Austin Rochon, Emily Xie, Mark Lock
APCOMP 209A: Data Science I
11-20-2019

Milestone 3: EDA and Baseline Model

Data Description:
For our initial EDA, we used two datasets:

1. "Million Playlist Dataset": This came from the 2018 Spotify Recsys Challenge. Per the website, it contains 1 million playlists created by Spotify users, and contains the below fields. A data dictionary containing full descriptions of each field is in Appendix A.
    - *pid (int)*
    - *pos (int)*
    - *artist_name (string)*
    - *track_uri (string)*
    - *artist_uri (string)*
    - *track_name (string)*
    - *album_uri (string)*
    - *duration_ms (float)*
    - *album_name (string)*

2. "Song Attributes Dataset": We generated this dataset by writing a python script that crawled the Spotify API, fetching audio data for each distinct track in our "Million Playlist Dataset". It contains the fields listed below. A data dictionary containing full descriptions of each field is in Appendix B.
    - *trackid (string)*
    - *duration_ms (int)*
    - *key (int)*
    - *mode (int)*
    - *time_signature (int)*
    - *acousticness (float)*
    - *danceability (float)*
    - *energy (float)*
    - *instrumentalness (float)*
    - *liveness (float)*
    - *loudness (float)*
    - *speechiness (float)*
    - *valence (float)*
    - *tempo (float)*

Data Issues / Data Engineering Challenges:
We encountered multiple issues with our dataset. For one, the "playlist-song" dataset comprised of just under 70 million rows total. A dataset like this was far too large to manage locally on our machines without being confined to importing selected CSVs as a subsample, which would be significantly limiting down the road.

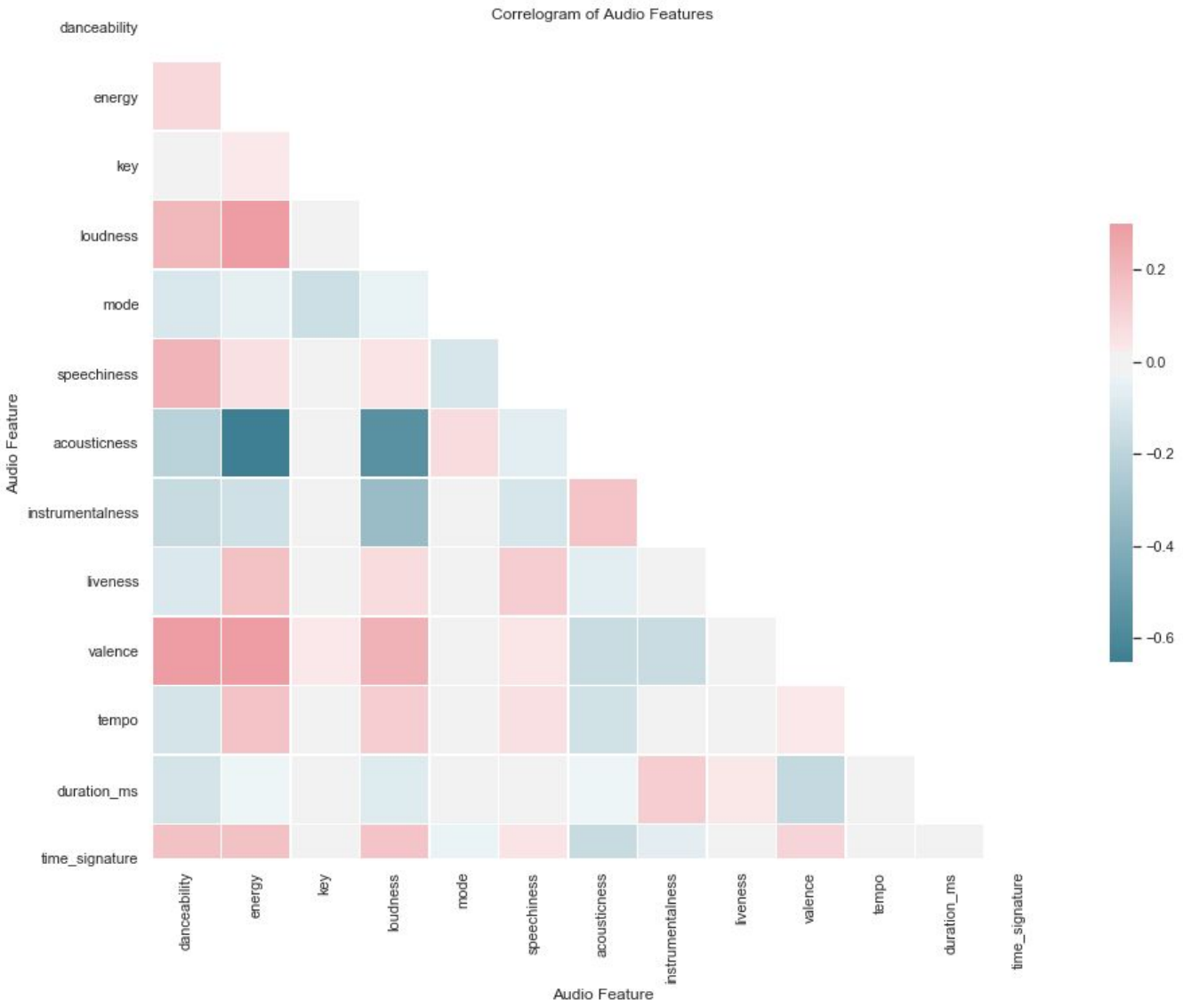
Thus, we decided to host our data on Google Cloud Platform (GCP) via their distributed big data store, BigQuery. This would allow us to efficiently query and manipulate any subset of the million playlist dataset using their SQL-like interface. After some research, we were able to configure our setups such that we could retrieve the data through BigQuery and store it in our Jupyer notebooks to perform the EDAs locally. Ultimately, this setup allowed us to analyze larger datasets; instead of confining ourselves to one CSV of ~70,000 rows, we were able to conduct analysis performantly on data sets of up to 500,000—and theoretically more, though we decided that this was a decent cutoff point.

Additionally, moving our data over into GCP proved invaluable during the initial steps of querying each song through the Spotify features API. Since songs are duplicated across playlists, using BigQuery, we were able to extract a uniquified dataset of just 2 million songs from the 70 million rows and save them to a separate table. With this technique, a scraping endeavor that would've taken an estimated 19+ hours total was reduced to under an hour. From there, it was a matter of performing a simple outer join query on the original songs dataset to associate the features back to each playlist-song row. Lastly, this ability to isolate and query uniquified songs enabled us to sample a truly random subset of the song data (without including duplicate songs or introducing bias by selecting songs clustered around a set of playlists in a given CSV) for our EDA.

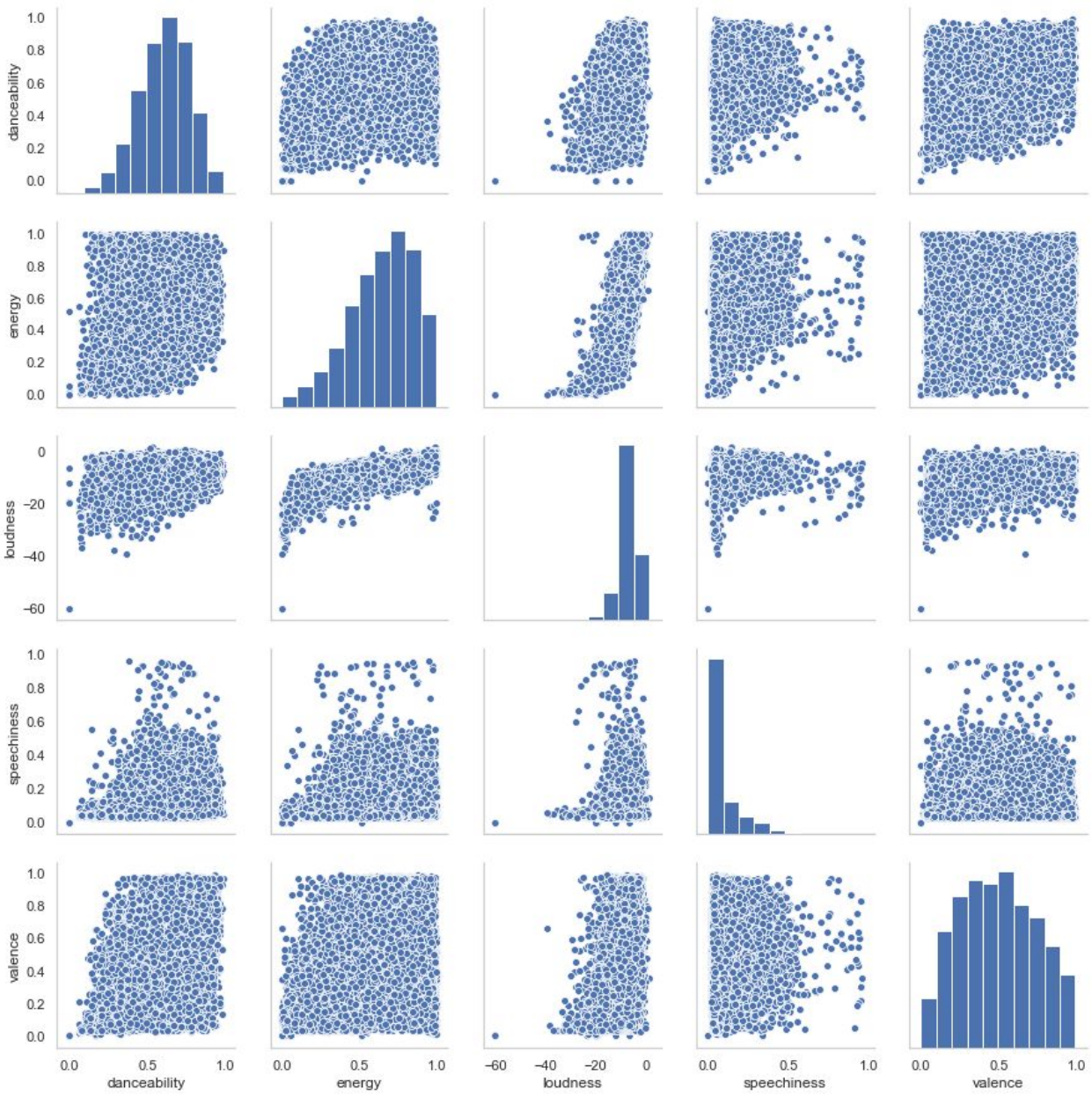
Further challenges arose when we discovered that the playlist IDs were unique only to each file, as opposed to being a globally unique playlist ID (each file included playlists with a PID of 0 - 999). This would become an issue for any analyses we wanted to run on playlist-specific data. To account for this, we re-processed each CSV, adding an additional column called "unique_pid" which combined the file number with the playlist ID, and re-processed the data in BQ.

Initial EDA:

We began by examining the correlations of the audio features in our data set. The correlogram below depicts correlations that are intuitive. For example, *acousticness* is strongly negatively correlated with *loudness* and *energy*. On the other hand, *energy* is positively correlated with *loudness* and *valence* (the musical "positiveness" associated with a track). In conclusion, although there is some multi-collinearity between the features, we can still include them all, given that their absolute correlations are under 0.75.



Pairwise relationships between audio features



Identifying Groups (i.e. Labeling Our Data)

The end goal of this project is to build a recommender engine that recommends new songs to add to a given playlist In this iteration, we assumed that recommendation is equivalent to finding similar songs (we will discuss more nuanced implications in our steps forward). However, we were quickly faced with an important question: how do we measure similarity? To answer this question, we pursued two paths in parallel: 1) using tracks within playlists as a proxy for similarity. The hypothesis here was that users who created playlists probably would group similar types of songs together. 2) algorithmically identifying groups using unsupervised learning to facilitate (1).

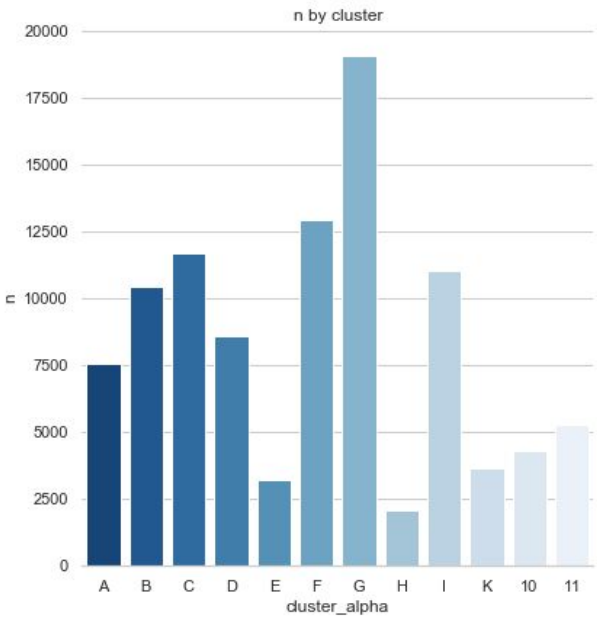
K-means Clustering to Identify Groups

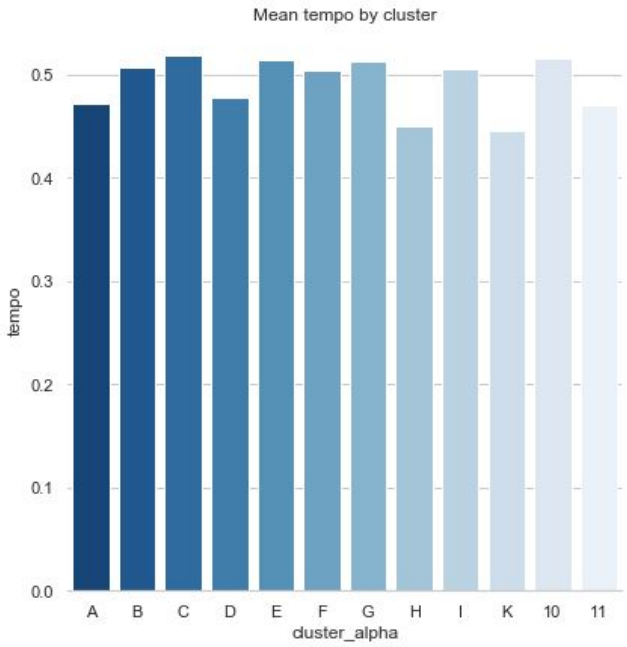
As described above, the second method we used to identify similar tracks was unsupervised learning. Specifically, to group our songs into logical clusters, we applied a k-means algorithm to our data. K-means is an unsupervised learning method that groups data into clusters according to the mean

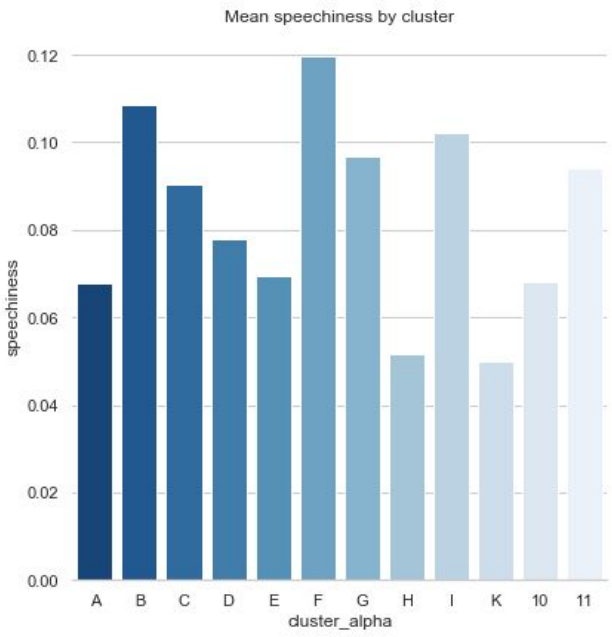
value of their features. At a high level, it works by initializing a user-specified number of *k* centroids, then assigns each data point to the closest centroid (Towards DS: K-Means).

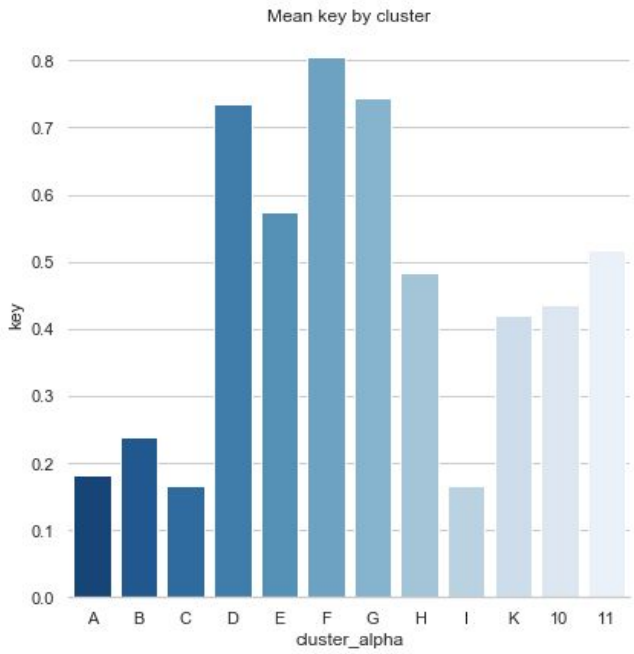
We chose k by optimizing the heterogeneity of the labels, after testing several cluster sizes. We landed on *k=12* as an optimal cluster size. Below are the mean values for each cluster for some selected features. As you can see, some features (*speechiness, key)* are quite helpful in differentiating our songs, while some are virtually useless (*tempo).*

cluster	n	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness
0	7541	0.505900	0.338919	0.182650	0.773325	1.0	0.067782	0.740213	0.023928
1	10437	0.623368	0.728970	0.237564	0.853785	0.0	0.108416	0.125291	0.023882
2	11681	0.529585	0.706261	0.166720	0.852176	1.0	0.090357	0.092735	0.022930
3	8580	0.542608	0.398339	0.734001	0.786810	1.0	0.077834	0.682388	0.023111
4	3228	0.599089	0.715469	0.574265	0.824760	0.0	0.069399	0.098801	0.753206
5	12912	0.636302	0.732383	0.804220	0.852720	0.0	0.119512	0.117266	0.022587
6	19077	0.600792	0.739688	0.744209	0.854241	1.0	0.096917	0.107959	0.018636
7	2052	0.378203	0.243854	0.482811	0.670325	0.0	0.051539	0.840178	0.829380
8	11045	0.681039	0.727627	0.165711	0.848257	1.0	0.102303	0.185040	0.017750
9	3642	0.380843	0.221129	0.419050	0.666101	1.0	0.050008	0.868227	0.834616
10	4277	0.567928	0.714587	0.435819	0.823181	1.0	0.068065	0.110187	0.767536
11	5285	0.537561	0.394093	0.518001	0.779155	0.0	0.094049	0.694022	0.031120



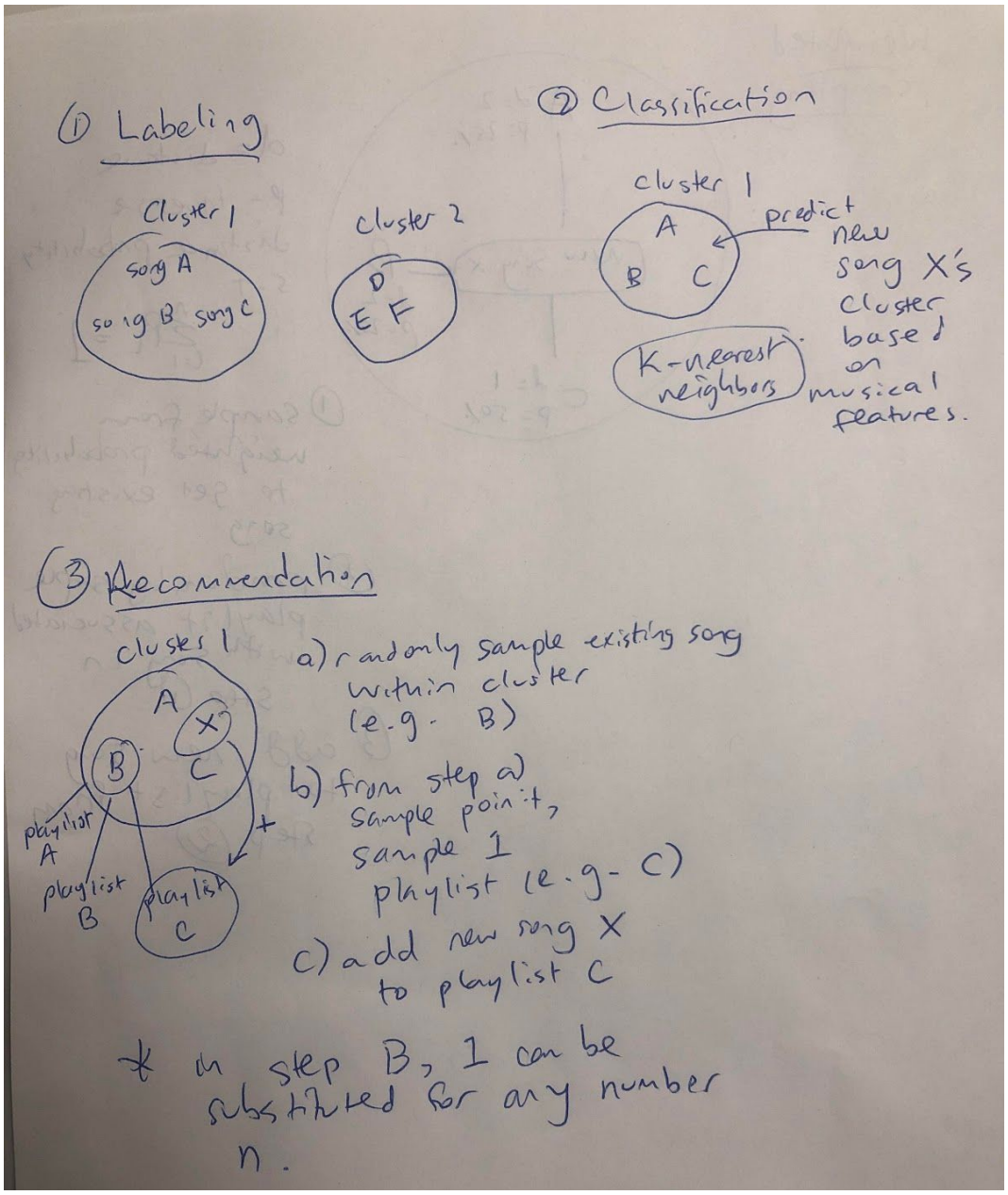






Baseline Model: K-Nearest Neighbors Classification and K-means Labeling

In this iteration, we have established the assumption of finding similar songs to songs already existing in playlists. Using a combination of supervised and unsupervised learning we have come up with the initial recommendation approach:



1. **Labeling** K-means: cluster universe of all songs in all playlists into 12 clusters based on spotify music features. This serves as a 1-dimensional label with 12 categories which makes computation extremely efficient if we run supervised learning over it.
2. **Classification** K-NN: Upon receiving a new song that is not labeled with one of the 10 categories, we can use K-nearest neighbors based on musical features to identify the cluster in which this new song falls. We found that the model with K=100 gave us the highest cross-validation scores (mean: 97.8%) and are confident in its ability to correctly put the new song in an appropriate cluster:

	k	cv_scores_mean
0	1	0.931746
1	5	0.952772
2	100	0.978347
3	250	0.978222
4	500	0.976643
5	1000	0.973097
6	2500	0.954715
7	5000	0.900745
8	10000	0.807318

Using k=100 yielded the following accuracy scores/misclassification rates on the training/test data:

```
Accuracy score on training data:                0.9810162270534427
Misclassification rate on training data:        0.018983772946557287
Accuracy score on test data:                    0.979200080192462
Misclassification rate on test data:            0.020799919807538036
```

3. **Recommendation** Monte Carlo: When a new song's cluster has been predicted, we can then say that this song is eligible to be added to an existing song's playlist within that cluster. All songs within the cluster are similar to one another. Which playlist gets the new song can be determined as follows:
    a. Randomly sample one song out of all existing songs in the cluster
    b. For the selected existing song, randomly select a playlist associated with that song (a song can have multiple playlists associated with it)
    c. Add the new song to the playlist sampled in step b

    Note that this is a case where we've artificially decided to only add a new song to one play list at any one time. You could imagine Spotify might have ideas around how rapidly they want to introduce novelty to users. The higher appetite for novelty would mean we could add the new song to more playlists. A lower appetite for novelty would mean we add the new song to fewere playlists.

We've identified the efficacy of steps 1 and 2, but why use a monte carlo approach to recommend? This makes sure we have the ability to scale up or scale down novelty/recommendation rate. We don't want to get into a situation where we add a new song to every existing song's playlists in a cluster - this could be in the hundreds. Although we haven't talked about this yet, but another consideration of recommendations is the ability to give something a little different from what's already existing but be familiar enough to be well received. The monte carlo approach introduces stochasticity into the system, so you could sample an in-cluster existing song that might be far away from the new song and that would still be a good thing for this use case. This works because the clusters are relatively large and there is some heterogeneity to the songs within the clusters.

Next steps

Our initial project proposal had outlined how we would recommend songs based on musical features that were predicted off of lyrics data. This would have required building an initial NLP model that connected lyrics to musical features and then fed forward to a recommender model. We have decided to not pursue the lyrics component due to complexities in reconciling Genius API (lyrics) with Spotify API (playlists and songs) and due to insufficient GPU resources to fully train our NLP model.

Our revised project goal is to:

Build a recommendation engine that recommends new songs to add to a given playlist purely based on musical features obtained from Spotify.

The operational definition is to:

Build a model that makes a decision of inclusion or exclusion for any given new song S_new in any given playlist X that contains existing songs S_1, S_2, S_3, …. S_n.

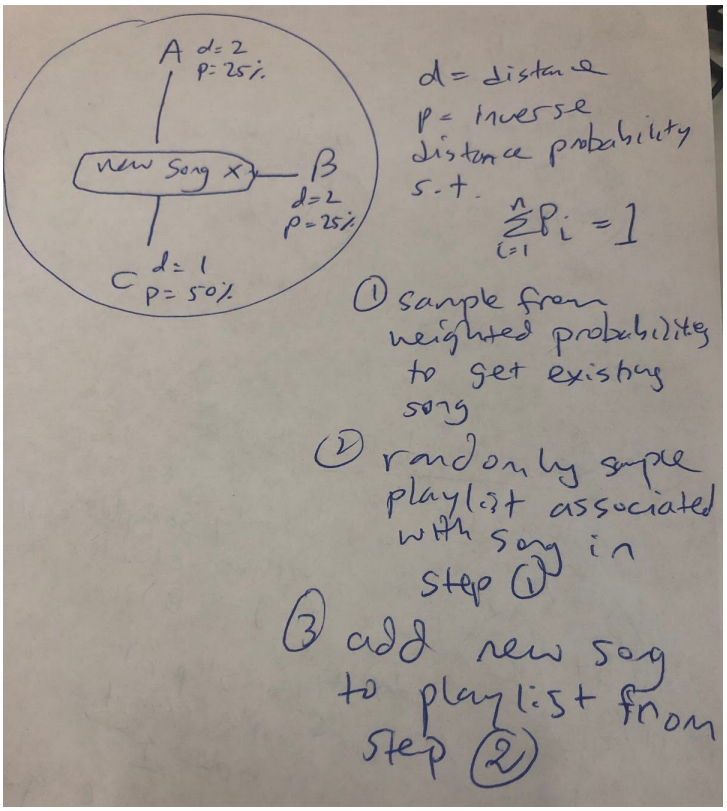
Note that this is unlike a traditional recommender model where we have to predict affinity through ratings or consumption (e.g. movies for netflix, items for amazon). We do not have a ground truth from Spotify on this so we will not be using a traditional collaborative filtering framework. This brings us to the assumptions we need to make.

Some additional considerations we would like to address in our next steps:

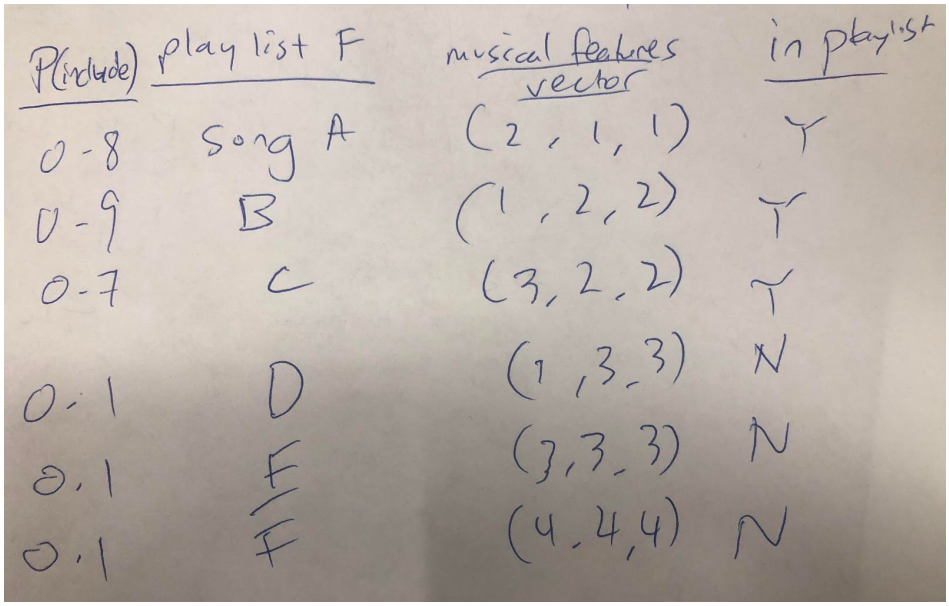
- **Similarity:** Similarity of songs within playlists is a sound assumption to make for recommending new songs for a given playlist
- **Novelty:** The value of song recommendations is that there is an element of novelty/exploration, so users like to see new songs that are familiar but offer just enough novelty to keep them interested. We will try to build this in structurally through our system.
- **Drift:** We want to build in bounds within our model to address song drift. This is the idea that if an algorithm recommends songs that move the music features away from the original average music features of a given playlist over time, we encounter drift. The user will then have to re-seed the playlist with their original tracks to get the algorithm back on track. We want to avoid this by imposing some bounds on our model. For example, in our preliminary model, our bounds were songs have to be in-cluster to be recommended in a relevant playlist.
- **Representativeness:** Is a song representative of the playlist and/or is a playlist representative of certain songs? Not all playlists are made equally in terms of diversity. We will need to devise models that account for differences in diversity of playlists' musical features.

New modelling approaches to explore:

1. **Enhanced baseline model - weighted sampling:** We can refine the sampling step in the initial model by weighting in-cluster songs based on distance of music features. Existing songs closer to the new song will be weighted more in the sampling procedure based on inverse distance weighting. See diagram below:



2. **New model - Unique playlist models:** So far we our baseline model has assumed that our existing songs are representative of the playlists in which they are in. But what if there are some playlists that are way more diverse in musical features than others (i.e. a John Mayer playlist has homogenous acousticness levels while a top 40 playlist will have very different levels across songs)? This calls for us to create playlist specific models whereby for each song we can predict the likelihood of playlist inclusion based on in-playlist and out-of-playlist songs' musical features. Since each model will be specific to a playlist, the diversity/homogeneity/representativeness of musical features will be fit better than in an omnibus model. These models can be simple logistic regressions but we will evaluate more complex variations if necessary.



3. **New model - Simplest K-NN using Mahalanobis distance:** This is a simplification of our baseline model. We want to test the simplest case so we can benchmark to see if a simple model can actually perform on par with a models that are more complex. In this model, we take each songs' musical features, calculate all possible pairwise distances (using Mahalanobis distance) and for each new song, add it to the top 5 closest existing song's playlists. This gets rid of the stochastic element and the clustering element from our baseline model.

Appendix A: Million Playlist Dataset Data Dictionary

- *pid (int):* playlist id
- *pos (int):* row id
- *artist_name (string):* artist name
- *track_uri (string):* spotify's unique track identifier
- *artist_uri (string):* spotify's unique artist identifier
- *track_name (string):* track name
- *album_uri (string):* spotify's unique album identifier
- *duration_ms (float):* duration of track in milliseconds
- *album_name (string):* album name

Appendix B: Song Attributes Dataset Data Dictionary
All field descriptions below are from the documentation in the Spotify API:

- *trackid (string):* The Spotify ID for the track.
- *duration_ms (int):* The duration of the track in milliseconds
- *key (int):* The estimated overall key of the track. Integers map to pitches using standard Pitch Class notation . E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on. If no key was detected, the value is -1
- *mode (int):* Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0
- *time_signature (int):* An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure)
- *acousticness (float):* A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic
- *danceability (float):* Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is least danceable and 1.0 is most danceable
- *energy (float):* Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy
- *instrumentalness (float):* Predicts whether a track contains no vocals. “Ooh” and “aah” sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly “vocal”. The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content
- *liveness (float):* Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live
- *loudness (float):* The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude)
- *speechiness (float):* Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value
- *valence (float):* A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry)
- *tempo (float):* The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration

Additional Citations and Resources

https://developer.spotify.com/documentation/web-api/reference/tracks/get-audio-features/
https://cloud.google.com/bigquery/docs/visualize-jupyter