Summary

The days in which what we do in our day to day lives is only knowledge to ourselves is over. But what if music recommendation could be based solely on the music? Not on its popularity, or if your friends were listening to it, or if an album was just released, or there was a concert for an artist in your city. For this project, I wanted to build a recommendation system solely based on the music features, without any social input. This allows for a wider selection of similar songs returned, not limited to other songs by the same artists or limited by only popular songs. A user would be able to enter a song and get recommendations of music that they might have not listened to before. This algorithm uses audio features to return 5 recommended songs based on the user's input. Although there is plenty of room for improvement, this algorithm was successfully implemented into a portion of the Million Song Dataset and should be able to be implemented on any large dataset of songs.

The Problem

Recommendation systems are usually built upon a social factor. "Other users purchased" or "others listened to" seem ubiquitous. A lot of music recommendation algorithms rely primarily on a social aspect or some sort of user interactivity or history to determine other songs that we might be interested in. This works to a degree as some of us are likely to check out albums that our friends recommend to us. Music tastes can change though, and a song that may have been a favorite a few months ago may no longer be. Since these tastes are both fluctuating and unique, I believe that it may be best to utilize a recommendation system based solely on audio features, allowing a playlist to be curated on just the audio similarities from one song to the next, allowing for a solo listening experience that can change as often as we would like.

The Client

The ultimate client would be any listener of music. Currently many people's listening habits are based on others without even realizing, the most popular songs that are chart toppers are due to the fact that everyone is listening to them, leaving some quality music that may not be as well-known entirely un-listened to. By implementing this type of recommendation system, it benefits both the consumer and the music maker, by allowing listeners to experience music that they otherwise would not have, and the music maker reaches a wider audience. More immediately, music streaming and library services such as Spotify, Pandora, YouTube, and iTunes could implement this type of system in addition to existing algorithms to generate a unique listening experience. Though since these services already have a recommendation algorithm in place, it would be most beneficial for a new streaming service to be created that focused their recommendations solely on audio, allowing independent on smaller artists to increase their audience, purely on the music's own merit.

The Dataset

A decent chunk of the project was spent on gathering the data. The initial intention was to work with all 1 million songs in the Million Song Dataset, but saw that hardware would be a limiting factor, and so for this project specifically, I decided to work with the random subset given by the compilers of the Million Song Dataset, which consisted of 10,000 songs randomly selected from the larger dataset as well as a small portion of the larger set, as broken up for distribution.

Cleaning/Wrangling

One of the largest challenges faced was getting the data into a useable format. The original dataset contained all the features needed on the single song level. Each hdf5 file was a song containing song features. The random subset of 10,000 of the million tracks nested in folders which was then zipped up into one file. The rest of the 1 million song dataset was structured similarly to this, single hdf5 files inside dozens of folders then compressed into a tar.gz file. After unpacking the subset, a python program was used which designed specifically to extract the features from the million song database set to read all the tracks into a single dataframe and did some exploratory data analysis. There was a ton of missing data where important song data features should have been. I knew that this alone would not be a good enough dataset for a project so I decided to use the subset as a list of songs that would be included in the dataset and turned to another source that I knew would have the data I needed, Spotify's API.

The million-song dataset was scraped off of an API, which was later acquired by Spotify so most of the songs from that dataset would be there, excluding the fact that Spotify has a nearly immeasurably large database of songs on its own. Therefore, from the original hdf5 dataset, only the artist and song name were extracted. Getting the song features from Spotify then became a two-fold process.

The first was passing in the artist name and song title into the search portion of Spotify's API in order to obtain the Spotify ID for it. The maximum number of queries able to be searched at a time in 50, so the artist/song pairs were grouped into smaller lists of 50. After running through the set, the total number of data points was less than 10,000 meaning that a portion of the songs in the original sample set could not be found in Spotify's library. Since a larger dataset is normally always better, I used the first tar.gz file of the entire dataset to gather more artist name/song title pairings to fill out a more robust dataset. After gathering the Spotify IDs, then Spotify's web API was used to get audio features for several tracks by having arrays of 100 Spotify IDs at a time to return audio features. They were returned as a json file which was subsequently turned into a dataframe. At that point, I realized that there was duplicates as the original subset did contain songs that were also in the first file of the large dataset. After checking for and removing duplicates, the entire dataset was compiled into one large dataframe with approximately 14000 tracks and all its accompanying audio features. The only identifying portion in the dataframe of audio features was the Spotify ID which doesn't bear any useful information to a person, so a dictionary was created with the Spotify IDs as well as the Artist and Name to join the set of audio features to a song title that people would actually recognize. Then extra columns were cleaned that had no relevance to the final product, leaving only relevant audio features that would be used in the clustering algorithm, one ID, and an artist name and song title. The features used in the algorithm included: acousticness, danceability, duration, energy, instrumentalness, key, liveness, loudness, mode, speechiness, tempo, time signature and valence.

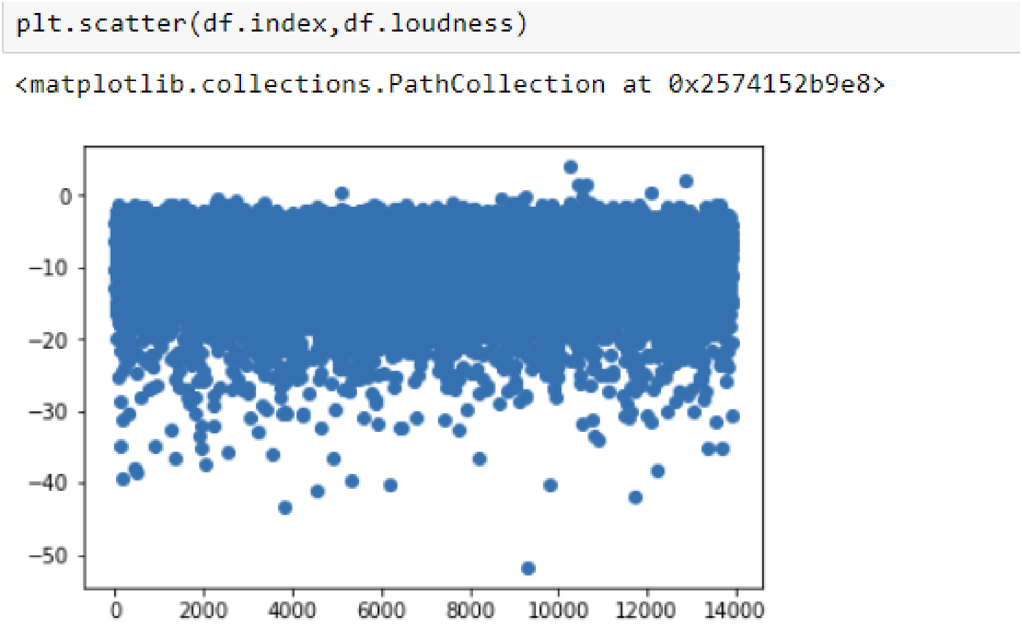
Then all numerical variables were normalized so that all values are between 0 and 1 to make differences able to be computed within al the same scale, so one category where all variables are hundreds of times larger does not skew distance/inertia in clustering.

In addition, after deciding to work with kmeans clustering, categorical features needed to be converted into a format recognizable by kmeans, which included the mode and key features. In addition, time signature was used in the categorical sense since the complete time signature was not provided, so the assumption was made that the songs with the same values in the time signature feature could be grouped together and therefore could be used as a categorical variable rather than a continuous one.

Exploratory Data Analysis

In terms of actual visual representation of this data set, it was a little bit difficult to do standard EDAs. In this clustering, each feature was a dimension, so it was difficult to represent a data point in 10-15 dimension. However, to do the regularization, and during the different clustering methods, there was a bit of visual analysis to be done.

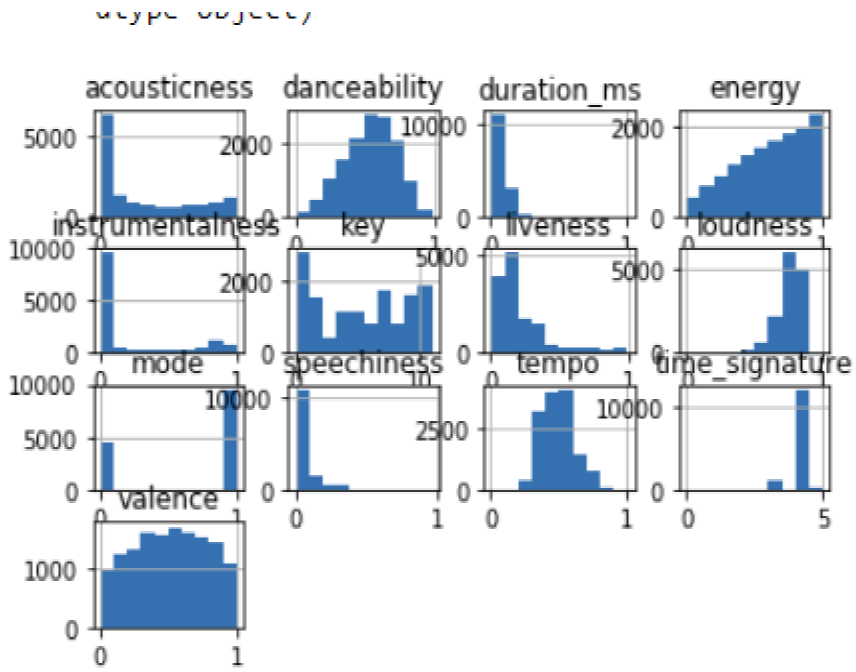
First, a simple plot was created of where the specific feature of data was actually located. For example, loudness is plotted below. However, a scatterplot of nearly 14,000 data points does not do much to actually show you where the data is located.



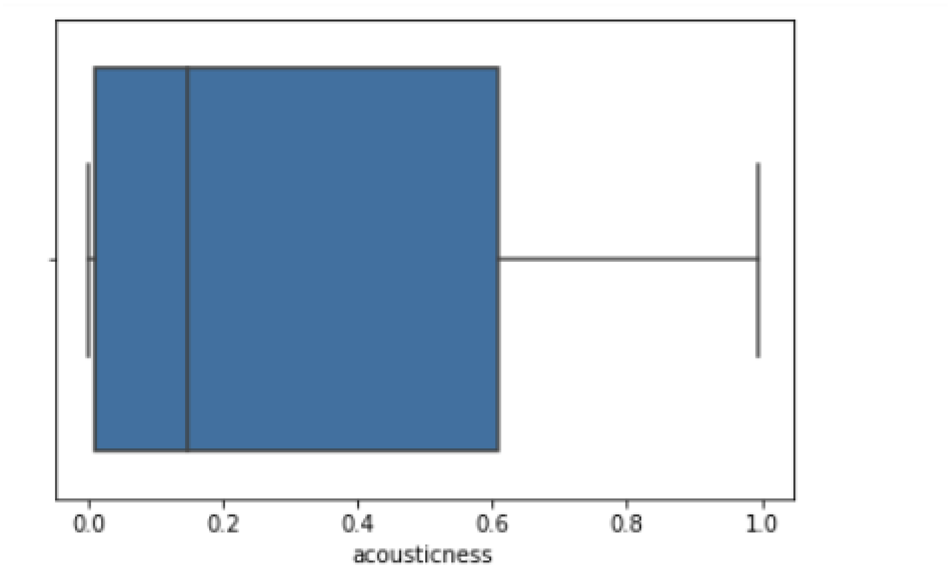
```
plt.scatter(df.index,df.loudness)
```

```
<matplotlib.collections.PathCollection at 0x2574152b9e8>
```

In addition, histograms of the features prior to encoding don’t tell us too much either.



The best way to visualize all of the numerical features, as well as simultaneously look at outliers, was the boxplot. One example is shown below, others can be viewed in the notebook.

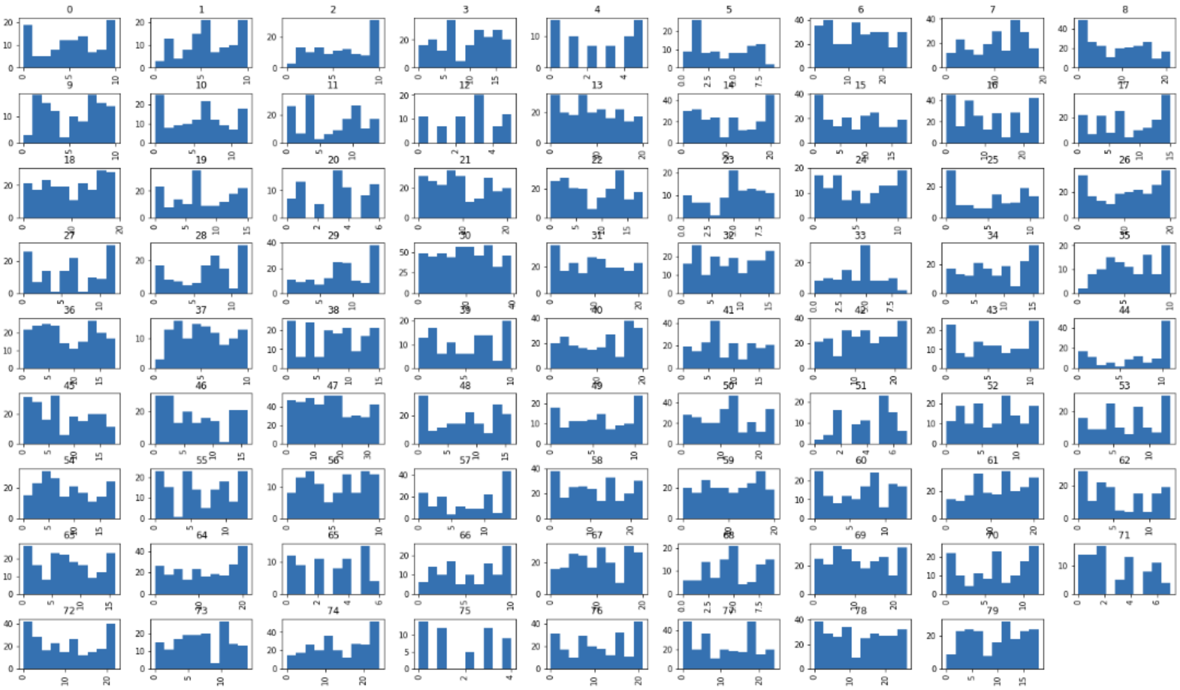


For the most part, there were no tails on the boxplots, with a few exceptions making cleaning the data a lot easier. The only boxplot with a true tail of outliers was speechiness, which marks how much talking there is in a track. For example, songs with no vocals or speech would be extremely low, where as spoken word pieces on spotify would rank as the highest. I left the tail in the data since I intended for spoken word pieces to be grouped with other spoken word pieces.

The Model

Trying to build a recommendation system ended up being a two step process. The primary method needed for this was clustering, and since my data had both categorical and numerical data, I immediately went to seek out a clustering algorithm that would be able to handle both categorical and numerical data together, and found the ensemble algorithm KPrototypes, which functions similar to kMeans except it can deal with categorical data so long as the algorithm is pointed to which features are categorical. The elbow method was usedto figure out the optimal number of clusters. However, in regards to this dataset, it would be best to pick the size of the clusters based on the number of recommendations that I was making. However, hardware limitations would limit the number of clusters so the maximum number of clusters the computer could handle was chosen, which was 80.

The kPrototypes algorithm was run once and the clusters were way too widespread to be able to get a good recommendation system and contained too many data points in each cluster, so I proceeded to take each cluster, and break it apart into separate clusters. The resulting dataset was that each song had a primary and secondary cluster associated with it. However, the cluster sizes were not consistent enough to be able to build a good enough recommendation system where a consistent amount of recommendations would be given each time a song was chosen as shown by the histogram of the secondary clusters based on primary clusters below.



As you can see, the size of the clusters are inconsistent, so an alternate way to do the clustering was needed. There was too much emphasis on keeping the categorical features, and I decided to revert back to KMeans as how to proceed with the modeling. In addition, there were no hardware limitations unlike KPrototypes. Therefore, after doing the cleaning necessary to encode the categorical features into numerical ones, the clusters were available immediately without additional data manipulation. The cluster number to be 1000 leading to a cluster size of roughly 13 to 14 for each as an average. Due to how clustering is performed I knew there would be variance in the actual number of data points in each cluster, so I wanted to pick a large enough number so that they could be close enough to each other but also provide a large enough cluster to make recommendations. As a result, I only needed to do a single clustering. The resulting model had a cost/inertia of about 20% of that of KPrototypes, leading me to believe that this was generating a better model and that I should use KMeans over KPrototypes.

However, due to the differing sizes of clustering there were quite a few small clusters, some with even only one song inside of them. A cutoff of 5 recommendations was made so that larger clusters would have ample recommendations and then all clusters in the dataset that were smaller than 5 had to be dealt with. This was done by singling out all the songs in the clusters that contained less than 5 songs. These would need to have recommendations already generated and stored since clustering did not find a large enough set. To manually generate recommendations, pairwise distances between the song in the small cluster and every other song in the dataset were generated, sorted by the shortest, and then 5 were picked to return as the most similar. In essence this is what clustering was doing on a larger scale, figuring out the closest song, but the reason why machine learning was implemented in this case, is that it is computationally too expensive to be able to constantly calculate pairwise distances between every single song in a dataset. This, however, ended up being a decent work around to dealing with the small clusters. The recommendations were saved in a separate dataframe.

Future Applications/What I would Change

In a perfect world, I would have been able to use all 1 million songs in the million song dataset, but there is nothing stopping this from working on Spotify's entire library. In addition, Spotify has not just audio features but also audio analysis but there does not seem to be a set format in which that data is returned, so I refrained from working on it for this project. The last time the million song dataset was nearly a decade ago, so if I was more creative I would have figured out how to incorporate more recent songs, however, Spotify's search algorithm put a big weight on popularity when searching for songs, which I wanted to refrain from, made it difficult to get a well-rounded set of both very popular but also very unpopular songs to create a balanced dataset. What I would have also liked that was not included in this dataset in additional audio features such as timbres, pitch, and other frequency type signals. Audio processing would have generated an even better recommendation system but unfortunately was not able to be completed in the time frame of this project, but is something I want to look into continuing into the future.

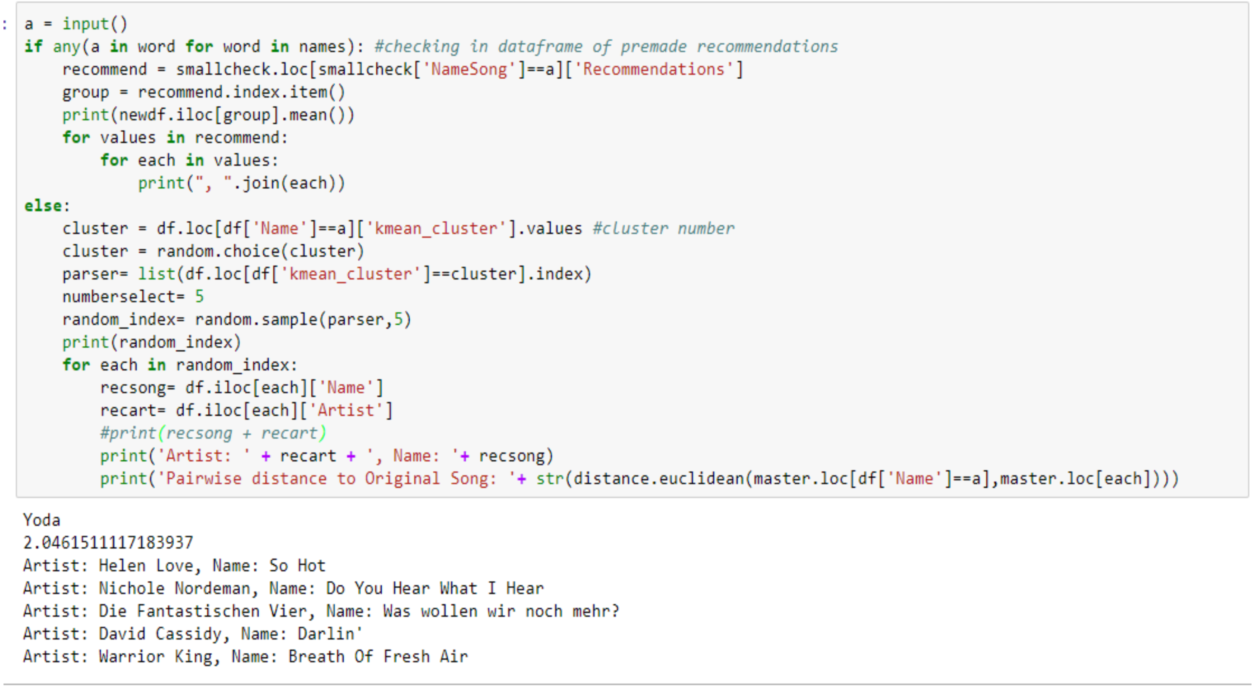
The hardware limitation as well as the size of the dataset were also constraints that I would have liked to look at. If I wanted to run through the entire dataset, this would've needed me to launch a cluster on something like AWS to be able to have the hardware capable of making this calculation in a reasonable amount of time. This would also allow me to move beyond the million song dataset as well. If it were within reason of me for obtain, I would have liked to do this on something like the entire Spotify library to compare with the algorithm that is currently used for song recommendation which takes into account popularity.

In addition, there is an argument to prepopulate even more than just the recommendations of small clusters, and introduce separate dataframes that would include songs by the same artists, or popular songs that are searched often and constantly listened to together. It would generate more noise in the original dataset, but could be implemented by allowing the user to choose what they would like to see more in their recommendations. This would all be implemented with a more robust and large dataset where it would make more sense, after rerunning clustering without the hardware limitation.

Results

The way that the recommendation system works is the user/whomever runs the cell inputs the name of a song that is in the dataset. The song is then searched in the dataframe of predetermined recommendations. If it is found in that dataframe, the recommendations are returned, along with the average pairwise distance of all of the songs that it was compared to. If the song is not in that dataframe, it searches the cluster the song is in, and gets 5 other random songs from that cluster, and returns their pairwise distance to the original song entered. Examples are shown below.

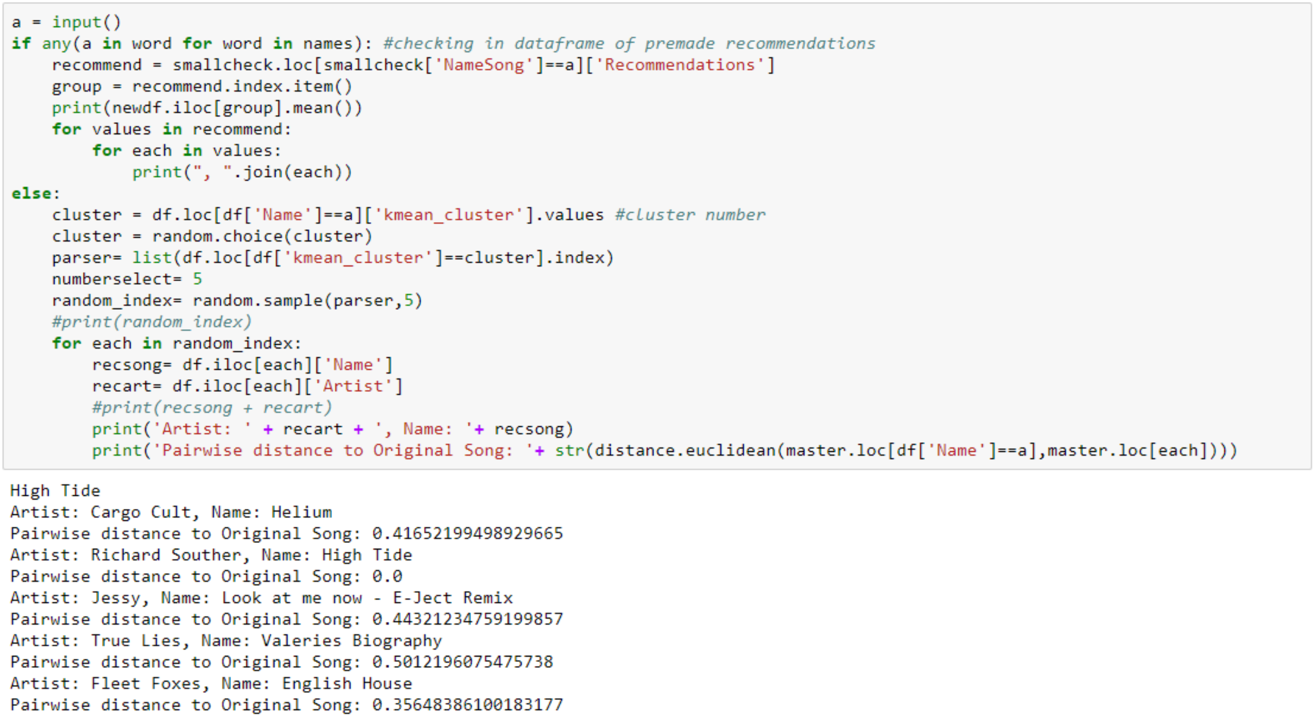
For songs that are in the premade dataframe:



```
a = input()
if any(a in word for word in names): #checking in dataframe of premade recommendations
    recommend = smallcheck.loc[smallcheck['NameSong']==a]['Recommendations']
    group = recommend.index.item()
    print(newdf.iloc[group].mean())
    for values in recommend:
        for each in values:
            print(", ".join(each))
else:
    cluster = df.loc[df['Name']==a]['kmean_cluster'].values #cluster number
    cluster = random.choice(cluster)
    parser= list(df.loc[df['kmean_cluster']==cluster].index)
    numberselect= 5
    random_index= random.sample(parser,5)
    print(random_index)
    for each in random_index:
        recsong= df.iloc[each]['Name']
        recart= df.iloc[each]['Artist']
        #print(recsong + recart)
        print('Artist: ' + recart + ', Name: '+ recsong)
        print('Pairwise distance to Original Song: '+ str(distance.euclidean(master.loc[df['Name']==a],master.loc[each])))
```

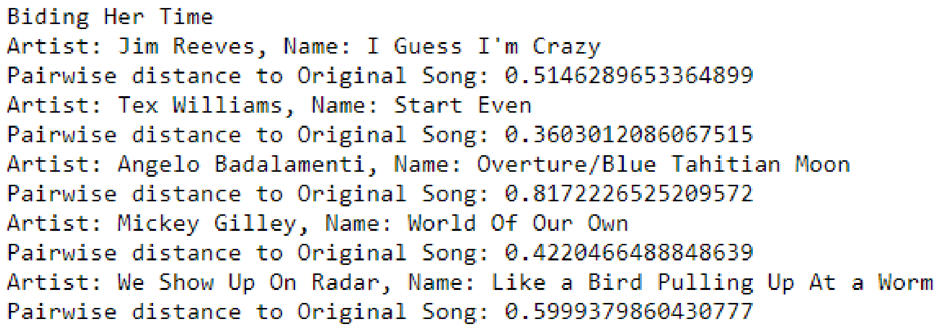
```
Yoda
2.0461511117183937
Artist: Helen Love, Name: So Hot
Artist: Nichole Nordeman, Name: Do You Hear What I Hear
Artist: Die Fantastischen Vier, Name: Was wollen wir noch mehr?
Artist: David Cassidy, Name: Darlin'
Artist: Warrior King, Name: Breath Of Fresh Air
```

And this is what the output would look like in circumstances that the song is not in the premade recommendation dataframe.

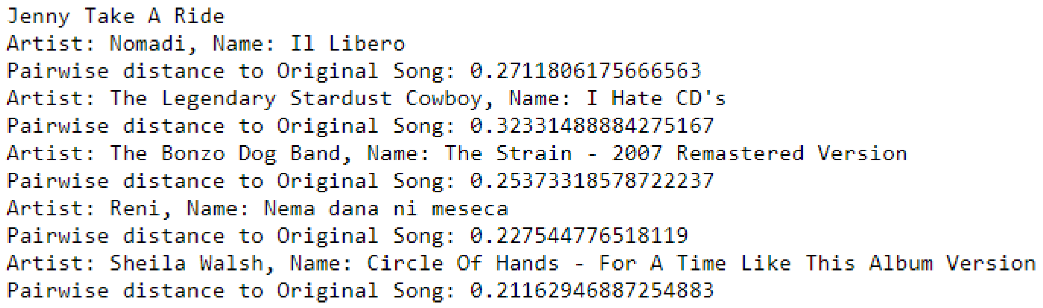


```python
a = input()
if any(a in word for word in names): #checking in dataframe of premade recommendations
    recommend = smallcheck.loc[smallcheck['NameSong']==a]['Recommendations']
    group = recommend.index.item()
    print(newdf.iloc[group].mean())
    for values in recommend:
        for each in values:
            print(", ".join(each))
else:
    cluster = df.loc[df['Name']==a]['kmean_cluster'].values #cluster number
    cluster = random.choice(cluster)
    parser= list(df.loc[df['kmean_cluster']==cluster].index)
    numberselect= 5
    random_index= random.sample(parser,5)
    #print(random_index)
    for each in random_index:
        recsong= df.iloc[each]['Name']
        recart= df.iloc[each]['Artist']
        #print(recsong + recart)
        print('Artist: ' + recart + ', Name: '+ recsong)
        print('Pairwise distance to Original Song: '+ str(distance.euclidean(master.loc[df['Name']==a],master.loc[each])))
```

```
High Tide
Artist: Cargo Cult, Name: Helium
Pairwise distance to Original Song: 0.41652199498929665
Artist: Richard Souther, Name: High Tide
Pairwise distance to Original Song: 0.0
Artist: Jessy, Name: Look at me now - E-Ject Remix
Pairwise distance to Original Song: 0.44321234759199857
Artist: True Lies, Name: Valeries Biography
Pairwise distance to Original Song: 0.5012196075475738
Artist: Fleet Foxes, Name: English House
Pairwise distance to Original Song: 0.35648386100183177
```



```
Biding Her Time
Artist: Jim Reeves, Name: I Guess I'm Crazy
Pairwise distance to Original Song: 0.5146289653364899
Artist: Tex Williams, Name: Start Even
Pairwise distance to Original Song: 0.3603012086067515
Artist: Angelo Badalamenti, Name: Overture/Blue Tahitian Moon
Pairwise distance to Original Song: 0.8172226525209572
Artist: Mickey Gilley, Name: World Of Our Own
Pairwise distance to Original Song: 0.4220466488848639
Artist: We Show Up On Radar, Name: Like a Bird Pulling Up At a Worm
Pairwise distance to Original Song: 0.5999379860430777
```



```
Jenny Take A Ride
Artist: Nomadi, Name: Il Libero
Pairwise distance to Original Song: 0.2711806175666563
Artist: The Legendary Stardust Cowboy, Name: I Hate CD's
Pairwise distance to Original Song: 0.32331488884275167
Artist: The Bonzo Dog Band, Name: The Strain - 2007 Remastered Version
Pairwise distance to Original Song: 0.25373318578722237
Artist: Reni, Name: Nema dana ni meseca
Pairwise distance to Original Song: 0.227544776518119
Artist: Sheila Walsh, Name: Circle Of Hands - For A Time Like This Album Version
Pairwise distance to Original Song: 0.21162946887254883
```

Conclusions

There is a pretty vast difference in the clustering distances overall from switching from KPrototypes back to kmeans, which solidifies that this was the correct clustering method to use all along. This falls in line with general kmeans usages, particularly dealing with large datasets, and the ability to keep on improving on clusters until convergence. To a consumer, the only thing they would be worried about is the recommendations that would be returned, and thus the inertia of the cluster or distance between the songs in the cluster would not really matter. These would be presented to the company trying to implement the algorithm.

The recommendation system above can only take input from the song titles that are in the original dataset, but are able to consistently output a set of recommended songs quickly. In addition, due to the clustering method, there is some variance that is tolerated when looking at the larger cluster sets, which is acceptable since the recommendations should not be the same each time the song is played, nor in instances of playlists of recommended songs would a listener want the same songs in the same order. The smaller clusters have premade lists that would not be cycled through as in the clusters, but this is for the sake of speed, as it is possible to compute all paired distance each time a song from a small cluster is entered, but in a recommendation system, we do not want to wait 8 minutes to get a list of pairwise smallest distances since the length of that first song is most likely less than 8 minutes.