier models used will give accurate and consistent results if preprocessing is done correctly.

After reading the data, we will divide the songinfo into two factors: Artist and Song Title. Additionally there are a few songs which are recorded in more than one genre. We can manually choose the more fitting genre for each duplicate song however for simplicity we will ignore the second genre.

Steps to clean the corpus

- Make all words lowercase
- Remove all formatting
- Remove punctuation
- Remove any whitespace
- Drop stopwords
- Stemming
- Remove sparse terms

Stopwords are words which add no meaning to the context of a document (words like the, and, which, and so on).
Words having similar root are known as stemming words (words love, loves, loved, and loving are derived from same root word - love). We have used snowball and Porter stemmer algorithm (porter_tokenizer) for stemming. Snowball (snowball_tokenizer) is a string processing language used to create stemming algorithms in Information Retrieval.

Words which appear in more than 2% of the songs in the dataset are only included. Rest of the words are known as sparse words and are removed.

Next, we have used LabelEncoder. It encodes labels with values between 0 and n_classes-1. Here we have used 6 labels [0-5] for each of the six genres.

Then we used CountVectorizer. It converts the dataset into a matrix of token counts.

Parameters of CountVectorizer used:

encoding='utf-8'
encoding: string, ‘utf-8’ by default.
If bytes or files are given to analyze, this encoding is used to decode.

decode_error='replace'
decode_error: {‘strict’, ‘ignore’, ‘replace’}
Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given *encoding*. By default, it is ‘strict’, meaning that a UnicodeDecodeErr or will be raised. Other values are ‘ignore’ and ‘replace’.

strip_accents='unicode'
strip_accents: {‘ascii’, ‘unicode’, None}
Remove accents during the preprocessing step. ‘ascii’ is a fast method that only works on characters that have a direct ASCII mapping. ‘unicode’ is a slightly slower method that works on any characters. None (default) does nothing.

analyzer='word'
analyzer: string, {‘word’, ‘char’, ‘char_wb’} or callable
Whether the feature should be made of word or character n-grams. Option ‘char_wb’ creates character n-grams only from text inside word boundaries; n-grams at the edges of words are padded with space.
If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

tokenizer=porter_tokenizer
tokenizer: callable or None (default)
Override the string tokenization step while preserving the preprocessing and n-grams generation steps. Only applies if analyzer == 'word'.

ngram_range=(1,1)
ngram_range: tuple (min_n, max_n)
The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.

stop_words=stop_words
stop_words: string {‘english’}, list, or None (default)
If ‘english’, a built-in stop word list for English is used.
If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if analyzer == 'word'.
If None, no stop words will be used. max_df can be set to a value in the range [0.7, 1.0) to automatically detect and filter stop words based on intra corpus document frequency of terms.

binary=True
binary: boolean, default=False

If True, all non-zero counts are set to 1. This is useful for discrete probabilistic models that model binary events rather than integer counts. (Scikit-learn.org, 2017)

We also used TfidfVectorizer. It converts the dataset into matrix of Tf-Idf features. It is equivalent to CountVectorizer followed by TfidfTransformer.

Parameters of TfidfVectorizer used:

encoding='utf-8'
encoding: string, ‘utf-8’ by default.
If bytes or files are given to analyze, this encoding is used to decode.

decode_error='replace'
decode_error: {‘strict’, ‘ignore’, ‘replace’}
Instruction on what to do if a byte sequence is given to analyze that contains characters not of the given *encoding*. By default, it is ‘strict’, meaning that a UnicodeDecodeErr or will be raised. Other values are ‘ignore’ and ‘replace’.

strip_accents='unicode'
strip_accents: {‘ascii’, ‘unicode’, None}
Remove accents during the preprocessing step. ‘ascii’ is a fast method that only works on characters that have a direct ASCII mapping. ‘unicode’ is a slightly slower method that works on any characters. None (default) does nothing.

analyzer='word'
analyzer: string, {‘word’, ‘char’, ‘char_wb’} or callable
Whether the feature should be made of word or character n-grams.
If a callable is passed it is used to extract the sequence of features out of the raw, unprocessed input.

tokenizer=porter_tokenizer
tokenizer: callable or None (default)
Override the string tokenization step while preserving the preprocessing and n-grams generation steps. Only applies if analyzer == 'word'.

ngram_range=(1,1)
ngram_range: tuple (min_n, max_n)
The lower and upper boundary of the range of n-values for different n-grams to be extracted. All values of n such that min_n <= n <= max_n will be used.

stop_words=stop_words
stop_words: string {‘english’}, list, or None (default)
If ‘english’, a built-in stop word list for English is used.
If a list, that list is assumed to contain stop words, all of which will be removed from the resulting tokens. Only applies if analyzer == 'word'.
If None, no stop words will be used. max_df can be set to a value in the range [0.7, 1.0) to automatically detect and filter stop words based on intra corpus document frequency of terms.

binary=True
binary: boolean, default=False

If True, all non-zero counts are set to 1. This does not mean outputs will have only 0/1 values, only that the tf term in tf-idf is binary. (Set idf and normalization to False to get 0/1 outputs.) (Scikit-learn.org, 2017)

We conclude that by using Porter stemmer algorithm and CountVectorizer, we obtain a higher accuracy. Hence we have used only these methods in preprocessing.

III. EXPERIMENTS

For all experiments, a percentage split was used to separate training data to build models on and testing data to evaluate them. 60% of the data was used to train the model, and the remaining 40% to test it.

A. *Multinomial Naïve Bayes*

In multinomial naïve bayes, we first check the probability that the given lyrics belongs to which genre. Then, we check the conditional probability for the words in the lyrics provided that we know that the lyrics belong to a particular genre. The genre with the highest probability is finally the result. For our python code, we imported MultinomialNB from sklearn.naive_bayes.

Parameters used:
alpha=0.1
alpha: float, optional (default=1.0)
Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

fit_prior=False
fit_prior: boolean, optional (default=True)

Whether to learn class prior probabilities or not. If false, a uniform prior will be used. (Scikit-learn.org, 2017)

B. *Bernoulli Naïve Bayes*

Bernoulli Naïve Bayes is used if the features of your document are binary (zeros or ones). The only difference between Bernoulli and Multinomial is that the Bernoulli also takes into consideration the non-occuring words in the lyrics. Multinomial model ignores the non-occuring words. Apart from this, the rest of the method is same for Bernoulli and Multinomial model. For our python code, we imported BernoulliNB from sklearn.naive_bayes.

Parameters used:

alpha=0.001
alpha: float, optional (default=1.0)
Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).

fit_prior=False
fit_prior: boolean, optional (default=True)

Whether to learn class prior probabilities or not. If false, a uniform prior will be used. (Scikit-learn.org, 2017)

C. *Decision Trees*

Decision trees are easy to interpret and give clarity. Hence, they are very popular and used for classifying data. Decision tree works by placing the best attribute of the dataset at root of the tree. Then, split the training set into subsets. Repeat the above mentioned steps on each subset till there are leaf nodes in all the branches of the tree. But, decision trees might not generally be the best choice relying upon the kind of data and attribute set. This method is not advisable for very large datasets. For our python code, we imported DecisionTreeClassifier from sklearn.tree.

Parameters used:
criterion=\"gini\"
criterion: string, optional (default=”gini”)
The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain.

splitter=\"best\"
splitter: string, optional (default=”best”)
The strategy used to choose the split at each node. Supported strategies are “best” to choose the best split and “random” to choose the best random split.

random_state=0
random_state: int, RandomState instance or None, optional (default=None)
If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

min_samples_split=2
min_samples_split: int, float, optional (default=2)
The minimum number of samples required to split an internal node:
If int, then consider *min_samples_split* as the minimum number.

If float, then *min_samples_split* is a percentage and *ceil(min_samples_split * n_samples)* are the minimum number of samples for each split. (Scikit-learn.org, 2017)

D. *Random Forest*

In this method, a number of decision trees are generated randomly and are utilized in combination with each other to perform testing. For prediction, the test features of every randomly generated decision tree are used to predict results. Then, votes of each prediction are calculated and highest voted prediction is considered as final prediction. Hence, this method will generate a higher accuracy as compared to a single decision tree as more specific splits are carried out. However, this method has disadvantage of leading to overfitting data due to more number of splits. We imported RandomForestClassifier from sklearn.ensemble.

Parameters used:
max_depth=40
max_depth: integer or None, optional (default=None)

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. (Scikit-learn.org, 2017)

E. K-Nearest-Neighbor

The k- nearest-neighbor is also known as lazy learner because it only has to classify the k closest nodes surrounding it and choose to classify based on these k nodes. Here k denotes the number of neighbors to consider to classify the lyrics. We imported KNeighborsClassifier from sklearn.neighbors.

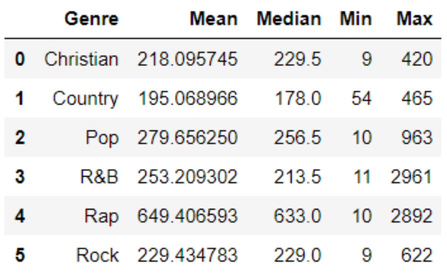
Parameters used:
n_neighbors=6
n_neighbors: int, optional (default = 5)

Number of neighbors to use by default for kneighbors queries. (Scikit-learn.org, 2017)

IV. RESULTS

A. Statistics for Songlyrics.com

Mean, median, minimum and maximum for 6 different genres.



	Genre	Mean	Median	Min	Max
0	Christian	218.095745	229.5	9	420
1	Country	195.068966	178.0	54	465
2	Pop	279.656250	256.5	10	963
3	R&B	253.209302	213.5	11	2961
4	Rap	649.406593	633.0	10	2892
5	Rock	229.434783	229.0	9	622

Figure1.
Mean, median, minimum and maximum (Songlyrics.com)

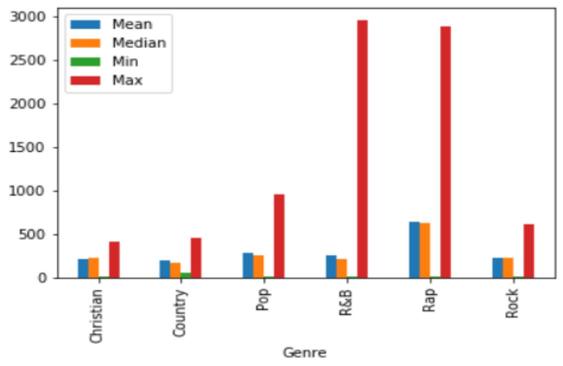


Figure2.
Mean, median, minimum and maximum (Songlyrics.com)

Wordcloud for:

1. Christian Genre

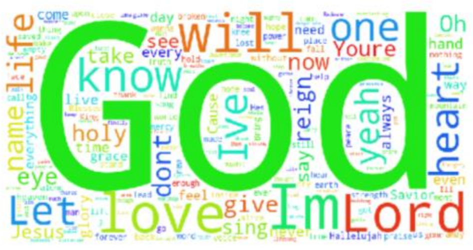


Figure3.
Wordcloud Christian genre

2. Country Genre

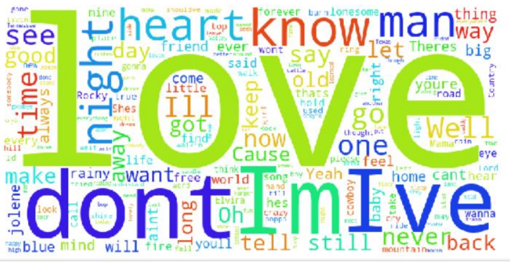


Figure4.
Wordcloud Country genre

3. Rap Genre

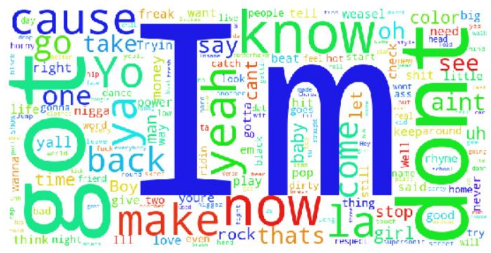


Figure5.
Wordcloud Rap genre

4. Pop Genre

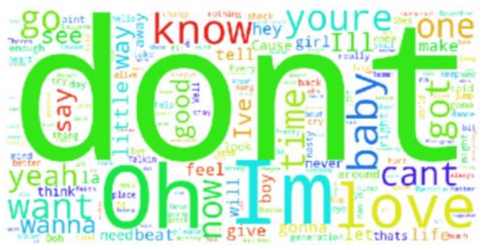


Figure6.
Wordcloud Pop genre

5. Rock Genre

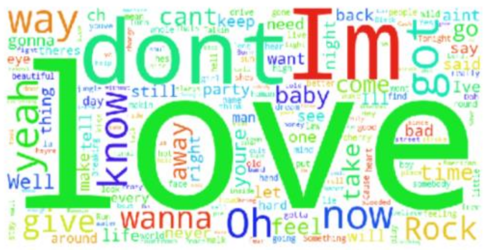


Figure7.
Wordcloud Rock genre

6. R&B Genre

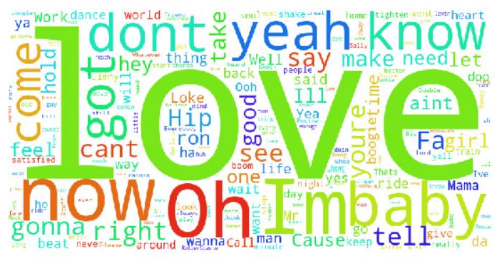


Figure8.
Wordcloud R&B genre

7. All Genres

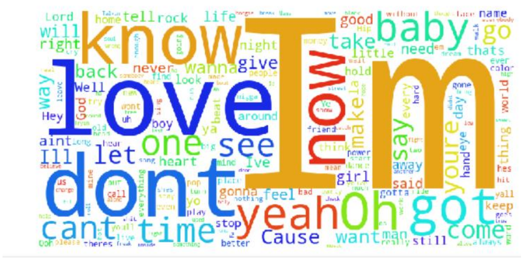


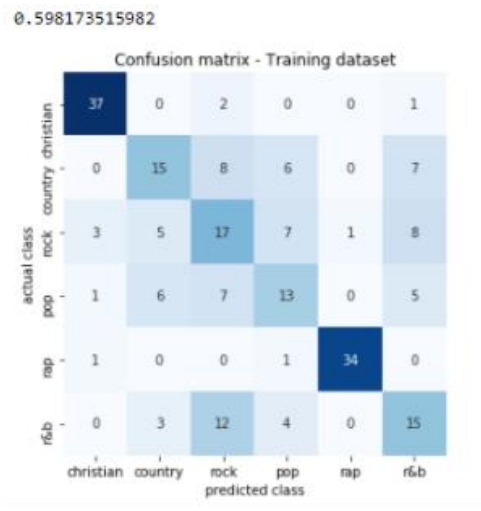
Figure9.
Wordcloud All genre

B. *Statistics for million subset dataset*

From million dataset, we took a subset of more than 22000 songs for genre classification but the accuracy was only 30%. This happened because the number of genres were large and hence the number of labels were large and the dataset was also large enough.

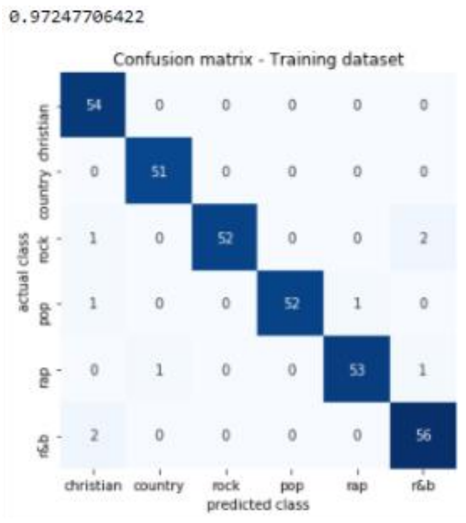
C. *Test and Train results for different models for songlyrics.com*

1. Bernoulli Naïve Bayes



```
0.598173515982
```

Figure10.
Bernoulli Naïve Bayes – Confusion matrix for test
Accuracy = 0.59



```
0.97247706422
```

Figure11.
Bernoulli Naïve Bayes - Confusion matrix for Train
Accuracy = 0.97

2. Multinomial Naïve Bayes

```
0.579908675799
```

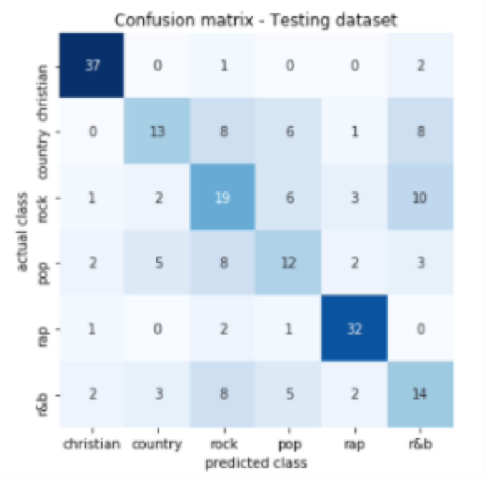


Figure12.
Multinomial Naïve Bayes - Confusion matrix for Test Accuracy = 0.57

```
0.981651376147
```

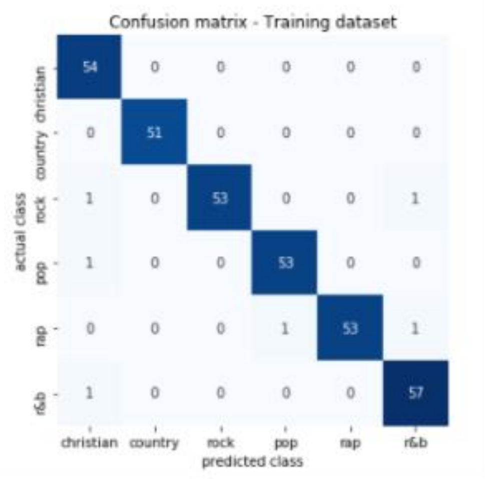


Figure13.
Multinomial Naïve Bayes - Confusion matrix for Train Accuracy = 0.98

3. Random Forest

```
0.506849315068
```

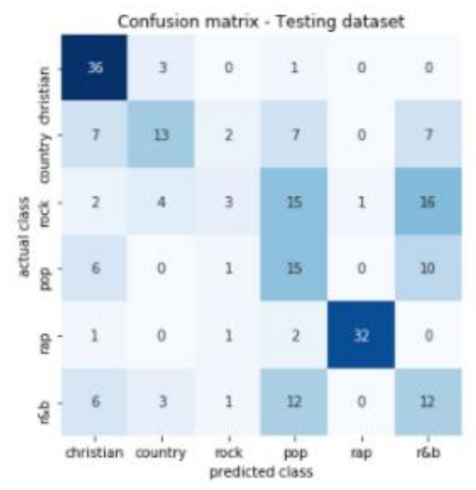


Figure14.
Random Forest - Confusion matrix for Test Accuracy = 0.50

```
1.0
```

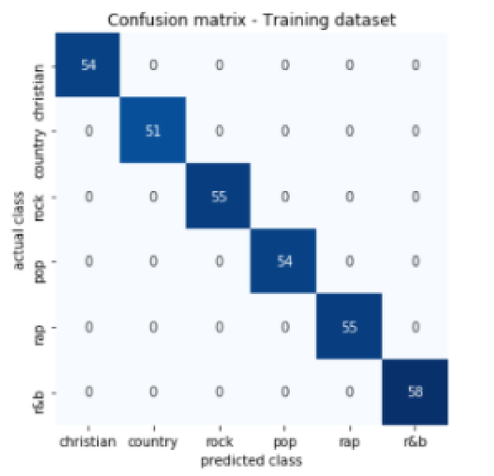


Figure15.
Random Forest - Confusion matrix for Train Accuracy = 1.0

4. Decision Trees

```
0.305936073059
```

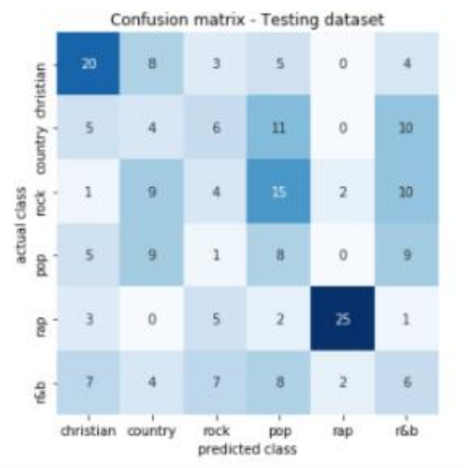


Figure16.
Decision Trees - Confusion matrix for Test Accuracy = 0.30

```
1.0
```

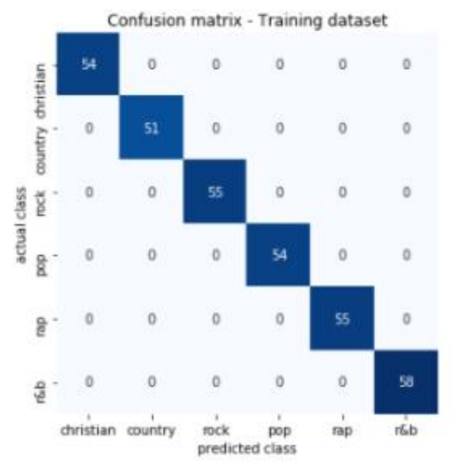
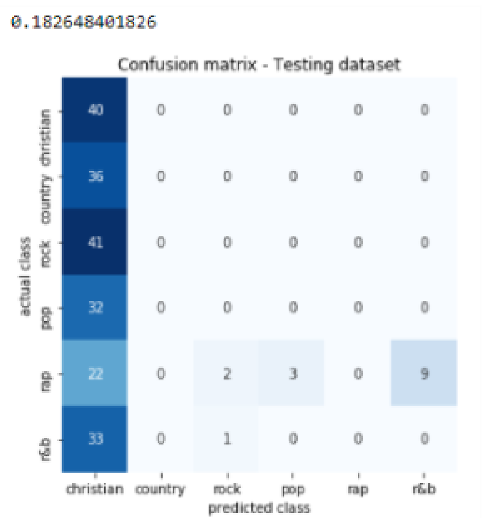


Figure17.

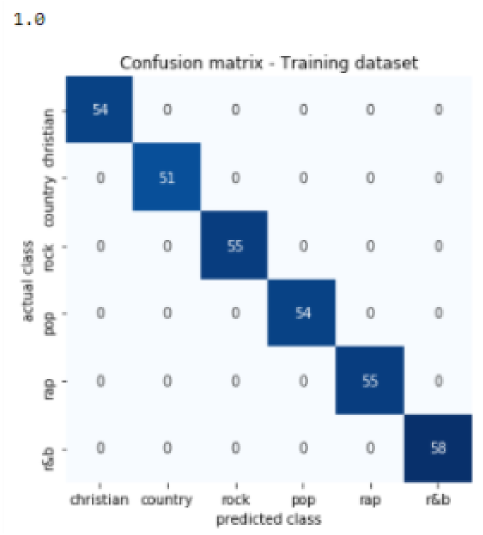
Decision Trees - Confusion matrix for Train Accuracy = 1.0

5. K-Nearest-Neighbor



```
0.182648401826
```

Figure18.
K-Nearest-Neighbor - Confusion matrix for Test Accuracy = 0.18



```
1.0
```

Figure19.
K-Nearest-Neighbor - Confusion matrix for Train Accuracy = 1.0

D. Results of million subset dataset

Accuracy obtained for different models:
Multinomial Naïve Bayes – 28%
Bernoulli Naïve Bayes – 31%
Decision Tree – 21%
Random Forest – 26%
Support Vector Machine (SVM) – 28%
Artificial Neural Network (ANN) – 17%

V. CONCLUSION

Any text mining experiment is the most dependent on the preprocessing stage. Hence, the relevance of the results obtained from the experiments is dependent on the successful completion of preprocessing stage. Hence, the results in the previous section could be improved.

In Singular Vector Machine Model, class weights are balanced. So if we have imbalanced classes i.e. one class contains more data than another, then it helps in SVM. In decision tree model, there is only one tree so the required estimators are less. But, random forest model compares multiple decision trees and considers their average result. Hence, accuracy of random forest model is more than that of decision tree model. Multinomial Naïve Bayes gives similar performance to Bernoulli Naïve Bayes. In conclusion, Naïve Bayes gives best results for our case.

VI. REFERENCES

[1] Xiao Hu, J. Stephen Downie, and Andreas F. Ehmann Lyric Text Mining in Music Mood Classification. American music, 2009.

[2] Dan Yangand Won Sook Lee Music Emotion Identification from Lyrics. Multimedia, 2009. ISM '09. 11th IEEE International Symposium on, San Diego, CA, 2009

[3]Scikit-learn.org. (2017). *sklearn.feature_extraction.text.CountVectorizer—scikit-learn 0.19.1 documentation*. [online] Available at: http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html [Accessed 14 Nov. 2017].

[4] Scikit-learn.org. (2017). *sklearn.naive_bayes.MultinomialNB-scikit-learn 0.19.1 documentation*. [online] Available at: http://scikitlearn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html [Accessed 14 Nov. 2017].

[5] Scikit-learn.org. (2017). *sklearn.tree.DecisionTreeClassifier-scikit-learn 0.19.1 documentation*. [online] Available at: http://scikitlearn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html [Accessed 14 Nov. 2017].

[6]Scikit-learn.org. (2017). *sklearn.neighbors.KNeighborsRegressor-scikit-learn 0.19.1 documentation*. [online] Available at: http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html [Accessed 14 Nov. 2017].

[7] Scikit-learn.org. (2017). *sklearn.naive_bayes.BernoulliNB — scikit-learn 0.19.1 documentation*. [online] Available at: http://scikitlearn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html [Accessed 14 Nov. 2017].

[8]Scikit-learn.org.(2017). *3.2.4.3.1. sklearn.ensemble.RandomForestClassifier-scikit-learn 0.19.1 documentation*. [online] Available at: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html [Accessed 14 Nov. 2017].

[9]Scikit-learn.org. (2017). *sklearn.feature_extraction.text.TfidfVectorizer — scikit-learn 0.19.1 documentation*. [online] Available at: http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html [Accessed 14 Nov. 2017].