CS109a Final Project - Spotify Recommendation System

Group #55: Nick Kochanek, Jack Connolly, Chris Jarrett, Andrew Soldini

Project Goal:
Our goal for this project was to develop a system that could recommend reasonable songs to continue a playlist give a certain number of "seed" tracks. We formalized this task as giving a model a list of K input songs (as Spotify uris) and having the model output 500 suggested uris (ideally ranked by relevance). This falls in line with the formal specifications for the Spotify RecSys challenge, and allows us to compare results and approaches with top teams there. As such, we decided to evaluate our models using the same metrics the contest was based on, which will be described further on.

In [1]:
```
import requests

from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS109A/master/content/styles/cs109.css").text
HTML(styles)
```

Out[1]:

In [2]:
```
import json, sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from prettytable import PrettyTable
import warnings
import random
warnings.filterwarnings("ignore")

import spotipy
from spotipy.oauth2 import SpotifyClientCredentials

import pickle

pd.options.mode.chained_assignment = None

import sklearn
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.model_selection import train_test_split

import keras
from keras.models import Sequential
from keras.layers import *

from pandas.io.json import json_normalize
import json

from collections import Counter
```

```
Using TensorFlow backend.
```

Loading in the data (Million Playlist Dataset)

In [3]:
```
# After running into issues trying to pickle large objects (ie our grap
h),
# It turns out that there's an issue in the pickle implementation. This
 stack overflow function
# allows for easy saving of big objects
# https://stackoverflow.com/questions/42653386/does-pickle-randomly-fail
-with-oserror-on-large-files
def save_as_pickled_object(obj, filepath):
    """
    This is a defensive way to write pickle.write,
    allowing for very large files on all platforms
    """
    max_bytes = 2**31 - 1
    bytes_out = pickle.dumps(obj)
    n_bytes = sys.getsizeof(bytes_out)
    with open(filepath, 'wb') as f_out:
        for idx in range(0, n_bytes, max_bytes):
            f_out.write(bytes_out[idx:idx+max_bytes])
```

In [8]:
```
# This is the code from 'build_network.py'

# 20 is a lot (the biggest we were able to run)
# Use ~10 to finish in a reasonable time
NUMBER_OF_FILES_TO_USE = 5

"""
The code below builds the network
It also builds relevant objects to use for the other models
"""
song_name_to_uri, uri_to_song_name = {}, {}
track_to_artist_album, network = {}, {}

track_counts, artist_counts = {}, {}
playlist_lens, artists_perplay = [], []


track_codes = set()
playlists, uri_input, uri_expected = [], [], []

K = 10

f_start = 1000
f_end = 1999
for i in range(NUMBER_OF_FILES_TO_USE):
    with open('./mpd.v1/data/mpd.slice.{}-{}.json'.format(f_start, f_end
)) as f:
        data = json.load(f)

    input_, expected = [], []
    for playlist in data['playlists']:
        playlist_count = 0
        play_artists = {}
        playlist_dict = playlist.copy()
        playlist_dict.pop('tracks', None)

        for k, song in enumerate(playlist['tracks']):
            track_name  = song['track_name']
            track_uri = song['track_uri']
            shared_songs = np.array([s['track_uri'] for s in
                                     playlist['tracks'] if s['track_uri'
] != track_uri])
            playlist_count += 1
            artist_name = song['artist_name']

            if track_uri not in track_codes:
                track_codes.add(track_uri)
                track_to_artist_album[track_uri] = {'artist': song['arti
st_name'], 'album': song['album_name']}
                uri_to_song_name[track_uri] = track_name

            if track_name not in song_name_to_uri:
                song_name_to_uri[track_name] = track_uri

            if track_uri not in network:
                network[track_uri] = np.array(shared_songs)
```

```
            else:
                network[track_uri] = np.append(network[track_uri], np.ar
ray(shared_songs))

            if k < K:
                input_.append(track_uri)
            else:
                expected.append(track_uri)

            # EDA Stats Collecting
            if artist_name not in artist_counts:
                artist_counts[artist_name] = 1
            else:
                artist_counts[artist_name] += 1

            if track_name not in track_counts:
                track_counts[track_name] = 1
            else:
                track_counts[track_name] += 1

            if artist_name not in play_artists:
                play_artists[artist_name] = 1
            else:
                play_artists[artist_name] += 1

        playlists.append(playlist_dict)
        uri_input.append(input_)
        uri_expected.append(expected)

        # For EDA
        playlist_lens.append(playlist_count)
        artists_perplay.append(len(play_artists.keys()))

    print ("done loading file", i)
    f_start += 1000
    f_end += 1000


# Clean the network -> counts per song (normalized)
print("Cleaning up the Network a bit")
for uri in network :
    unique, counts = np.unique(network[uri], return_counts=True)
    network[uri] = {'songs' : unique, 'counts': counts / np.sum(counts)}


# Save all of the objects as pickles
save_as_pickled_object(network, 'pickled_network.pickle')

with open('songs_to_uri.pickle', 'wb') as f:
    pickle.dump(song_name_to_uri, f)

with open('uri_to_song.pickle', 'wb') as f:
    pickle.dump(uri_to_song_name, f)

with open('track_to_artist_album.pickle', 'wb') as f:
    pickle.dump(track_to_artist_album, f)
```

```
done loading file 0
done loading file 1
done loading file 2
done loading file 3
done loading file 4
```

Getting audio features from the Spotify API This takes awhile - you can just load to saved pickles 'audio_features.pickle' and 'uris_10.pickle' below.

In [9]:
```
# Load the pickles instead of rescraping
with open('./cs109_final_backend/cs109_final_backend/cluster_files/uris_10.pickle', 'rb') as f:
    uris = pickle.load(f)
with open('./cs109_final_backend/cs109_final_backend/cluster_files/audio_features.pickle', 'rb') as f:
    audio_features = pickle.load(f)

audio_df = pd.DataFrame(audio_features)

print(len(uris))
print(len(audio_df))
```

```
170380
170380
```

In [ ]:
```
from spotipy.oauth2 import SpotifyClientCredentials

cid ="1b81d49177e5464781a4957e5e0c1ae6"
secret = "c444a35689e247f8b5f9830662bae244"
client_credentials_manager = SpotifyClientCredentials(client_id=cid, cli
ent_secret=secret)
sp = spotipy.Spotify(client_credentials_manager=client_credentials_manag
er)
sp.trace=False

uris = list(uris)
spotify = spotipy.Spotify(auth='BQDuG3_3-tAv09LQlZKwYc-oodCcDdau9O-I-Ep_
O6IK-Uqvc5S3FKwdr5qtVu5Kq1khJCwkeaR9PnQJjvL6fBRPWdjJ9H_KRoGCrSlMP5DjdcLM
lWJybPF1VvJDuSwBpoxiLS_Qmr9R4z-RoDWDPNiZnlzCeJNxMMvLRg')

keys_to_remove = ["duration_ms", "type", "id", "uri", "track_href", "ana
lysis_url"]

start = 0
audio_features = []
while start < len(uris):
    response = sp.audio_features(uris[start:(100+start)])
    small_response = []
    for track in response:
        if track is not None:
            small_dict = {key:track[key] for key in track.keys() - keys_
to_remove}
        else:
            print('here')
            small_dict = {key:0.0 for key in response[0].keys() - keys_t
o_remove}
        small_response.append(small_dict)

    audio_features.extend(small_response)

    start += 100
    if start % 1000 == 0: print(start)

audio_df = pd.DataFrame(audio_features)


# Save the pickles
with open('uris_10.pickle', 'wb') as handle:
    pickle.dump(uris, handle, protocol=pickle.HIGHEST_PROTOCOL)
with open('audio_features.pickle', 'wb') as handle:
    pickle.dump(audio_df, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

Train Test Split:

As well as scaling so KNN and KMeans is meaningful

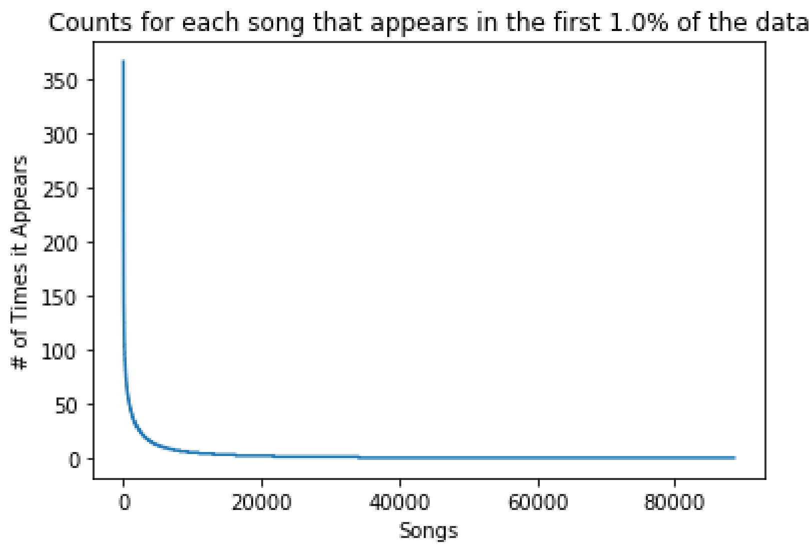
In [10]:
```
# Scale the features in audio_df to mean=0 and variance=1
from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
audio_scaled = ss.fit_transform(audio_df)

print(len(uris))
print(len(audio_scaled))

audio_dict = {}
for i in range(len(uris)):
    audio_dict[uris[i]] = audio_scaled[i]
```

```
170380
170380
```

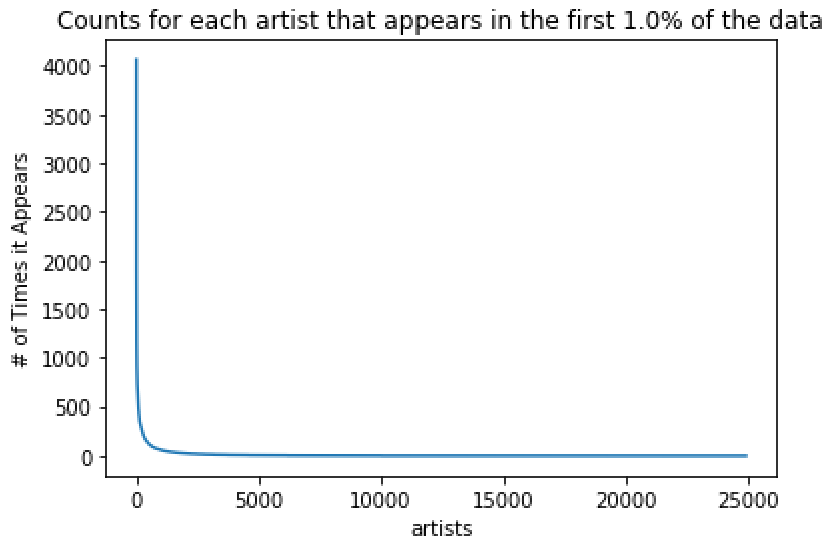
Data and EDA

The data sources we utilized in our explorations were the Million Playlist Dataset and the Spotify API. The MPD gave us the essential data to train models on subsets of playlists and evaluate the accuracy of our predictions. We used to Spotify API to get pre-computed audio features for songs, which allowed us to expore alternative model choices more in line with what we looked at in class.

In [11]:
```
# MPD EDA
track_names = list(track_counts.keys())
counts = list(track_counts.values())

num_files = 10
percent_data = (num_files / 1000) * 100

plt.plot(range(len(track_names)), sorted(counts, reverse=True))
plt.xlabel('Songs')
plt.ylabel('# of Times it Appears')
plt.title(f'Counts for each song that appears in the first {percent_dat
a}% of the data')
plt.show()
```

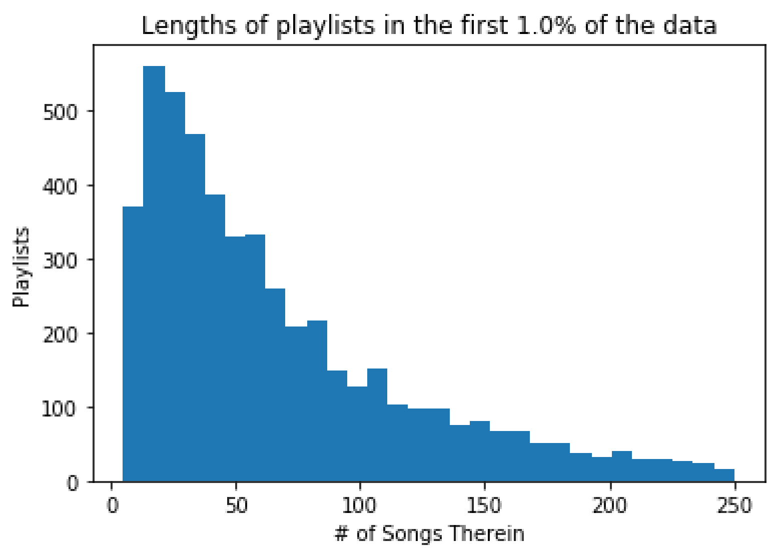


In [12]:
```
artist_names = list(artist_counts.keys())
counts = list(artist_counts.values())

plt.plot(range(len(artist_names)), sorted(counts, reverse=True))
plt.xlabel('artists')
plt.ylabel('# of Times it Appears')
plt.title(f'Counts for each artist that appears in the first {percent_da
ta}% of the data')
plt.show()
```



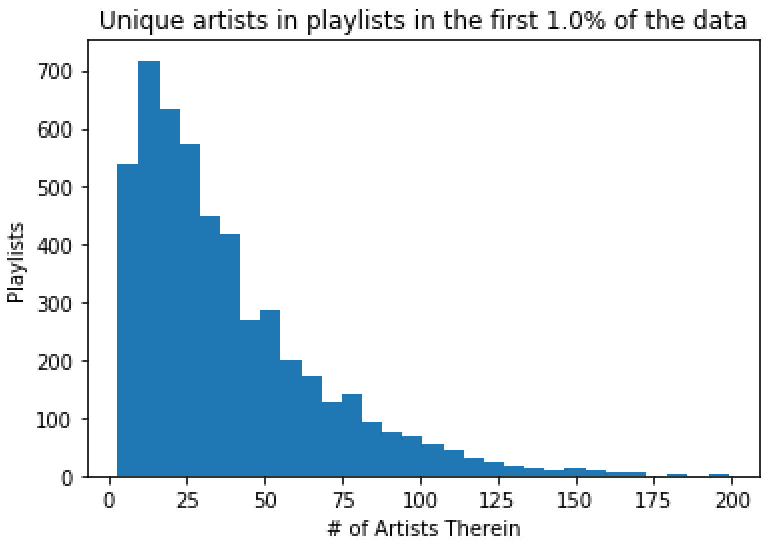
In [13]:
```
print(len(artist_names))
print(len(track_names))
print(np.mean(playlist_lens))
print(np.mean(artists_perplay))

plt.hist(playlist_lens, bins=30)
plt.xlabel('# of Songs Therein')
plt.ylabel('Playlists')
plt.title(f'Lengths of playlists in the first {percent_data}% of the dat
a')
plt.show()
```

```
24917
88522
66.9026
38.2456
```

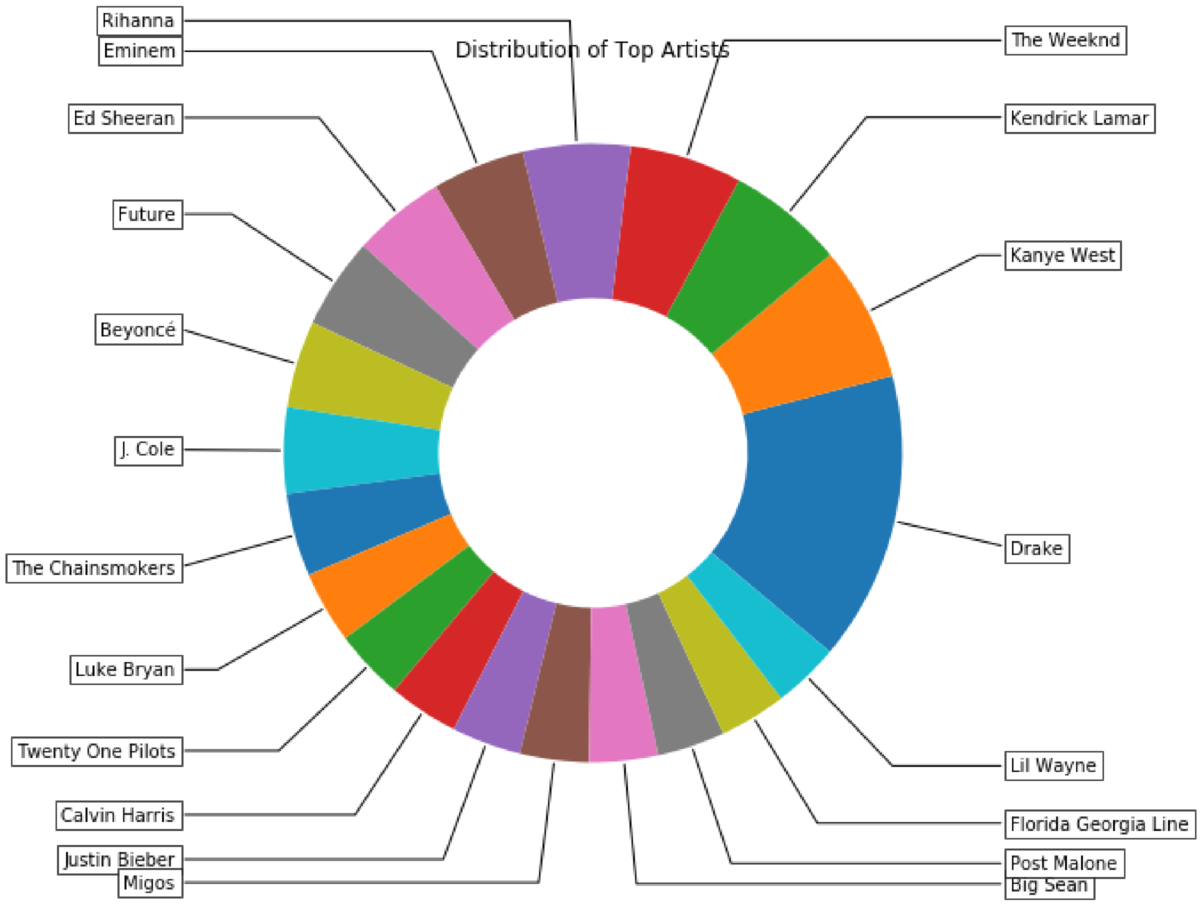


In [14]:
```
# Median Playlist length and number of artists per playlist
print(np.percentile(playlist_lens, q=50))
print(np.percentile(artists_perplay, q=50))

plt.hist(artists_perplay, bins=30)
plt.xlabel('# of Artists Therein')
plt.ylabel('Playlists')
plt.title(f'Unique artists in playlists in the first {percent_data}% of
 the data')
plt.show()
```

```
50.0
30.0
```



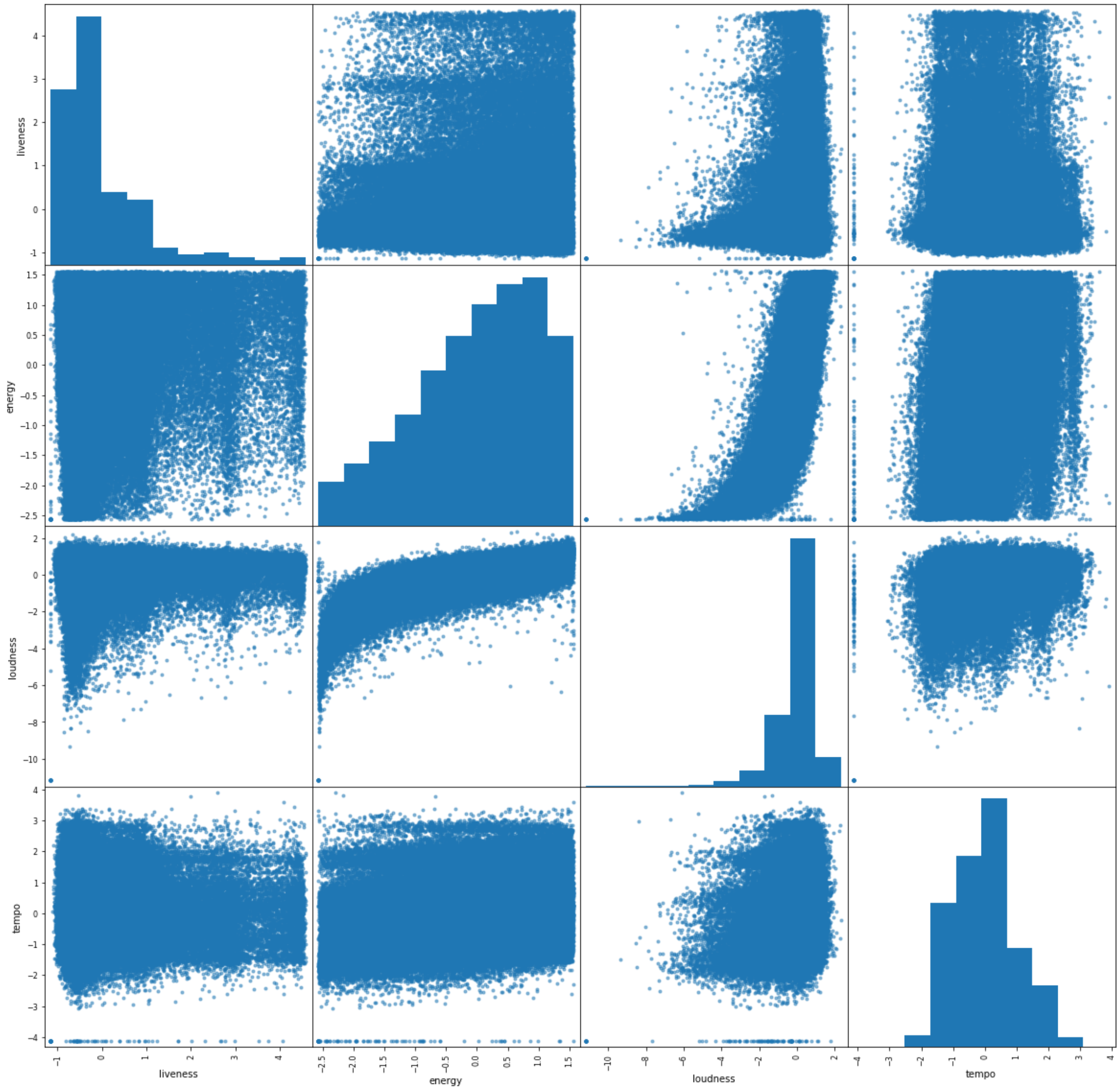
In [15]:
```
# Display a donut plot of the top artists
fig, ax = plt.subplots(figsize=(12, 8), subplot_kw=dict(aspect="equal"))

sorted_counts = [(k, artist_counts[k]) for k in sorted(artist_counts, ke
y=artist_counts.get, reverse=True)]
num_to_show = 20

artists = [el[0] for el in sorted_counts[:num_to_show]]
counts = [el[1] for el in sorted_counts[:num_to_show]]

wedges, texts = ax.pie(counts, wedgeprops=dict(width=0.5), startangle=-4
0)

bbox_props = dict(boxstyle="square,pad=0.3", fc="w", ec="k", lw=0.72)
kw = dict(xycoords='data', textcoords='data', arrowprops=dict(arrowstyle
="-"),
          bbox=bbox_props, zorder=0, va="center")

for i, p in enumerate(wedges):
    ang = (p.theta2 - p.theta1)/2. + p.theta1
    y = np.sin(np.deg2rad(ang))
    x = np.cos(np.deg2rad(ang))
    horizontalalignment = {-1: "right", 1: "left"}[int(np.sign(x))]
    connectionstyle = "angle,angleA=0,angleB={}".format(ang)
    kw["arrowprops"].update({"connectionstyle": connectionstyle})
    ax.annotate(artists[i], xy=(x, y), xytext=(1.35*np.sign(x), 1.4*y),
                 horizontalalignment=horizontalalignment, **kw)

ax.set_title("Distribution of Top Artists")

plt.show()
```



These graphs and statistics explore the features of the million playlist dataset. The first two show the number of songs that are played in the The second two show the number of unique songs and artists within playlists.

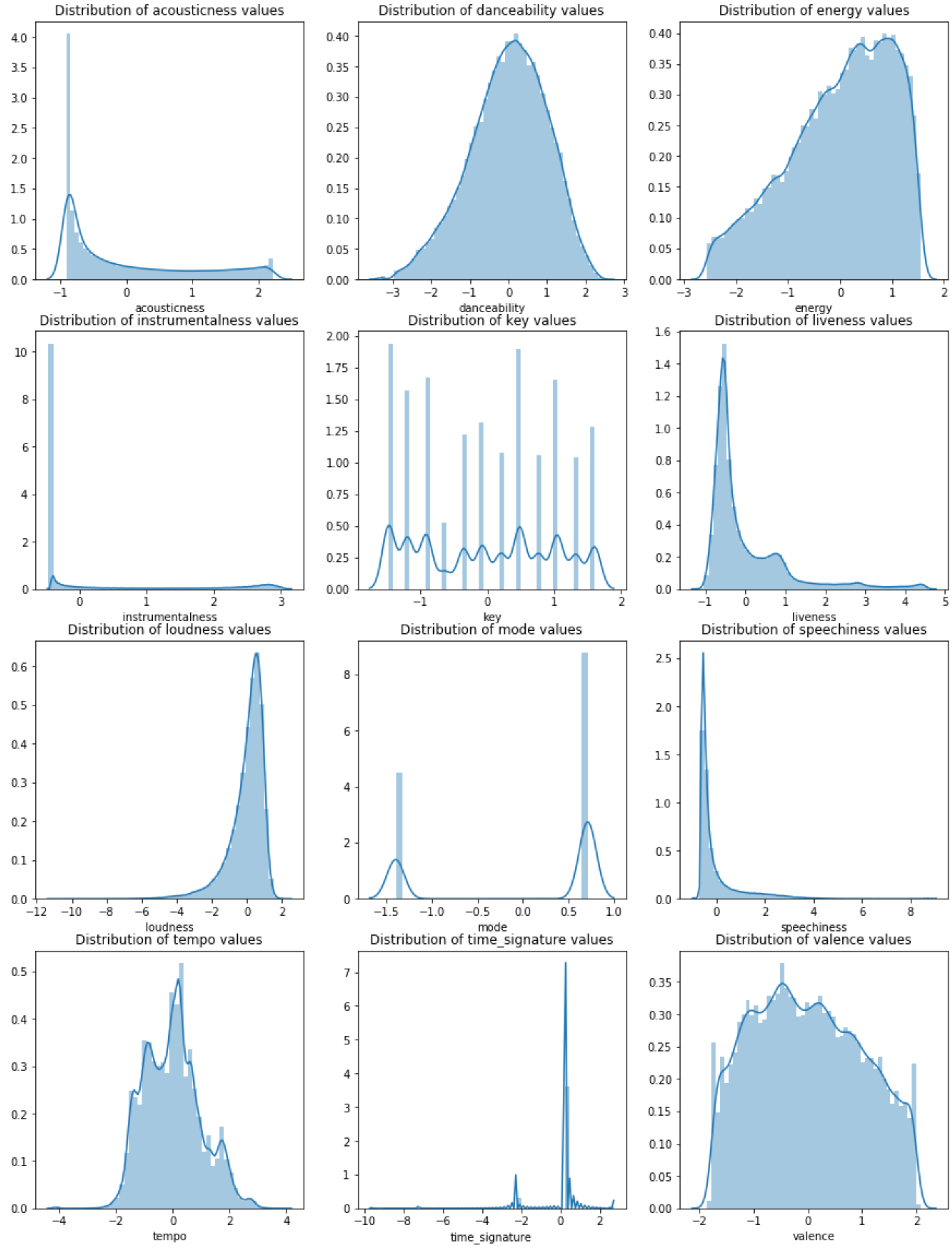
In [16]:
```
# More graphs and charts, both for MPD and for Spotify audio features
col_names = ['acousticness', 'danceability', 'energy', 'instrumentalnes
s',
             'key', 'liveness', 'loudness', 'mode', 'speechiness',
             'tempo', 'time_signature', 'valence']
audio_df_scaled = pd.DataFrame(audio_scaled, columns=col_names)
```

In [17]:
```
sub_df_audio = audio_df_scaled[['liveness', 'energy', 'loudness', 'temp
o']]
pd.scatter_matrix(sub_df_audio, figsize=(20,20))
plt.show()
```



This scatter matrix is mainly of interest because of the relationship between energy and loudness. The two features seem to be relatively positively correlated, so we will need to be careful of colinearity when we fit models with these predictors. As for the other predictors, we can see that we have a relatively good distribution across the range, which is to be expected because we have scaled this data to have $\mu = 0$ and $\sigma = 1$. This should make it work well when we apply distance based techniques like KMeans and KNN to the data.

In [18]:
```
# General comparison of the scores
fig, axes = plt.subplots(4,3, figsize=(15,20))
axs = np.ravel(axes)

for i, col_name in enumerate(col_names):
    data = audio_df_scaled[col_name]
    sns.distplot(data, ax=axs[i], label=col_name)
    axs[i].set_title(f'Distribution of {col_name} values')

plt.show()
```



These plots are of great interest to us, as they display how our Spotify API features are distributed. This is super important when we apply these features to KMeans and KNN, as outliers or wide spreads could possibly sway the distance metrics inordinantly. However, since we have already scaled our data, most of these plots look relatively normal. There are some features (like valence and tempo) that look much more normal, while others (like loudness and liveness) that are skewed either left or right. Additionally, time signature, mode, and key appear to be discrete values, so we may have to handle those carefully.

Models

We explored a variety of different models and model types, but because the style of problem was different than those we studied in class, we were forced to explore different techniques than those we had seen. Our baseline models use a combination of KMeans Clustering and/or KNN, while the top two performing models are a Markov Walk on a Network and Collaborative Filtering.

Baseline: Song Based KMeans Clustering and KNN

In [ ]:
```
# Cluster songs and build:
# - dict with list of URIs for each cluster
# - dict with each song mapped to its cluster
n_clusters = 75
km_songs = KMeans(n_clusters=n_clusters)
song_clusters = km_songs.fit_predict(audio_scaled)

cluster_to_songs, song_to_cluster = {}, {}
for i, cluster_num in enumerate(song_clusters):
    if cluster_num not in cluster_to_songs:
        cluster_to_songs[cluster_num] = []

    cluster_to_songs[cluster_num].append(uris[i])
    song_to_cluster[uris[i]] = cluster_num

# Save the pickles
with open('cluster_to_songs.pickle', 'wb') as handle:
    pickle.dump(cluster_to_songs, handle, protocol=pickle.HIGHEST_PROTOC
OL)
with open('song_to_cluster.pickle', 'wb') as handle:
    pickle.dump(song_to_cluster, handle, protocol=pickle.HIGHEST_PROTOCO
L)
```

In [19]:
```
# Load the pickles instead of rescraping
with open('./cs109_final_backend/cs109_final_backend/cluster_files/clust
er_to_songs.pickle', 'rb') as f:
    cluster_to_songs = pickle.load(f)
with open('./cs109_final_backend/cs109_final_backend/cluster_files/song_
to_cluster.pickle', 'rb') as f:
    song_to_cluster = pickle.load(f)
```

Explanation:
This clustering model is very much a baseline for the other models. Intuitively, it clusters all songs then for each input of seed songs, finds an the 'closest' songs to those, in some way. They all use the Spotify API audio features to cluster and make predictions, relatively ignoring the MPD aside as input and output uris. The following class has three different predict methods, namely `predict` , `predict2` , and `predict3` . The first simply calculates the most populous cluster among the input data, then randomly samples 500 songs from that cluster. The second tries to match the distribution of input songs more closely, outputting the number of songs in the input per cluster scaled up for a total of 500. Finally, the last method uses the audio features even more, calculating the 'distance' of every song in the same cluster to the 'average' of the input songs, outputting the 500 songs closest to the average in order. Overall, these methods do fairly poorly at matching the held out songs, only retrieving relevant songs fairly rarely. The R-precision of these methods is in the range of 0.005-0.011.

In [20]:
```
class ClusterModel:
    def __init__(self, cluster_to_song, song_to_cluster, audio_dict=None
, n_clusters=20, K=25):
        self.name = 'cluster_model'
        self.n_clusters = n_clusters
        self.cluster_to_song = cluster_to_song
        self.song_to_cluster = song_to_cluster
        self.K = K
        self.audio_dict = audio_dict

    def fit(self, X, y):
        pass

    def predict(self, X):
        predictions = []
        for playlist in X:
            clusters = [self.song_to_cluster[song] for song in playlist
if song in self.song_to_cluster]
            unique, counts = np.unique(clusters, return_counts=True)

            if len(unique) == 0:
                max_cluster_id = np.random.randint(0, len(cluster_to_son
gs))
            else:
                max_cluster_id = unique[np.argmax(counts)]

            max_cluster = self.cluster_to_song[max_cluster_id]
            max_cluster = self.cluster_to_song[max_cluster_id]
            try:
                predicted = np.random.choice(max_cluster, size=500, repl
ace=False)
            except ValueError:
                predicted = max_cluster
            predictions.append(predicted)
        return predictions

    def predict2(self, X):
        predictions = []
        for playlist in X:
            clusters = [self.song_to_cluster[song] for song in playlist
if song in self.song_to_cluster]
            unique, counts = np.unique(clusters, return_counts=True)
            predicted = []
            for cl, count in zip(unique, counts):
                size = count * (500 // self.K)
                cluster = self.cluster_to_song[cl]
                preds = np.random.choice(cluster, size=size, replace=Fal
se)

                predicted.extend(preds)
            predictions.append(predicted)
        return predictions

    def predict3(self, X):
        assert(self.audio_dict is not None)
        predictions = []
        for playlist in X:
```

```
            clusters = [self.song_to_cluster[song] for song in playlist
if song in self.song_to_cluster]
            unique, counts = np.unique(clusters, return_counts=True)

            if len(unique) == 0:
                max_cluster_id = np.random.randint(0, len(cluster_to_son
gs))
            else:
                max_cluster_id = unique[np.argmax(counts)]
            max_cluster = self.cluster_to_song[max_cluster_id]

            avg_feat = self.get_average_features(playlist)
            distances = [(uri, self.distance(avg_feat, self.audio_dict[u
ri])) for uri in max_cluster]
            distances.sort(key=lambda tup: tup[1])

            predictions.append([uri for uri, _ in distances[:500]])
        return predictions

    def get_average_features(self, playlist):
        average_features = None
        for uri in playlist:
            features = self.audio_dict[uri]
            if average_features is None:
                average_features = features
            else:
                average_features = average_features + features
        average_features = average_features / len(playlist)
        return average_features

    def distance(self, audio1, audio2):
        distance = np.sqrt(np.sum((audio1 - audio2) ** 2.0))
        return distance
```

Another baseline: Playlist Based KNN

In [21]:
```
class KNNModel():
    def __init__(self, K=5):
        self.K = K

    def fit(self, playlists):
        self.playlists = playlists
        self.songs = set([y for x in self.playlists for y in x])

        self.mlb = MultiLabelBinarizer(classes=list(self.songs))
        self.matrix = self.mlb.fit_transform(self.playlists)

    def recommendations(self, tracks, n_recs):
        known_tracks = [track for track in tracks if track in self.songs
]
        vector = self.mlb.transform([known_tracks])[0]
        neigh = NearestNeighbors(self.K, algorithm='brute', metric='cosi
ne')
        neigh.fit(self.matrix)
        kneighbors = neigh.kneighbors([vector])

        recs = []
        for i in kneighbors[1][0]:
            recs += self.playlists[i]

        for uri in tracks:
            if uri in recs:
                recs.remove(uri)

        return [uri for uri,_ in Counter(recs).most_common(n_recs)]
```

The K Nearest Neighbors algorithm gives another basic method for making recommendations. In laymans terms, the model simply finds the pre-existing playlists most similar to a set of given tracks and then gives the tracks on those playlists as recommendations. To do this, we create an $N * M$ binary matrix Q representing playlist membership where N is the number of playlists given to train the model and M is the total number of unique songs within those playlists. If playlist i contains song j, $Q_{i,j} = 1$. Every other entry is 0. To generate predictions from a list of tracks, the model converts the list to this format and finds the most similar k playlists in terms cosine distance. All of the tracks from these playlists are counted and returned by rank.

Collaborative Filtering

In [22]:
```
from keras.models import Model
from keras.layers import Embedding, Input, Dense, Concatenate, Flatten

class MLPCFModel():
    def __init__(self, K=5, layers=[60,30]):
        self.K = K
        self.layers = layers
        self.n_layers = len(layers)

    def fit(self, playlists):

        # Restructure data
        self.playlists = playlists
        self.songs = set([y for x in self.playlists for y in x])

        self.n_playlists = len(self.playlists)
        self.n_songs = len(self.songs)

        playlist = np.array([[i] * self.n_songs for i in range(self.n_pl
aylists)]).reshape(-1,1)
        track_uri = np.array(list(range(self.n_songs)) * self.n_playlist
s).reshape(-1,1)

        self.mlb = MultiLabelBinarizer(classes=list(self.songs))
        matrix = self.mlb.fit_transform(self.playlists)
        interaction = matrix.flatten().reshape(-1,1)

        # Build the model
        playlist_input = Input(shape=(1,), dtype='int32', name = 'playli
st_input')
        song_input = Input(shape=(1,), dtype='int32', name = 'song_inpu
t')

        playlist_embedding = Embedding(input_dim = self.n_playlists, out
put_dim = int(self.layers[0]/2), name='playlist_embedding')
        song_embedding = Embedding(input_dim = self.n_songs, output_dim
= int(self.layers[0]/2), name='song_embedding')

        playlist_latent = Flatten()(playlist_embedding(playlist_input))
        song_latent = Flatten()(song_embedding(song_input))

        vector = Concatenate(axis=-1)([playlist_latent, song_latent])

        for i in range(1,self.n_layers):
            layer = Dense(self.layers[i], activation='relu', name = 'lay
er{}'.format(i))
            vector = layer(vector)

        predictions = Dense(1, activation='sigmoid')(vector)

        self.model = Model([playlist_input, song_input], predictions)
        self.model.compile(optimizer='adam',loss= 'mean_absolute_error')

        # Train the model
        self.model.fit([playlist, track_uri], interaction, batch_size=32
, epochs=25, validation_split=0.2)
```

```
    def recommendations(self, tracks, n_recs):
        known_tracks = [track for track in tracks if track in self.songs
]
        songs = [list(self.songs).index(uri) for uri in known_tracks]
        vector = self.mlb.transform([known_tracks])

        x1 = np.array([[i] * len(songs) for i in range(self.n_playlists
)]).reshape(-1,1)
        x2 = np.array(songs * self.n_playlists).reshape(-1,1)

        predictions = np.array([y for x in self.model.predict([x1,x2]) f
or y in x])

        indices = predictions.argsort()[-k:][::-1]

        recs = []
        for i in indices:
            recs += self.playlists[int(i/len(songs))]

        for uri in tracks:
            if uri in recs:
                recs.remove(uri)

        return [uri for uri,_ in Counter(recs).most_common(n_recs)]
```

Network Based Markov Model

This model is a probabilistic one that builds up a network where each vertex represents a song and each edge represents two songs sharing a playlist (where more shared playlists leads to higher weighting). Then for prediction, for each of the input K songs, many one step random walks are taken, and the most popular songs that show up in these walks are then returned as a list of 500 song recommendations (after getting rid of duplicates that are already in the playlist). This model consistently performed the best of our models, although building up a large network takes a significant amount of time and space.

Motivation: The motivation for using a Network/Markov Chain approach was that people will want to put songs together into playlists in a similar manner that other people have put songs together into playlists. Thus, looking at what songs are normally put into playlists together, and how often, should be a good indication of what songs will be put into playlists together at a later date.

In [23]:
```
# Network Building code

NETWORK_FILE_PATH = './cs109_final_backend/cs109_final_backend/network_f
iles/pickled_network.pickle'

with open(NETWORK_FILE_PATH, 'rb') as f:
    NETWORK = pickle.load(f)

def n_top_songs(playlist_songs, network, num_samples=4000, num_top_songs
=500):

    # for if we need to fill in with random songs... (see keyerror)
    all_songs = list(network)

    key_errors = 0
    all_samples = np.array([])
    for song_uri in playlist_songs:
        try :
            sample = np.random.choice(network[song_uri]['songs'], num_sa
mples, p=network[song_uri]['counts'])
            all_samples = np.append(all_samples, sample)
        except KeyError:
            # if we get a key error, just randomly choose 1000 songs and
 add them to the samples
            # this could be fixed with a larger network / training set t
hat has
            # every song on at least one playlist... for now lets use ra
ndomness
            key_errors += 1
            all_samples = np.append(all_samples, random.sample(all_songs
, int(num_samples/4)))


    unique, counts = np.unique(all_samples, return_counts=True)

    counts = counts.astype(float) / np.sum(counts)
    counted_samples = zip(unique, counts)
    counted_samples = [sample for sample in counted_samples if sample[0]
 not in playlist_songs]
    counted_samples = sorted(counted_samples, key=lambda x: x[1], revers
e=True)

    num_to_return = min(num_top_songs, len(counted_samples))

    return counted_samples[:num_to_return]
```

In [24]:
```
def evaluate_network_accuracy(train, test, network, num_predictions=500
):
    print ("starting with {} songs, and trying to find {} songs".format(
len(train), len(test)))
    preds = n_top_songs(train, network, num_top_songs = num_predictions)
    preds = [p[0] for p in preds]
    correct_ratio = len([x for x in preds if x in test])/(1. * len(test
))
    print(correct_ratio)
    return correct_ratio
```

Evaluation

We decided to evaluate our models based on the same metrics used in the Spotify RecSys [contest rules (https://recsys-challenge.spotify.com/rules)](https://recsys-challenge.spotify.com/rules), namely R-Precision (RPrec), Normalized Discounted Cumulative Gain (NDCG), and Recommended Song Clicks (RSC). In the following definitions, G is the set of ground truth tracks representing the held out songs from each playlist and R is the ordered list of recommended songs returned by the recommendation system.

- R-Precision: The metric counts "number of retrieved relevant tracks divided by the number of known relevant tracks," rewarding the total number of retrieved relevant tracks, regardless of order.
$$\text{R-precision} = \frac{|G \cap R_{1:|G|}|}{|G|}$$
- Normalized Discounted Cumulative Gain (NDCG): This metric takes into account the order of the returned songs, rewarding relevant songs placed higher in the returned list. It is calculated as Discounted Cumulative Gain (DCG), divided by the Ideal Discounted Cumulative Gain (IDCG), where the returned songs are ordered perfectly. That calculation looks like:
$$DCG = rel_1 + \sum_{i=2}^{|R|} \frac{rel_i}{\log_2(i+1)}$$
$$IDCG = 1 + \sum_{i=2}^{|G|} \frac{1}{\log_2(i+1)}$$
$$NDCG = \frac{DCG}{IDCG}$$
- Recommended Songs Clicks (RSC): This measures how many "clicks" a Spotify user would need to find the first relevant song in the recommendations (the first song actually in the rest of the playlist G), where Spotify displays recommended songs in groups of 10. Therefore it's simply finding the first relevant song and returning its position in the list divided by 10 and truncated. Or more formally:
$$\text{clicks} = \left\lfloor \frac{\arg\min_i \{R_i : R_i \in G|\} - 1}{10} \right\rfloor$$

We have implemented these metrics in code below:

In [25]:
```
from math import log2

class Evaluator():
    """Superclass for evaluation functions"""

    def __init__(self, name):
        self.name = name

    def evaluate(self, output, expected):
        """
        Output will be the output of the model for some list of playlist
s
        - Shape of (# playlists, 500)

        Expected will be the held out songs from each playlist
        - List of lists of various sizes

        Note: Each "song" will be the unique spotify uri of a song
        """
        raise NotImplementedError


class RPrecision(Evaluator):
    """
    R-precision measures the number of held out songs correctly
        retrieved by the model output
    """
    def __init__(self):
        Evaluator.__init__(self, 'R-Precision')

    def evaluate(self, output, expected, return_all=False):

        def rprec_one(output_, expected_):
            expected_size = len(expected_)
            common_set = set(output_).intersection(set(expected_))
            common_size = len(common_set)
            if expected_size == 0 or common_size == 0:
                return 0.0
            return 1. * common_size / expected_size

        r_precs = [rprec_one(out, exp) for (out, exp) in zip(output, exp
ected)]
        if return_all:
            return np.mean(r_precs), r_precs
        return np.mean(r_precs)


class NDCG(Evaluator):
    """
    Normalized discounted cumulative gain also takes into
        account how the system ordered the suggestions
    """
    def __init__(self):
        Evaluator.__init__(self, 'NDCG')

    def evaluate(self, output, expected, return_all=False):
```

```
        def ndcg_one(output_, expected_):
            dcg, idcg = 0.0, 0.0

            if len(output_) == 0 or len(expected_) == 0:
                return 0.0

            expected_ = set(expected_)
            for i in range(len(output_)):
                # Prediction DCG
                if output_[i] in expected_:
                    if i == 0:
                        dcg += 1.0
                    else:
                        dcg += 1.0 / log2(i + 2.0)

                if i < len(expected_):
                    if i == 0:
                        idcg += 1.0
                    else:
                        idcg += 1.0 / log2(i + 2.0)

            return dcg / idcg

        precs = [ndcg_one(out, exp) for (out, exp) in zip(output, expect
ed)]
        if return_all :
            return np.mean(precs), precs
        else :
            return precs


class RSC(Evaluator):
    """
    Recommended Song Clicks measures how many times a user
    would have to click through the suggestions to find a song that
    was a ground truth song
    """
    def __init__(self):
        Evaluator.__init__(self, 'RSC')

    def evaluate(self, output, expected, return_all=False):

        def rsc_one(output_, expected_):
            if len(output_) == 0 or len(expected_) == 0:
                return 51

            output_len = len(output_)
            expected_ = set(expected_)
            for i in range(output_len):
                if output_[i] in expected_:
                    return i//10
            return 51

        scores = [rsc_one(out, exp) for (out, exp) in zip(output, expect
ed)]
        if return_all :
```

```
            return np.mean(scores), scores
        else :
            return np.mean(scores)
```

In [26]:
```
def evaluate_model(output, expected, title=''):
    r_prec = RPrecision().evaluate(output, expected)
    ndcg = NDCG().evaluate(output, expected)
    rsc = RSC().evaluate(output, expected)
    print("{}: R-Precision: {}, NCDG: {}, RSC: {}".format(title, r_prec,
 ndcg, rsc))

def build_evaluation_dataset(start, blocks = 1, n_predictors=10, min_rem
aining = 100, max_remaining = 125) :
    """ Build a list of first n song lists, and a list of last total - n
 song lists

    Args:
        start : (int) the starting playlist slice
        blocks : (int) The number of playlist slices to use
        n_predictor : (int) The number of songs to be in the list of pre
dictor lists
        min_remaining : (int) The minimum number of songs remaining on t
he playlist
        max_remaining : (int) The maximum number of songs remaining on t
he playlist

    Returns:
        predictor_songs : ((str list) list) List of predictor song lists
        remainder_songs : ((str list) list) List of remaining songs (the
 ones we're trying to guess)

    """
    f_start = start * 1000
    f_end = start * 1000 + 999
    predictor_songs = []
    remainder_songs = []
    for i in range(blocks):
        with open('./mpd.v1/data/mpd.slice.{}-{}.json'.format(f_start, f
_end)) as f :
            data = json.load(f)

            for playlist in data['playlists'] :
                tracks = [t['track_uri'] for t in playlist['tracks']]
                if len(tracks) >= min_remaining + n_predictors and len(t
racks) <= max_remaining + n_predictors:
                    predict = tracks[:n_predictors]
                    remain = tracks[n_predictors:]

                    predictor_songs.append(predict)
                    remainder_songs.append(remain)


    return predictor_songs, remainder_songs
```

Evaluation of each model:

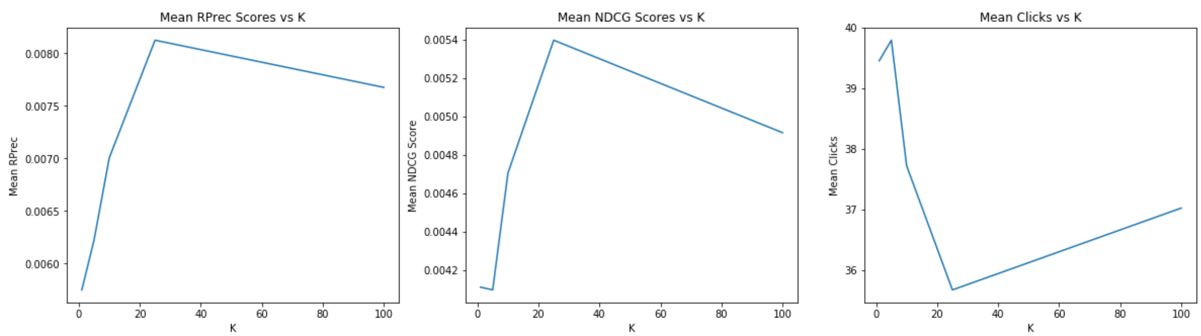
1. Baseline Clustering We follow similar evaluation to the Network Evaluation, on the same test set.

In [27]:
```
def get_accuracies_from_cluster(k, start_block, cluster_to_songs,
                                song_to_cluster, audio_dict, blocks=10,
                                min_remaining=50, max_remaining=200):
    """ Builds the prediction/remainder sets for a given k starting
        at slice=start_block and returns a list of the scores based on e
ach metric for
        each test playlist

    Args:
        k : (int) Number of predictor songs
        start_block : (int) The slice number to start at
        cluster_to_songs : (dict) Mapping each cluster to the songs in t
hat cluster
        song_to_cluster : (dict) Reverse mapping from songs -> cluster
        audio_dict : (dict) Mapping uri -> audio features
        blocks : (int) The number of slices to read
        min_remaining : (int) The min number of remaining tracks to allo
w
        max_remaining : (int) The max number of remaining tracks to allo
w

    Returns:
        r2_results : (float list) The Rprec results for each playlist
        ndcg_results : (float list) The NDCG results for each playlist
        rsc_results : (float list) The click scores for each playlist

    """

    # build the prediction/remainder data
    predictors, remainders = build_evaluation_dataset(start_block, block
s=blocks,
                                                      n_predictors=k, mi
n_remaining=min_remaining,
                                                      max_remaining=max_
remaining)


    # get the predictions from the network
    cm = ClusterModel(cluster_to_songs, song_to_cluster, audio_dict)
    predictions = cm.predict(predictors)

    # evaluate the model based on the 3 metrics
    r_prec = RPrecision()
    r2_results = r_prec.evaluate(predictions, remainders, return_all=Tru
e)[1]

    ndcg_eval = NDCG()
    ndcg_results = ndcg_eval.evaluate(predictions, remainders, return_al
l=True)[1]

    rsc_eval = RSC()
    rsc_results = rsc_eval.evaluate(predictions, remainders, return_all=
True)[1]


    return r2_results, ndcg_results, rsc_results
```

In [28]:
```
r2_acc, nd_acc, rsc_res = get_accuracies_from_cluster(5, 10, cluster_to_
songs,
                                song_to_cluster, audio_dict, blocks=2)

r2s = []
nds = []
rscs = []
table = PrettyTable()
table.field_names = ['K', 'Mean RPrec', 'Mean NCDG', 'Mean Clicks']
for k in [1,5,10,25,100] :
    r2_acc, nd_acc, rsc_res = get_accuracies_from_cluster(k, 100, cluste
r_to_songs,
                                song_to_cluster, audio_dict, blocks=1)
    r2s.append(r2_acc)
    nds.append(nd_acc)
    rscs.append(rsc_res)

    table.add_row([k, round(np.mean(r2_acc), 4), round(np.mean(nd_acc),4
), round(np.mean(rsc_res),4)])


print(table)
```

```
+-----+------------+-----------+-------------+
|  K  | Mean RPrec | Mean NCDG | Mean Clicks |
+-----+------------+-----------+-------------+
|  1  |   0.0058   |   0.0041  |   39.4533   |
|  5  |   0.0062   |   0.0041  |   39.7927   |
|  10 |   0.007    |   0.0047  |   37.7231   |
|  25 |   0.0081   |   0.0054  |    35.679   |
| 100 |   0.0077   |   0.0049  |    37.027   |
+-----+------------+-----------+-------------+
```

In [29]:
```
x = [1,5,10,25, 100]
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(20,5))

sns.lineplot(x, [np.mean([x]) for x in r2s], ax=ax1)
ax1.set_title('Mean RPrec Scores vs K')
ax1.set_xlabel('K')
ax1.set_ylabel('Mean RPrec')

sns.lineplot(x, [np.mean([x]) for x in nds], ax=ax2)
ax2.set_title('Mean NDCG Scores vs K')
ax2.set_xlabel('K')
ax2.set_ylabel('Mean NDCG Score')

sns.lineplot(x, [np.mean([x]) for x in rscs], ax=ax3)
ax3.set_title('Mean Clicks vs K')
ax3.set_xlabel('K')
ax3.set_ylabel('Mean Clicks')

plt.show()
```



Because this model is meant to be very much a baseline, these metrics confirm our assumption that it does only a little better than random chance. While it regularly finds some songs (generally no more than 1 or 2) from the held out songs, the random chance model we tested nearly never retrieved a relevant song. We thought this would be a good baseline as well as giving us a chance to apply some models we learned in class, whereas the two better models we implemented were extensions that we didn't cover at all this semester.

2. Network:

Letting K be the number of seed songs, and S be the number of remaining songs that we are trying to predict, we will evaluate the network model as follows.

We will use $k = [1, 5, 10, 25, 100]$ (spotify challenge requirements) while setting $25 \leq S \leq 200$ in order to keep the number of tracts to predict slightly more consistant in order to ensure that the changing numer of tracks doesn't affect the accuracy as much as with an even larger range. The number of predicted songs will be a constant 500 as in the official RecSys Challenge.

In [30]:
```
def get_accuracies_from_network(k, network, start_block, blocks=10,
                                min_remaining=25, max_remaining=200,
                                num_samples = 1000, num_top_songs=500):
    """ Builds the prediction/remainder sets for a given k starting
        at slice=start_block and returns a list of the scores based on e
ach metric for
        each test playlist

    Args:
        k : (int) Number of predictor songs
        start_block : (int) The slice number to start at
        blocks : (int) The number of slices to read
        min_remaining : (int) The min number of remaining tracks to allo
w
        max_remaining : (int) The max number of remaining tracks to allo
w
        num_samples  : (int) The number of samples to take from each pre
dictor track
        num_top_songs : (int) The number of song predictions to return

    Returns:
        r2_results : (float list) The Rprec results for each playlist
        ndcg_results : (float list) The NDCG results for each playlist
        rsc_results : (float list) The click scores for each playlist

    """

    # build the prediction/remainder data
    predictors, remainders = build_evaluation_dataset(start_block, block
s=blocks,
                                                      n_predictors=k, mi
n_remaining=min_remaining,
                                                      max_remaining=max_
remaining)


    # get the predictions from the network
    predictions = []
    for i in range(len(predictors)):
        p = [s[0] for s in n_top_songs(predictors[i], NETWORK,
                                       num_samples=num_samples, num_top_
songs=num_top_songs)]
        predictions.append(p)

    # evaluate the model based on the 3 metrics
    r_prec = RPrecision()
    r2_results = r_prec.evaluate(predictions, remainders, return_all=Tru
e)[1]

    ndcg_eval = NDCG()
    ndcg_results = ndcg_eval.evaluate(predictions, remainders, return_al
l=True)[1]

    rsc_eval = RSC()
    rsc_results = rsc_eval.evaluate(predictions, remainders, return_all=
True)[1]
```

```
    return r2_results, ndcg_results, rsc_results
```

In [ ]:

```
# sns.distplot(get_accuracies_from_network(5, NETWORK, 100))
# Evaluate the network model for the given k values, using 2 1000 playli
st
# slices, starting at slice 100. (The model was build from slices 2-15)
r2s = []
nds = []
rscs = []
table = PrettyTable()
table.field_names = ['K', 'Mean RPrec', 'Mean NCDG', 'Mean Clicks']
for k in [1,5,10,25,100] :
    r2_acc, nd_acc, rsc_res = get_accuracies_from_network(k, NETWORK, 10
0, blocks=1)
    r2s.append(r2_acc)
    nds.append(nd_acc)
    rscs.append(rsc_res)

    table.add_row([k, round(np.mean(r2_acc), 4), round(np.mean(nd_acc),4
), round(np.mean(rsc_res),4)])


print(table)
```

Above we can start to see trends in the various evaluation methods. It's important to note that because of constraints on local processing power, the network does not necessarily have nodes for all of the test songs which is hurting performance, but a natural drawback of the Markov-chain approach: namely the model itself is quite large. We counter this by randomly sampling songs from the network with equal weight whenever a seed song is not in the network.

Looking at the performance of the network approach based on the three scoring systems, we can see that the model seems to perform best for the $RPrec$ score and $NCDG$ score methods using $K = 25$ seed songs, while the mean clicks required to find a relevant song is best with $K = 100$ seed songs.

Furthermore, while we do not have access to the official test sets used by Spotify, we can start to see the benefits and drawbacks of the Markov-chain based model. The Markov-chain based model seems to perform best (relative to the other metrics) on the $RPrec$ score. The best performing Spotify challenge contestant had a score of $RPrec = .224$ on the official test set, whereas, depending on the K value, our model had a mean $RPrec \in \{0.17, 0.28, 0.32, 0.28\}$. This should of course be taken with a grain of salt as our model most likely would not have had the best performance in the challenge, but does signal that it is a reasonably good approach.

Moving on to mean NCDG score, which takes into account the actual ranking of importance of the predictions into account, our model falls much closer to the middle of the pack in the RecSys leaderboards (again, this is a tough comparison given that we don't have the test set used in the challenge). With scores in the low 0.2 range, (roughly ~60th/110 in the actual challenge) we can start to see the drawbacks of the Markov model.

Lastly our mean Click Score again places the model in the relative middle of the pack for the RecSys leaderboards. Where exactly is unclear because there is no info on the test data and what the distribution of K was, however, it seems that the Markov-chain model can hold its own.

But why does the Markov-chain model apparently do so much better with the $RPrec$ scoring method than the others? At its base, this makes sense because the whole idea of the model is that tracks that people put together on their own will likely be put together by people again. The one step random walk should do a fairly good job of getting the most likely songs as a set, but the ranking system of appearances in the sample set does not seem to perform as well as the other Spotify challenge methods for ensuring the most relevant songs are given first. This would suggest a need to work on the weighting system used by the algorithm. The mean Clicks Score also shows the issues as with the NCDG score, namely the member songs are not always ranked as the most relevant.

All and all, however, it seems like this model would do an ok job of keeping up with the pack in the actual RecSys challenge based on the limited information that we have.

In [ ]:
```
x = [1,5,10,25, 100]
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(20,5))

sns.lineplot(x, [np.mean([x]) for x in r2s], ax=ax1)
ax1.set_title('Mean RPrec Scores vs K')
ax1.set_xlabel('K')
ax1.set_ylabel('Mean RPrec')

sns.lineplot(x, [np.mean([x]) for x in nds], ax=ax2)
ax2.set_title('Mean NDCG Scores vs K')
ax2.set_xlabel('K')
ax2.set_ylabel('Mean NDCG Score')

sns.lineplot(x, [np.mean([x]) for x in rscs], ax=ax3)
ax3.set_title('Mean Clicks vs K')
ax3.set_xlabel('K')
ax3.set_ylabel('Mean Clicks')

plt.show()
```

Above: Here we can see the same mean scores discussed above, but in a graphical form. The R2 and NDCG scores seem to improve as K (the number of seed tracks) increases, and then decreases. While the Mean Clicks score improves, then gets worse, and then improves again.

In [ ]:
```
# General comparison of the scores
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(15,5))

sns.kdeplot(r2s[0], label='K=1', ax=ax1)
sns.kdeplot(r2s[1], label='K=5', ax=ax1)
sns.kdeplot(r2s[2], label='K=10', ax=ax1)
sns.kdeplot(r2s[3], label='K=25', ax=ax1)
sns.kdeplot(r2s[4], label='K=100', ax=ax1)
ax1.set_title('Distribution of RPrec Scores by K')

sns.kdeplot(nds[0], label='K=1', ax=ax2)
sns.kdeplot(nds[1], label='K=5', ax=ax2)
sns.kdeplot(nds[2], label='K=10', ax=ax2)
sns.kdeplot(nds[3], label='K=25', ax=ax2)
sns.kdeplot(nds[4], label='K=100', ax=ax2)
ax2.set_title('Distribution of NDCG Scores by K')

sns.kdeplot(rscs[0], label='K=1', ax=ax3)
sns.kdeplot(rscs[1], label='K=5', ax=ax3)
sns.kdeplot(rscs[2], label='K=10', ax=ax3)
sns.kdeplot(rscs[3], label='K=25', ax=ax3)
sns.kdeplot(rscs[4], label='K=100', ax=ax3)
ax3.set_title('Distribution of Click Scores by K')
ax3.set_xlabel('')

plt.show()
```

Above: Here we can see the distribution of each of the three metrics colored by K. Though hard ot pick out the indivual distributions, it shows how they move together in general. To look at something a little easier to interpret, lets look at the distributions for each scoring metric by K below.

In [ ]:
```
ks = [1,5,10,25,100]
fig, axes = plt.subplots(5,3, figsize=(15,30))
for i in range(5) :
    ax1 = axes[i,0]
    ax2 = axes[i,1]
    ax3 = axes[i,2]

    sns.distplot(r2s[i], ax=ax1)
    ax1.set_title('RPrec Scores | K = {}'.format(ks[i]))
    ax1.set_xlabel('RPrec Score')

    sns.distplot(nds[i], ax=ax2)
    ax2.set_title('NDCG Scores | K = {}'.format(ks[i]))
    ax2.set_xlabel('NDCG Score')

    sns.distplot(rscs[i], ax=ax3)
    ax3.set_title('Click Scores | K = {}'.format(ks[i]))
    ax3.set_xlabel('Required Number of Clicks')
```

Above: In this more in-depth view of the scoring methods by K value we can see the mean trends discussed earlier but in more detail. In terms of $RPrec$ score we can watch the distribution shift right showing the improved RPrec score as K increase, but then shift left again once $K = 100$. We see a similar result in the $NDCG$ scores. Interestingly we can see in the click scores that there are two main 'bumps' in the distribution. The first is around 1, showing that most times there is relevant song on the first page or in the first couple pages for the most part. Then there is another bump around 51 (the max value allowed per the SysRec evaluation specs), which shows that sometimes the reccomendations contain none of the expected songs. As K increases, however, we can see that the bump at 1 grows higher and higher, while the bump at around 51 shrinks showing the overal improvement in the predictions (based on click score) as K increases.

3. Playlist based KNN

In [32]:
```
def get_accuracies_from_KNN(k, knn_model, start_block, blocks=10,
                            min_remaining=25, max_remaining=200, num_top
_songs=500):
    """ Builds the prediction/remainder sets for a given k starting
        at slice=start_block and returns a list of the scores based on e
ach metric for
        each test playlist

    Args:
        k : (int) Number of predictor songs
        start_block : (int) The slice number to start at
        blocks : (int) The number of slices to read
        min_remaining : (int) The min number of remaining tracks to allo
w
        max_remaining : (int) The max number of remaining tracks to allo
w
        num_samples  : (int) The number of samples to take from each pre
dictor track
        num_top_songs : (int) The number of song predictions to return

    Returns:
        r2_results : (float list) The Rprec results for each playlist
        ndcg_results : (float list) The NDCG results for each playlist
        rsc_results : (float list) The click scores for each playlist

    """

    predictors, remainders = build_evaluation_dataset(start_block, block
s=blocks,
                                                       n_predictors=k, mi
n_remaining=min_remaining,
                                                       max_remaining=max_
remaining)

    predictions = []
    for predictor in predictors:
        p = knn_model.recommendations(predictor,num_top_songs)
        predictions.append(p)

    r_prec = RPrecision()
    r2_results = r_prec.evaluate(predictions, remainders, return_all=Tru
e)[1]

    ndcg_eval = NDCG()
    ndcg_results = ndcg_eval.evaluate(predictions, remainders, return_al
l=True)[1]

    rsc_eval = RSC()
    rsc_results = rsc_eval.evaluate(predictions, remainders, return_all=
True)[1]


    return r2_results, ndcg_results, rsc_results
```

In [ ]:
```
playlists = build_evaluation_dataset(2, blocks=2,
                                     n_predictors=1, min_remaining=25,
                                     max_remaining=200)[0]
knn_model = KNNModel(100)
knn_model.fit(playlists)

r2s = []
nds = []
rscs = []
table = PrettyTable()
table.field_names = ['K', 'Mean RPrec', 'Mean NCDG', 'Mean Clicks']
for k in [1,5,10,25,100] :
    r2_acc, nd_acc, rsc_res = get_accuracies_from_KNN(k, knn_model, 100,
 blocks=1)
    r2s.append(r2_acc)
    nds.append(nd_acc)
    rscs.append(rsc_res)

    table.add_row([k, round(np.mean(r2_acc), 4), round(np.mean(nd_acc),4
), round(np.mean(rsc_res),4)])


print(table)
```

In [ ]:
```
x = [1,5,10,25, 100]
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(20,5))

sns.lineplot(x, [np.mean([x]) for x in r2s], ax=ax1)
ax1.set_title('Mean RPrec Scores vs K')
ax1.set_xlabel('K')
ax1.set_ylabel('Mean RPrec')

sns.lineplot(x, [np.mean([x]) for x in nds], ax=ax2)
ax2.set_title('Mean NDCG Scores vs K')
ax2.set_xlabel('K')
ax2.set_ylabel('Mean NDCG Score')

sns.lineplot(x, [np.mean([x]) for x in rscs], ax=ax3)
ax3.set_title('Mean Clicks vs K')
ax3.set_xlabel('K')
ax3.set_ylabel('Mean Clicks')
plt.show()
```

In [ ]:
```
# General comparison of the scores
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(15,5))

sns.kdeplot(r2s[0], label='K=1', ax=ax1)
sns.kdeplot(r2s[1], label='K=5', ax=ax1)
sns.kdeplot(r2s[2], label='K=10', ax=ax1)
sns.kdeplot(r2s[3], label='K=25', ax=ax1)
sns.kdeplot(r2s[4], label='K=100', ax=ax1)
ax1.set_title('Distribution of RPrec Scores by K')

sns.kdeplot(nds[0], label='K=1', ax=ax2)
sns.kdeplot(nds[1], label='K=5', ax=ax2)
sns.kdeplot(nds[2], label='K=10', ax=ax2)
sns.kdeplot(nds[3], label='K=25', ax=ax2)
sns.kdeplot(nds[4], label='K=100', ax=ax2)
ax2.set_title('Distribution of NDCG Scores by K')

sns.kdeplot(rscs[0], label='K=1', ax=ax3)
sns.kdeplot(rscs[1], label='K=5', ax=ax3)
sns.kdeplot(rscs[2], label='K=10', ax=ax3)
sns.kdeplot(rscs[3], label='K=25', ax=ax3)
sns.kdeplot(rscs[4], label='K=100', ax=ax3)
ax3.set_title('Distribution of Click Scores by K')
ax3.set_xlabel('')

plt.show()
```

In [ ]:
```
ks = [1,5,10,25,100]
fig, axes = plt.subplots(5,3, figsize=(15,30))
for i in range(5) :
    ax1 = axes[i,0]
    ax2 = axes[i,1]
    ax3 = axes[i,2]

    sns.distplot(r2s[i], ax=ax1)
    ax1.set_title('RPrec Scores | K = {}'.format(ks[i]))
    ax1.set_xlabel('RPrec Score')

    sns.distplot(nds[i], ax=ax2)
    ax2.set_title('NDCG Scores | K = {}'.format(ks[i]))
    ax2.set_xlabel('NDCG Score')

    sns.distplot(rscs[i], ax=ax3)
    ax3.set_title('Click Scores | K = {}'.format(ks[i]))
    ax3.set_xlabel('Required Number of Clicks')
```

Because this model was more of a baseline for the better two, the scores are significantly worse, yet is on the same level of performance as KNN based on audio features. This is likely because it is able to take into account the structure of the playlist, and so it can better find direct relationships between certain songs without relying to finding those through the secondary audio features that may not be totally accurate. However, it even though it was trained on approximately the same amount of input data as the other models, it wasn't able to capture much information from that training data. Our best guess is that the cosine similarity metric wasn't very accurate, especially for small K, as the input vector is very sparse (only k ones out of hundreds of thousands of songs). Therefore, when searching for 'close' playlists, it isn't able to find relevant playlists very well.

4. Neural CF

In [35]:
```
def get_accuracies_from_CF(k, cf_model, start_block, blocks=10,
                           min_remaining=25, max_remaining=200, num_top_
songs=500):
    """ Builds the prediction/remainder sets for a given k starting
        at slice=start_block and returns a list of the scores based on e
ach metric for
        each test playlist

    Args:
        k : (int) Number of predictor songs
        start_block : (int) The slice number to start at
        blocks : (int) The number of slices to read
        min_remaining : (int) The min number of remaining tracks to allo
w
        max_remaining : (int) The max number of remaining tracks to allo
w
        num_samples  : (int) The number of samples to take from each pre
dictor track
        num_top_songs : (int) The number of song predictions to return

    Returns:
        r2_results : (float list) The Rprec results for each playlist
        ndcg_results : (float list) The NDCG results for each playlist
        rsc_results : (float list) The click scores for each playlist

    """

    predictors, remainders = build_evaluation_dataset(start_block, block
s=blocks,
                                                      n_predictors=k, mi
n_remaining=min_remaining,
                                                      max_remaining=max_
remaining)

    predictions = []
    for predictor in predictors:
        p = cf_model.recommendations(predictor,num_top_songs)
        predictions.append(p)

    r_prec = RPrecision()
    r2_results = r_prec.evaluate(predictions, remainders, return_all=Tru
e)[1]

    ndcg_eval = NDCG()
    ndcg_results = ndcg_eval.evaluate(predictions, remainders, return_al
l=True)[1]

    rsc_eval = RSC()
    rsc_results = rsc_eval.evaluate(predictions, remainders, return_all=
True)[1]


    return r2_results, ndcg_results, rsc_results
```

In [ ]:
```
playlists = build_evaluation_dataset(80, blocks=1,n_predictors=1, min_re
maining=25, max_remaining=200)[0]
cf_model = MLPCFModel(100)
cf_model.fit(playlists)

r2s = []
nds = []
rscs = []
table = PrettyTable()
table.field_names = ['K', 'Mean RPrec', 'Mean NCDG', 'Mean Clicks']
for k in [1,5,10,25, 100] :
    r2_acc, nd_acc, rsc_res = get_accuracies_from_CF(k, cf_model, 100, b
locks=1)
    r2s.append(r2_acc)
    nds.append(nd_acc)
    rscs.append(rsc_res)

    table.add_row([k, round(np.mean(r2_acc), 4), round(np.mean(nd_acc),4
), round(np.mean(rsc_res),4)])


print(table)
```

In [ ]:
```
x = [1,5,10,25, 100]
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(20,5))

sns.lineplot(x, [np.mean([x]) for x in r2s], ax=ax1)
ax1.set_title('Mean RPrec Scores vs K')
ax1.set_xlabel('K')
ax1.set_ylabel('Mean RPrec')

sns.lineplot(x, [np.mean([x]) for x in nds], ax=ax2)
ax2.set_title('Mean NDCG Scores vs K')
ax2.set_xlabel('K')
ax2.set_ylabel('Mean NDCG Score')

sns.lineplot(x, [np.mean([x]) for x in rscs], ax=ax3)
ax3.set_title('Mean Clicks vs K')
ax3.set_xlabel('K')
ax3.set_ylabel('Mean Clicks')
plt.show()
```

In [ ]:
```
# General comparison of the scores
fig, (ax1, ax2, ax3) = plt.subplots(1,3, figsize=(15,5))

sns.kdeplot(r2s[0], label='K=1', ax=ax1)
sns.kdeplot(r2s[1], label='K=5', ax=ax1)
sns.kdeplot(r2s[2], label='K=10', ax=ax1)
sns.kdeplot(r2s[3], label='K=25', ax=ax1)
sns.kdeplot(r2s[4], label='K=100', ax=ax1)
ax1.set_title('Distribution of RPrec Scores by K')

sns.kdeplot(nds[0], label='K=1', ax=ax2)
sns.kdeplot(nds[1], label='K=5', ax=ax2)
sns.kdeplot(nds[2], label='K=10', ax=ax2)
sns.kdeplot(nds[3], label='K=25', ax=ax2)
sns.kdeplot(nds[4], label='K=100', ax=ax2)
ax2.set_title('Distribution of NDCG Scores by K')

sns.kdeplot(rscs[0], label='K=1', ax=ax3)
sns.kdeplot(rscs[1], label='K=5', ax=ax3)
sns.kdeplot(rscs[2], label='K=10', ax=ax3)
sns.kdeplot(rscs[3], label='K=25', ax=ax3)
sns.kdeplot(rscs[4], label='K=100', ax=ax3)
ax3.set_title('Distribution of Click Scores by K')
ax3.set_xlabel('')

plt.show()
```

In [ ]:
```
ks = [1,5,10,25,100]
fig, axes = plt.subplots(5,3, figsize=(15,30))
for i in range(5) :
    ax1 = axes[i,0]
    ax2 = axes[i,1]
    ax3 = axes[i,2]

    sns.distplot(r2s[i], ax=ax1)
    ax1.set_title('RPrec Scores | K = {}'.format(ks[i]))
    ax1.set_xlabel('RPrec Score')

    sns.distplot(nds[i], ax=ax2)
    ax2.set_title('NDCG Scores | K = {}'.format(ks[i]))
    ax2.set_xlabel('NDCG Score')

    sns.distplot(rscs[i], ax=ax3)
    ax3.set_title('Click Scores | K = {}'.format(ks[i]))
    ax3.set_xlabel('Required Number of Clicks')
```

The collaborative filtering model did decently well. It didn't achieve anywhere close to the scores seen above for the network model, but it improved upon the two baseline models. The main reasons we think it didn't perform as well is because it is only trained on a small number of training examples, as the size of the model required quickly blows up as it gets more training examples. If given more time and space, the model could do reasonably better. However, we would likely need to utilize other tricks or simplifications in order to get the model working better for more training data.

Conclusions and Interpretations

Overall, we have seen that the filtering and network models perform the best, significantly improving over the baseline models using nearest neighbor techniques. Our final models were comparable with some of the top models in the RecSys challenge, so we are very satisfied with our results. If we had more time and computing power, we would have liked to scale both of those models up larger, as they were both limited in terms of their size (the network was trained on about 20000 playlists and ended up being about 7GB while filtering was only able to handle about **HOW MANY PLAYLISTS**). Ideally, we would be able to utilize sklearn's suppoer for sparse matrices to scale up filtering, but we weren't able to finalize that.

Music recommendation in general is a challenging problem, with millions of songs to choose from and a large variety of songs within. More complex techniques like deep RNNs and autoencoders seemed attractive at the beginning of the project, but ultimately weren't feasible for us to complete. This forced us to adapt and implement the fairly different models seen here. Overall, we feel confident in our model's ability to find relevant songs to continue and put together a great playlist.