MIDN Richard Jin, MIDN Jose Quiroz, MIDN Jaxon Harris
Dr. Chambers
SI425
Technical Writeup

Song Genre Classifier/Predicter

Introduction

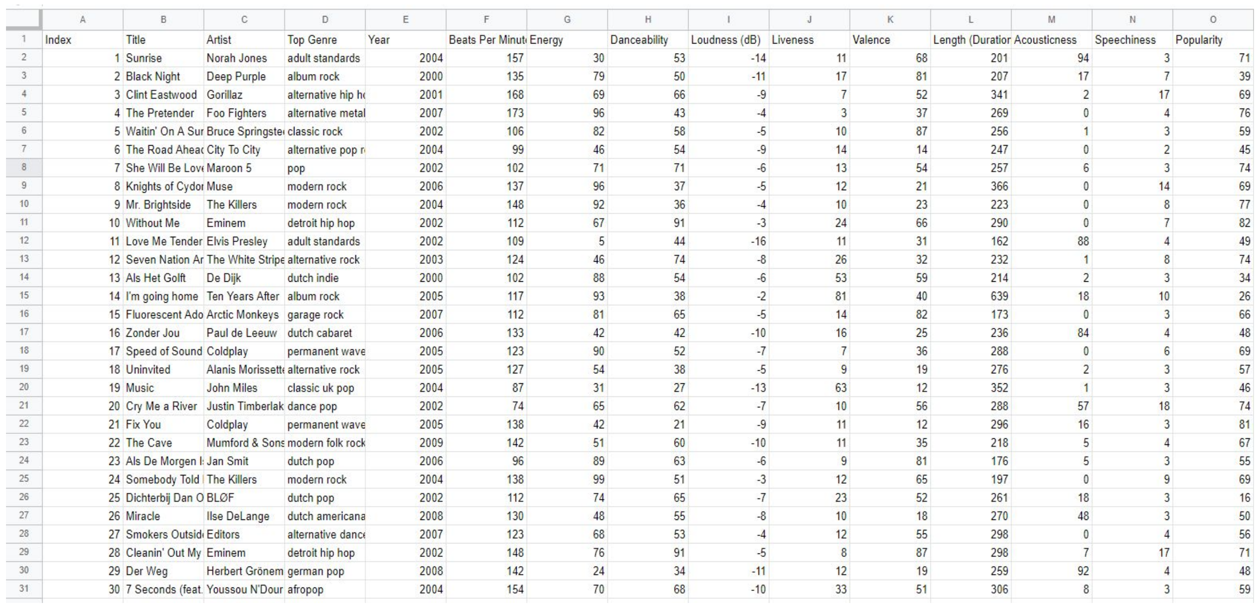
Throughout history, music has been a key in reflecting the society and social aptitude of people during a specific time period. However, as songs have progressed, evolved, and changed, there was a need to classify different songs and place them in different groupings. Eventually, the idea of a "genre" was created, where different types of music were established, and different songs were placed in these genres. Since these are groupings, the actual number of genres is based on how specific one wants to classify these genres. According to the popular music streaming service Spotify, there are over 1,300 music genres in the world. However, a lot of these genres have sub-genres, therefore the argument that there are only a handful of general genres applies here. For example, in the general genre "Country", there may be a lot of sub-genres such as alternative country, Australian country, Cajun, etc.

Our group's motivation stems from the fact that there are so many songs out there with different genres. On top of this, many songs could be a part of multiple genres as well. We wanted to create a classifier that determined the genre of a song by training the classifier with some data that shows what songs are what genres. This can be useful in determining different song genres. One catch with our initial classifier is that we only used song titles. This is important to note because song genre classifiers are usually with the song lyrics, holding a couple hundred words on a normal basis. However, song titles usually have around 2 or 3 words, with some a little more. The point is that we wanted to find a way to classify songs ONLY with the title of the song. This contingency provides a new challenge and right off the bat, we predicted a lower accuracy rate than with lyrics, just because of the actual amount of data given to the classifier.

Regarding the actual algorithm, there were a couple of different paths to go. For our classifier, we used the Logistic Regression Classifier as well as the Word2Vec Model. This paper will go more in depth with what each model includes, as well as the implementation of it for our song genre classifier.

Datasets Used

For the datasets used, we included a couple of different files. These CSV files (comma separated values) include the index for each song, song title, artist, genre, year it was created, and multiple sections that include the attributes for the song. The attributes include the BPM, intensity, etc., however these attributes were not utilized in this project. This could potentially be a project for the future in order to improve the algorithm, but due to the time constraints this was not done for this classifier. Attached below is a screenshot of the actual CSV file.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Index	Title	Artist	Top Genre	Year	Beats Per Minut	Energy	Danceability	Loudness (dB)	Liveness	Valence	Length (Duratior	Acousticness	Speechiness	Popularity
2	1	Sunrise	Norah Jones	adult standards	2004	157	30	53	-14	11	68	201	94	3	71
3	2	Black Night	Deep Purple	album rock	2000	135	79	50	-11	17	81	207	17	7	39
4	3	Clint Eastwood	Gorillaz	alternative hip ho	2001	168	69	66	-9	7	52	341	2	17	69
5	4	The Pretender	Foo Fighters	alternative metal	2007	173	96	43	-4	3	37	269	0	4	76
6	5	Waitin' On A Sur	Bruce Springste	classic rock	2002	106	82	58	-5	10	87	256	1	3	59
7	6	The Road Aheac	City To City	alternative pop r	2004	99	46	54	-9	14	14	247	0	2	45
8	7	She Will Be Love	Maroon 5	pop	2002	102	71	71	-6	13	54	257	6	3	74
9	8	Knights of Cydor	Muse	modern rock	2006	137	96	37	-5	12	21	366	0	14	69
10	9	Mr. Brightside	The Killers	modern rock	2004	148	92	36	-4	10	23	223	0	8	77
11	10	Without Me	Eminem	detroit hip hop	2002	112	67	91	-3	24	66	290	0	7	82
12	11	Love Me Tender	Elvis Presley	adult standards	2002	109	5	44	-16	11	31	162	88	4	49
13	12	Seven Nation Ar	The White Stripe	alternative rock	2003	124	46	74	-8	26	32	232	1	8	74
14	13	Als Het Golft	De Dijk	dutch indie	2000	102	88	54	-6	53	59	214	2	3	34
15	14	I'm going home	Ten Years After	album rock	2005	117	93	38	-2	81	40	639	18	10	26
16	15	Fluorescent Ado	Arctic Monkeys	garage rock	2007	112	81	65	-5	14	82	173	0	3	66
17	16	Zonder Jou	Paul de Leeuw	dutch cabaret	2006	133	42	42	-10	16	25	236	84	4	48
18	17	Speed of Sound	Coldplay	permanent wave	2005	123	90	52	-7	7	36	288	0	6	69
19	18	Uninvited	Alanis Morissette	alternative rock	2005	127	54	38	-5	9	19	276	2	3	57
20	19	Music	John Miles	classic uk pop	2004	87	31	27	-13	63	12	352	1	3	46
21	20	Cry Me a River	Justin Timberlak	dance pop	2002	74	65	62	-7	10	56	288	57	18	74
22	21	Fix You	Coldplay	permanent wave	2005	138	42	21	-9	11	12	296	16	3	81
23	22	The Cave	Mumford & Sons	modern folk rock	2009	142	51	60	-10	11	35	218	5	4	67
24	23	Als De Morgen I:	Jan Smit	dutch pop	2006	96	89	63	-6	9	81	176	5	3	55
25	24	Somebody Told I	The Killers	modern rock	2004	138	99	51	-3	12	65	197	0	9	69
26	25	Dichterbij Dan O	BLØF	dutch pop	2002	112	74	65	-7	23	52	261	18	3	16
27	26	Miracle	Ilse DeLange	dutch americana	2008	130	48	55	-8	10	18	270	48	3	50
28	27	Smokers Outside	Editors	alternative dance	2007	123	68	53	-4	12	55	298	0	4	56
29	28	Cleanin' Out My	Eminem	detroit hip hop	2002	148	76	91	-5	8	87	298	7	17	71
30	29	Der Weg	Herbert Grönem	german pop	2008	142	24	34	-11	12	19	259	92	4	48
31	30	7 Seconds (feat.	Youssou N'Dour	afropop	2004	154	70	68	-10	33	51	306	8	3	59

This is just one example of a CSV file with the songs included. There were multiple files used for this project. The dataset shown above had an entry of around 2000 songs, while the other files included a couple more thousand songs. One of the more successful CSV files that gave us the highest accuracy out of the ones we tested was a file of 10,000 songs. The song CSV files were retrieved from Kaggle, which included multiple files for different popularity of genres during different time periods.

Obtaining Information from the Dataset

This part of the project was a python programming language challenge. Python3 has a very good and easy way to deal with CSV files. With a couple lines of code shown below, we were able to populate a list of pairs, where each pair was a tuple of the song title and genre (essentially ignoring every other column of the song), and returning a list of song titles and its respective genre.

```
''' Given a corpus of song titles and their genres,
    this function returns a list of pairs, where each
    pair is (title, genre) tuple.
    fname: (str) name of file containing corpus
    Returns: a list of pairs (title, genre)
'''
def getDataset(fname):
    data =[()]
    with open(fname, 'r', encoding="ISO-8859-1") as read_obj :
        csv_reader = reader(read_obj)
        for row in csv_reader :
            found = False

            for genre in genres :
                if genre in row[3] :
                    data.append((row[1], genre))
                    found = True
                    break

            if not found :
                data.append((row[1], row[3]))

    # get rid of 'title' and 'genre'
    data.pop(0)
    return data
```

Once we were able to obtain the actual songs and the genres of the dataset, and was able to automate that process with every single file, we were able to continue and input these into the algorithms that we wanted to test this on.

Algorithm and/or Model and the Experiment

As mentioned before, we have incorporated two different models. One is training with a logistic regression classifier, and the other is training with a Word2Vec model. We will explain both in depth on what each is, as well as the implementation of both models regarding our project.

When talking about the logistic regression classifier, it is important to understand the general idea of what it actually is. There are weights that are incorporated into the classifier. You choose weights that give the "best results" or the "least error." Using gradient descent, you update the weights and continue to essentially "jiggle" the weights up and down based on each iteration of mistakes. In our model, we incorporated the logistic regression classifier from sklearn. Given the list of song titles and their known genre, we trained the logistic regression classifier. Y_train was the genre of the song, and X_train was the title of the song.

```
'''
Given a list of song titles and their known genre, train logistic regression classifier.
trainset: a list of pairs, where each pair is (title, genre) tupleReturns: void
'''
def trainWithLogit(trainset) :

    # Y_train -> List of genres for each song
    # X_train -> List of song titles
    Y_train = []
    X_train = []

    # fill lists #
    for pair in trainset :
        Y_train.append(pair[1])
        X_train.append( preProcess(pair[0]) )

    # Create Matrix #
    X_train_counts = vectorizer.fit_transform(X_train)

    # Train #
    logit.fit(X_train_counts, Y_train)
```

After training the classifier, we needed to test the logistic regression classifier. Using another song file list, we filled X_test with song titles, and for each song title, the classifier predicted teh genre based on the data given beforehand. The actual python code returned a list of song genre guesses. Attached below demonstrates the actual syntax for this operation.

```
'''
Given a list of song titles and their genres, predict the
genre for each song title.
testset: a List of pairs, where each pair is (title, genre) tuple
Returns: a list of genre names, which are the classifier's prediction for each given song title.
'''
def testLogit(testset):

    # X_test -> list of song titles #
    X_test = []

    # Fill List #
    for pair in testset:
        X_test.append( preProcess(pair[0]) )

    # Transform Unseen Text #
    X_test_counts = vectorizer.transform(X_test)

    # Get Predictions #
    guesses = logit.predict(X_test_counts)
    return guesses
```

Once the testing for the logistic regression classifier was completed, and we received a result, we moved onto our Word2Vec model. Obviously, before diving into the application of Word2Vec in our specific project, we must discuss what Word2Vec is in a very general sense. As mentioned before in the class, we wanted to learn word embeddings, or vectors, by predicting neighboring words. Originally, words were created with a random vector. After giving the classifier data, each word would "predict" its neighbor and provide scaling (jiggling of the weights) to the appropriate word. Once this is finished, you would compute the probabilities by scoring all of the words. Similar words will have similar values, while words that do not associate often will not have similar values. However, it is interesting to note that antonyms are closely scored in this situation because of the context, NOT the actual meaning of the word.

For our specific Word2Vec Model, we created a list of token numbers, and filled the list with song titles. We created and trained the model by having each title getting its vector with representation with the vector scores. For testing, we looped over every song title. We turned the title into a word vector, and used the Cosine function to compute its score.

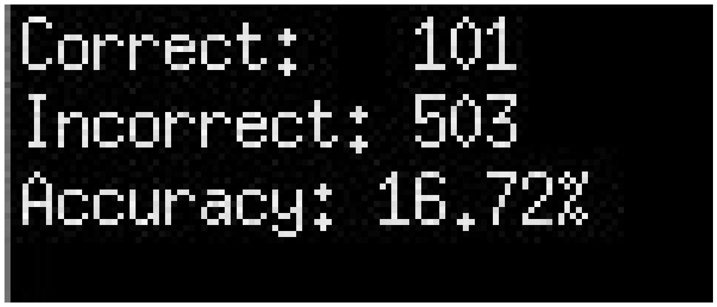
```
'''
Computes a Score for each song title in testset
and then returns a list of scores for each testset!
'''
def testVectors(vTitles, testset):

    scores = []
    # Loop over every song title in testset #
    for title in testset :
        title_score = []
        # Add up every Word in Title and Make it a Vector #
        vTitle = get_sentence_vector((preProcess(pair[0])).split(" "))
        # Compute Score using Cosine function using every title
        # in trainingset
        for index, vector in enumerate(vTitles):
            score = cosine(vTitle, vector)

            if not nmpy.isnan(score):
                title_score.append((score, index))
            else :
                title_score.append((0, index))
        scores.append(title_score)
```

There are a handful of different ways to compute the score, however for the purposes of this project we incorporated the cosine function.

Final Approach and Results

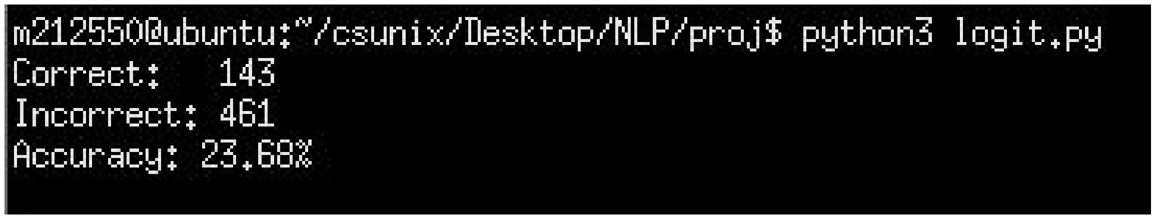
As with any testing and training, it is important to gauge a sense as to how successful the classifier was, and figure out the reasons why the accuracy rate was not high or why it was high. This would also give us an indication to understand if our original hypothesis was valid or not. We were curious about classifying song TITLES instead of just lyrics, as it is substantially less data, therefore we were expecting lower accuracy results. The intent was to see relatively how the scores were.

Our Word2Vec results are as follows:



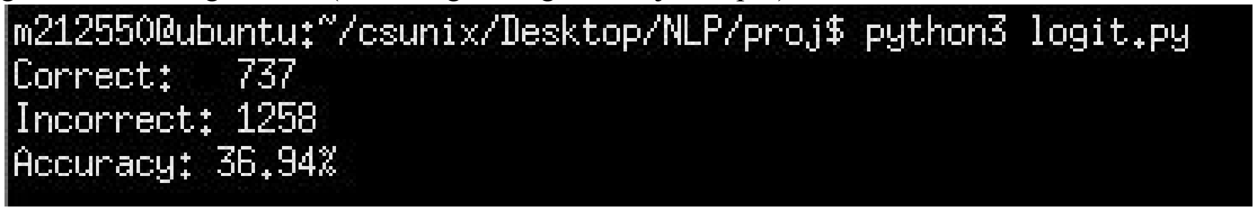
```
Correct:   101
Incorrect: 503
Accuracy: 16.72%
```

Our original logistic regression classifier results are as follows:



```
m212550@ubuntu:~/csunix/Desktop/NLP/proj$ python3 logit.py
Correct:   143
Incorrect: 461
Accuracy: 23.68%
```

After looking at these results, which are obviously quite low, we were curious to understand if the algorithm was incorrect, or the dataset was not diverse, or whatever the reason was for the low accuracy. Using this information, we decided to take another random music list (this one with 10,000 songs instead of the 2000 previously used), and got the following results. (No change in algorithm, just input).



```
m212550@ubuntu:~/csunix/Desktop/NLP/proj$ python3 logit.py
Correct:   737
Incorrect: 1258
Accuracy: 36.94%
```

This new dataset was a much larger dataset, and was a lot more diversified as well, which may have played a role in the significant improvement relatively. We will discuss the implications of this, as well as potential variants of algorithm failures along with result failures.

Variants of Algorithm Failures (and result failures)

Obviously, when working on a project, the ultimate goal is to compute the highest probability and have the highest accuracy. After all, a project usually does not want to fail! So why were our scores so low? Why were the accuracy rates, at BEST, 36%?

From the beginning of the project, we hypothesized that the accuracy rate of this project would be lower than if full song lyrics were used. Intuitively this makes sense, as song lyrics usually have a couple hundred words per song. Factor this in with a couple tens of thousands of songs, this gives a lot of training data, hence optimizing the classifier. For the purposes of our project, we only decided to use song titles. Song titles usually have around 2-3 words, sometimes a sentence, sometimes only one word. This decrease in training data would affect the classifier, hence the lower rate of which it is successful.

When we were testing the algorithm, our initial hypothesis was that the algorithm was not optimized correctly. After spending a lot of time actually trying to fix and figure out what the bug in our program was, we ended up inputting new data with a higher number of songs. This (as shown above) significantly increased the accuracy, from 23% to 37%. Without altering the data, it became optimized significantly.

Usually, with lyric analysis, the accuracy rate is around 70%. The fact that our classifier produced roughly 37% accuracy rate with only a handful of words, as opposed to the entire lyrics, demonstrates that the algorithm is correct and fine. Obviously there is more to optimize, but for the purposes of this project, the analysis shows that song titles CAN be used, however not as reliable as lyrics would be.

Another possible problem with this project is the actual dataset we were using. A lot of the songs we have been using for this project are songs that are popular, especially as of recently. Pop, rock and hip hop have been more prevalent, so these genres were in the training data a lot more. Hence, a lot of our errors were the classifier guessing hip hop and pop, or implementing rock as well. Understanding what the data is composed of is important to understanding why we receive the results we get, and it is important not to assume what we are given as a diverse and smoothed dataset.

Final Approach and Implementation

To walkthrough and better explain our final approach and to run the programs too, please clone our project from our team's github repository. The command is given below:

git clone https://github.com/jequiroz99/NLP-Project-2021

One of the first things that our team was determined to find is how well we could predict a song's genre using the song's lyrics instead of just the words in its title. As you would have guessed, the accuracy rate for predicting if the song's genre is rock, pop, or hip hop is much higher than the one we were able to produce using only the song's title. The hardest thing for us to do was to find a dataset of songs lyrics and their genre. Initially, we found a file with around

77.9k song names with a special ID for each song. We discovered that MusixMatch API maps songs and their lyrics and genres to those unique identifiers. We made scripts that made calls to the API requesting the song's lyrics and genre by providing their unique ID (and we had 77,900 of them). However, we immediately found issues such as the API not wanting to provide the song's lyrics completely (because of copyright restrictions). Also, we could only make 2,000 API calls daily, but still that would be useless since the API doesn't allow us to get the complete lyrics of the song.

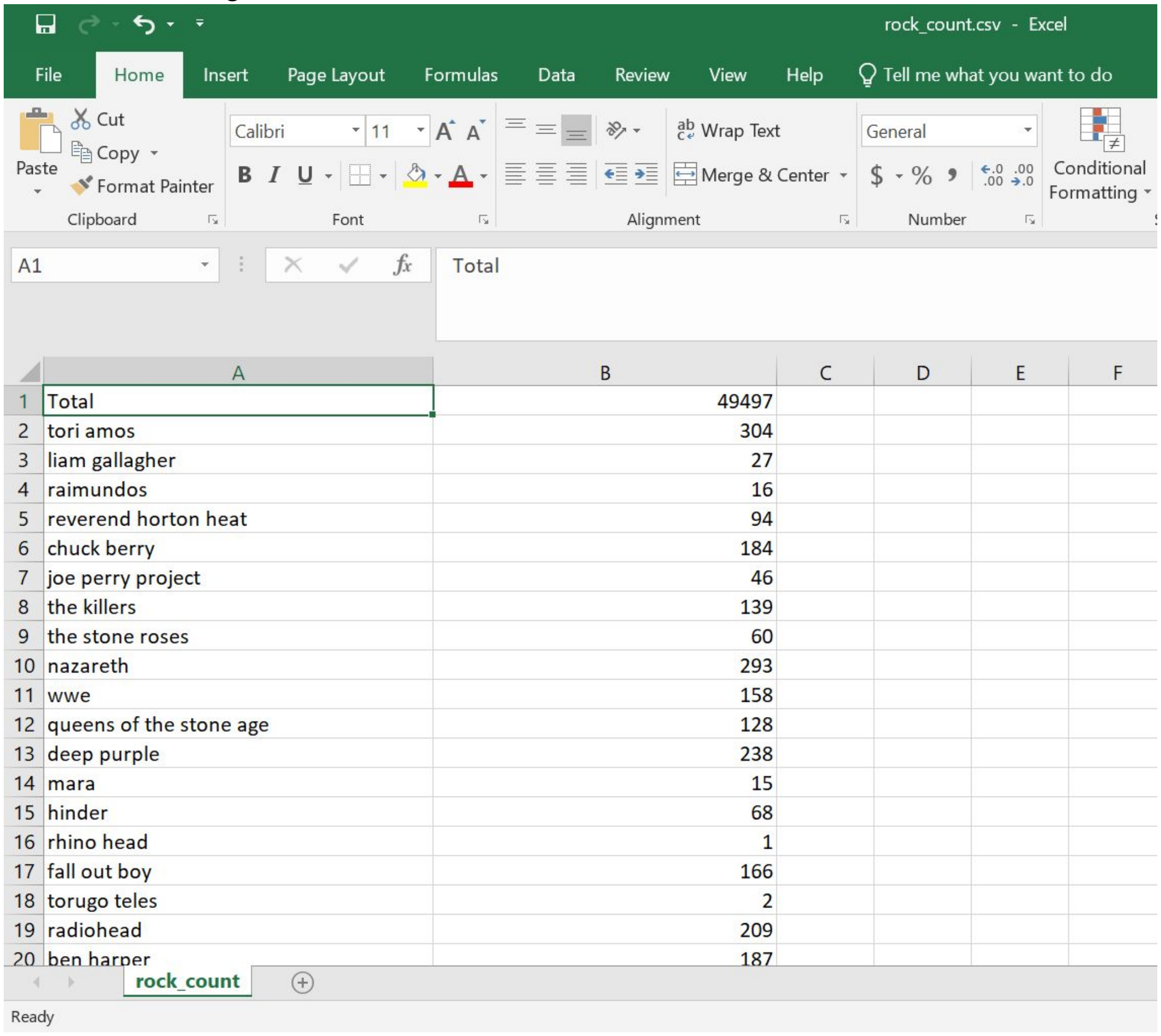
Our new Dataset

As we continued to look for more and more datasets, we finally found a big dataset containing exactly what we wanted, and even more! The original dataset that we found is in the main directory, under the folder *Original Dataset*. We found 2 big csv files. The first one, called *artists-data.csv* contains about 3000 artists' names and their main genre. This file included a couple of Brasilian and Portguese artists that made songs in spanish or portuguese. The second file, the most interesting one, is called *lyrics-data.csv* and contains song lyrics for almost all of the artists featured in the *artists-data.csv* file. After a couple of minutes of looking at the dataset, we found out that it features 6 different genres: rock, pop, hip hop, samba, funk carioca, and sertanejo. The last 3 of these genres were purely portuguese genres, so we were not interested in those. Also, we found out that not all artists in *artists-data.csv* are featured in *lyrics-data.csv*, and vice-versa. Additionally, some rock, pop, and hip-hop songs contained lyrics in spanish, which is also something that we do not want in our dataset. Our team also thought about many different ideas we could experiment with using this huge dataset, such as training on the lyrics of specific artists and seeing which one produces the best accuracy, we also wanted to see if creating embeddings of each artist is better than creating embeddings of each song or of each genre, etc. To do this, however, our team reached the consensus that it would be better to organize our dataset better so that we don't have to be constantly checking for english lyrics only, or for certain genres, etc.

Organizing the Dataset

In the *code/scripts* directory of the project repository, we documented most of the scripts that we used to actually organize and get the information we wanted from this big dataset. We started by using *code/scripts/names.py* to extract the artists that both appear in *lyrics-data.csv* and *artists-data.csv*, and that belong to either rock, pop, or hip hop genre. The script then creates a new file called *artists.csv* that contains the name of all of the artists extracted above, as well as their main genre, which can only be rock, pop, or hip hop.

Then, we used the script *code/scripts/fileMaker.py* to create three folders, one for each genre, under the *data/* directory. Then, the script would loop over all artists in *artists.csv*, create a file using the artist's name, and place the file on *data/rock/* if the artist's main genre is rock, or on *data/pop/* if the artist's main genre is pop, etc. Finally, each artist's file contains 2 columns, one for each song's name and the other one for the corresponding song's lyrics.Since our data is now organized and contains the information we want, and we used the script *code/scripts/songCounter.py* to find out the number of songs on each genre, and also the number of songs each artist has. For instance, the *counts/rock_count.csv* show the total number of rock songs in the dataset, and also how many songs each artist has. The *songCounter.py* creates this information for all genres.



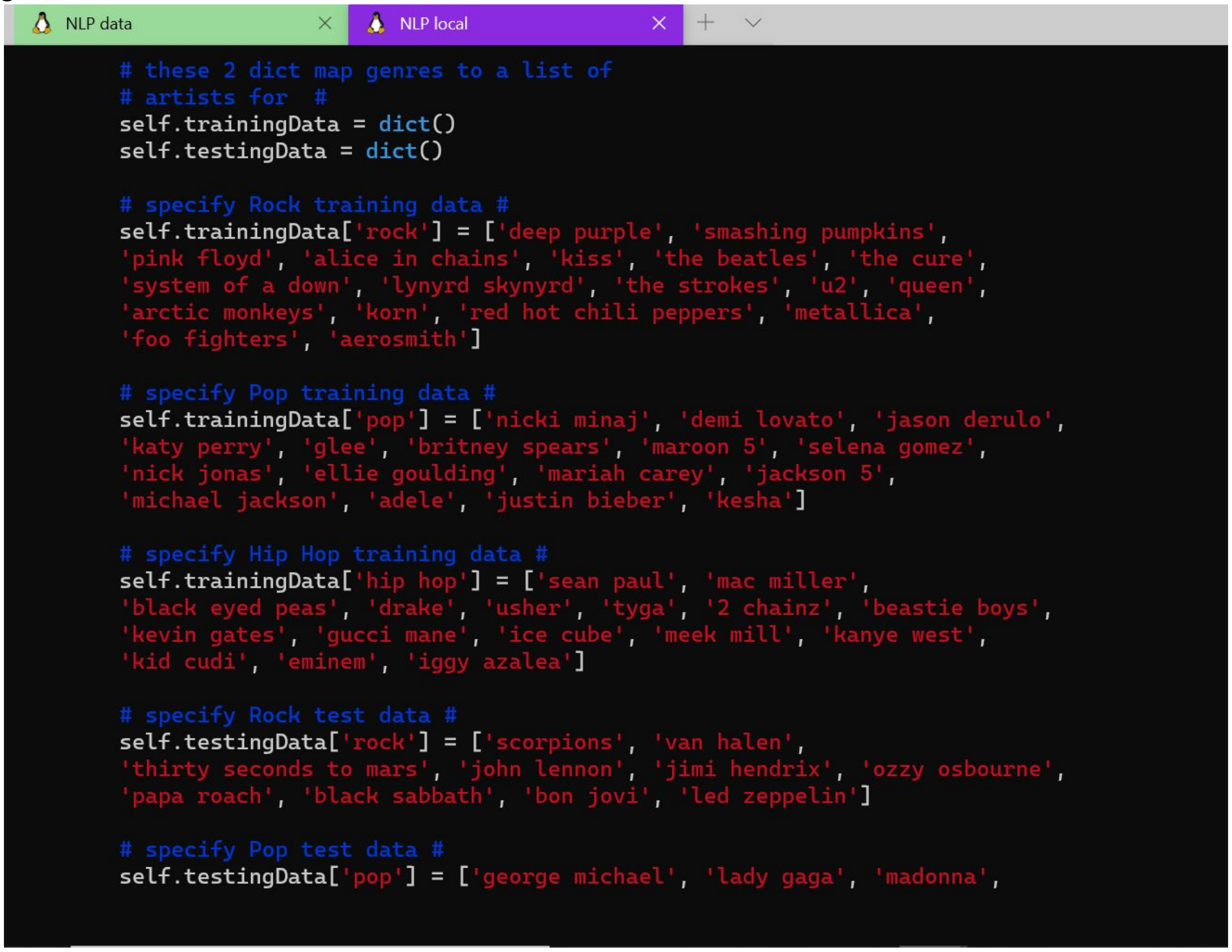
	A	B
1	Total	49497
2	tori amos	304
3	liam gallagher	27
4	raimundos	16
5	reverend horton heat	94
6	chuck berry	184
7	joe perry project	46
8	the killers	139
9	the stone roses	60
10	nazareth	293
11	wwe	158
12	queens of the stone age	128
13	deep purple	238
14	mara	15
15	hinder	68
16	rhino head	1
17	fall out boy	166
18	torugo teles	2
19	radiohead	209
20	ben harper	187

For situational awareness, we have 49,497 rock songs, 32,021 pop songs, and 14,888 hip hop songs. To compensate for the small amount of hip hop songs available, we used approximately the same amount of songs for each genre in both training and testing.

Logistic Regression Classifier

Our goal is to create a classifier that trains on song lyrics and their genres, so that it is able to predict a song's genre given its lyrics.

Part 1. Data Extraction. The first part of our algorithm is data extraction. We have 2 files: *code/LogisticRegression/Main/data.py* and *code/LogisticRegression/Main/shortData.py*. For all of our classifiers, we will always have a *data.py* and *shortData.py* file that is used to extract the training data set and testing data set from our *data/* directory that we created, as well as evaluate the accuracy of our model. The main difference between both files is that *data.py* extracts about 4500 songs from each genre to create the training dataset, while *shortData.py* uses less, (about 2500 songs per genre for training). The way that both of these files work is that we first hardcode the name of the artists that we want to use for both training and testing for each genre.

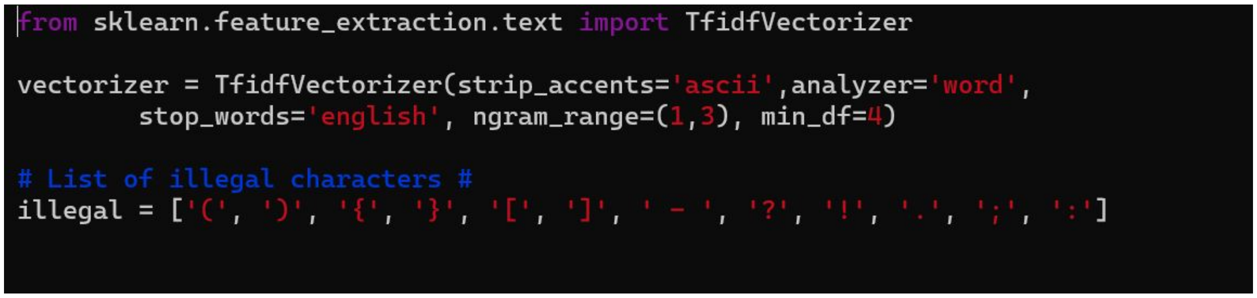


```
        # these 2 dict map genres to a list of
        # artists for  #
        self.trainingData = dict()
        self.testingData = dict()

        # specify Rock training data #
        self.trainingData['rock'] = ['deep purple', 'smashing pumpkins',
        'pink floyd', 'alice in chains', 'kiss', 'the beatles', 'the cure',
        'system of a down', 'lynyrd skynyrd', 'the strokes', 'u2', 'queen',
        'arctic monkeys', 'korn', 'red hot chili peppers', 'metallica',
        'foo fighters', 'aerosmith']

        # specify Pop training data #
        self.trainingData['pop'] = ['nicki minaj', 'demi lovato', 'jason derulo',
        'katy perry', 'glee', 'britney spears', 'maroon 5', 'selena gomez',
        'nick jonas', 'ellie goulding', 'mariah carey', 'jackson 5',
        'michael jackson', 'adele', 'justin bieber', 'kesha']

        # specify Hip Hop training data #
        self.trainingData['hip hop'] = ['sean paul', 'mac miller',
        'black eyed peas', 'drake', 'usher', 'tyga', '2 chainz', 'beastie boys',
        'kevin gates', 'gucci mane', 'ice cube', 'meek mill', 'kanye west',
        'kid cudi', 'eminem', 'iggy azalea']

        # specify Rock test data #
        self.testingData['rock'] = ['scorpions', 'van halen',
        'thirty seconds to mars', 'john lennon', 'jimi hendrix', 'ozzy osbourne',
        'papa roach', 'black sabbath', 'bon jovi', 'led zeppelin']

        # specify Pop test data #
        self.testingData['pop'] = ['george michael', 'lady gaga', 'madonna',
```

The purpose of hardcoding the artists' name for each genre is that we are trying to optimize performance, so we are trying to choose artists from all different backgrounds and styles, so that we encompass every aspect of the genre. For example, in the 'rock' genre we have different backgrounds, such as heavy metal, metal, grunge, alternative, pop rock, etc. So, that is why we decided to include a wide diversity of artists for the rock genre. Note that the number of songs for each genre sums up to about 4500 songs in *data.py* and 2500 for *shortData.py*.

Another interesting thing that we did for the testing and training dataset for hip hop is that in *data.py* we divide the number of Chris Brown songs in 2 parts, half goes to training and half to testing. Chris Brown's music is very diverse, so we thought that partitioning his songs would be the best, since we have 1174 songs from him.

Part 2. Preprocessing Lyrics. The next step is pre-processing the song's lyrics before training. Initially, we did basic pre-processing by replacing stop words such as commas, periods, question and exclamation marks, hyphens, parenthesis, brackets, etc. Then, we passed the lyrics to the ***sklearn Count Vectorizer*** from the ***scikit learn toolkit*** library in python for training the classifier. Through experimentation and by reading the scikit learn user guide, we found that there are other text feature extraction models that are faster and that do a better job, at least for song lyrics, than the Count Vectorizer we used for our labs. The one that gives us the best results is the **TfidfVectorizer**. Here are the parameters that we used that gave us the best results in terms of accuracy and speed. **Note**: we attempted to also change the parameters for the Count Vectorizer, and performance increased by doing that! However, the best results were obtained from using the **TfidfVectorizer** text feature extraction model.



```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(strip_accents='ascii',analyzer='word',
        stop_words='english', ngram_range=(1,3), min_df=4)

# List of illegal characters #
illegal = ['(', ')', '{', '}', '[', ']', ' - ', '?', '!', '.', ';', ':']
```

The best performance is achieved by using trigrams, ignoring terms with a document frequency less than 4, performing character normalization using ‘ascii’ characters (that’s what strip_accents does) and by using the english stop words that the library provides.

In accordance with the Scikit Learn User Guide, this TfidfVectorizer is the combination of using the CountVectorizer model for tokenization and counting occurrences, and then using the TfidfTransformer model to normalize the frequencies in the sparse matrix. The TfidfVectorizer combines both of those into one.

Part 3. Training a Model. The linear model that gave us the best performance on both accuracy and speed is the Logistic regression classifier model using the solver ‘saga’. The configurations for the model are the following :

from sklearn.linear_model import LogisticRegression
model = LogisticRegression(solver=“saga”, max_iter=100)

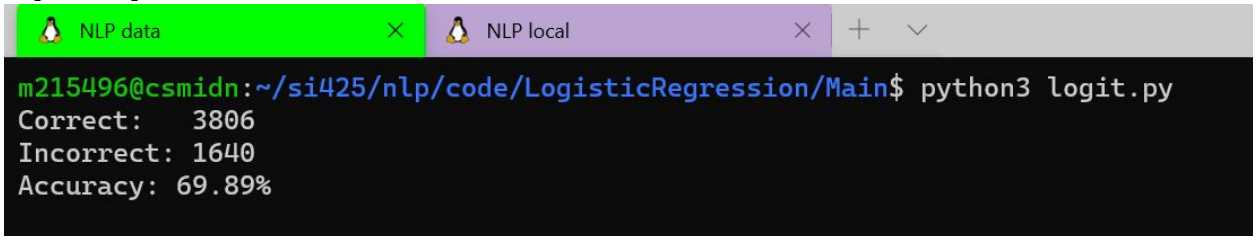
The saga solver is faster than the other models when using large data, which is essentially what we are doing.

Part 4. Testing and Error Analysis. Our Logistic regression classifier had an accuracy of **69.98%** using *data.py* to create the dataset (remember that *data.py* provides a larger dataset) and an accuracy of **63.58%** with *shortData.py*. To evaluate the accuracy, we use the same method of dividing the number of right predictions by the total number of predictions.

Error Analysis: A disadvantage that our dataset has is that it does not give us the genre of every single song. Instead, it gives us the genre of the artist that made the song. This is a problem because, when we inspect *wrong.txt* which tells us which songs we predicted incorrectly, we see that for artists like ‘kesha’, ‘rihanna’, and ‘britney spears’, not all of their songs are pop songs, instead, some of them are actually hip hop, which brings a big issue for us.

Something else that we noticed is that some artists like 'bon jovi', 'george michaels', and 'john lennon', are artists that have made a lot of rock and pop songs in their career, so it is hard for us to know which one of their hundreds of songs are rocks and which ones are pop.

Initially, our accuracy was in 68.3% with the *data.py* dataset. Something that we did was just add more songs from more artists to the training dataset, as well as using artists with a couple of hundred of songs in the test dataset instead of artists with less than 100 songs. That improved performance too!



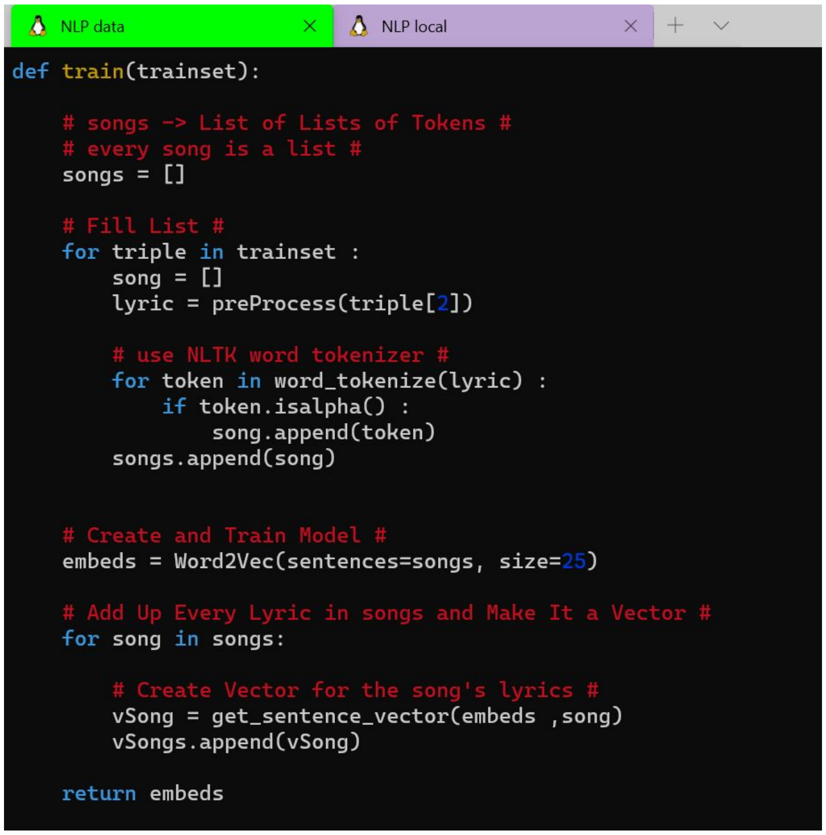
```
m215496@csmidn:~/si425/nlp/code/LogisticRegression/Main$ python3 logit.py
Correct:    3806
Incorrect:  1640
Accuracy:  69.89%
```

Word2Vec Model with Song Lyrics

Overview. Since our team counts with a big corpus of song lyrics and their artists, we decided to make a word2vec model for predicting song genres and some other interesting things too. These implementations are more for experimentation purposes. The Dataset extraction methods that we used in our Logistic Regression classifier is the same for this model.

Pre Processing. For preprocessing on all word2vec models, we used the **word_tokenize** class from the **nltk.tokenize** library in order to split a string of sentences (lyrics, in our case) into an array of tokens. Then, we looped over every token and if the token consisted of just letters, we stored it. Otherwise, we got rid of that token. This tokenization tool proved to work well for separating question marks, periods, brackets, parenthesis, etc. from words.

Training. For training, our model creates a vector for every entry in the dataset. For the *code/Word2Vec/Song2Vec/model.py* model, every entry is a song, so it makes a vector for every song (not that efficient, by the way). For our *code/Word2Vec/Artist2Vec/model.py* model, every entry is all of the songs for a single artist, so essentially we will make a vector for every artist! Lastly, as you may guess, for our *code/Word2Vec/Genre2Vec/model.py* model, every entry is all of the songs for a single genre. Notice that we save each vector for later use when we test!



```python
def train(trainset):

    # songs -> List of Lists of Tokens #
    # every song is a list #
    songs = []

    # Fill List #
    for triple in trainset :
        song = []
        lyric = preProcess(triple[2])

        # use NLTK word tokenizer #
        for token in word_tokenize(lyric) :
            if token.isalpha() :
                song.append(token)
        songs.append(song)


    # Create and Train Model #
    embeds = Word2Vec(sentences=songs, size=25)

    # Add Up Every Lyric in songs and Make It a Vector #
    for song in songs:

        # Create Vector for the song's lyrics #
        vSong = get_sentence_vector(embeds ,song)
        vSongs.append(vSong)

    return embeds
```

Testing. The testing algorithm is almost the same for all models. For every entry in the testing set, we will make a vector. Then for that vector, we will compute a score using the cosine function between that vector and all vectors that we created and stored in training! The genre of the training vector with the highest score is the predicted genre.

Models: For the model in *code/Word2Vec/**Song2Vec/model.py***, it makes a vector for every song in the dataset. This is a lot of vectors, so it really isn't that accurate or efficient. For the model in *code/Word2Vec/**Artist2Vec/model.py***, every dataset entry is the all of the songs of a single artist. Therefore, this model makes a vector for all artists. Its accuracy is around 90% for predicting the genre! Lastly, *code/Word2Vec/**Genre2Vec/model.py*** makes a vector for all songs of a single genre. Its accuracy is 100%!

More Experimentation

In *code/LogisticRegression/*, the *Main/* directory is for the model that performed the best, which is the logistic regression classifier. All other folders are for other Linear Regression models that our group made.

Future Work

We would have liked to make an RNN model to predict song genres too. We have a large enough corpus to feed the RNN a lot of data and probably get better predictions than the ones we got.

Our Artist2Vec implementation can have other uses, such as predicting the most similar artist for a given artist, which is something we wanted to implement but we could not because of time constraints. We could even make a lyric generator that uses the lyrics of the most similar artist of the given artist.