title: Modelling

nav_include: 5

Modelling

The goal is to create a playlist, starting from one song, as this could be easily extended to starting from multiple songs. Dataset is split into three parts:

- Training
- Validation
- Test

(please see "Splitting Data" page for details)

We use the following three components for modelling:

- Similar songs lookup (recursive)
- Regression model that estimates number of followers (coefficients calculated based on training dataset)
- KMeans clustering on playlist engineered features to add songs from playlists which belong to the same centroid as the "base" playlist. "Top" similar (belonging to the same cluster, ordered by number of followers, descending) playlists are used to supply songs (chosen at random)

`Metrics`:

- Find which songs generated and original playlists have in common and calculate number of songs which code guessed correctly
- Calculate aggregated (engineered) values from songs to generated playlist and come up with (Euclidean) "distance" between generated playlist and original playlist
- Using regression model that was fit on training data, predict number of followers for generated playlist and calculate how different it is from true num_followers
- Sum these metrics together to find the loss but invert number of song matches to make sure smaller metric is better
- Alternative summary metric("metric2" in Modelling Results): we found out that resulting summary metric has large variance. Majority of this variance comes from number of followers estimation. Metric2 includes only inverted number of song matches plus distance of generated playlist's engineered features to the original playlist. This metric has much lower variance

`Metaparameters`

- Number of clusters. We tried 2 / 10 / 50 / 100 cluster splits
- Number of playlists (5) to choose songs from the same cluster. Refers to taking songs from only 5 playlists which came from closest cluster (ordered by num_followers, descending)
- Number of similar songs (10) to fetch at each step. Similar songs are added recursively (i.e. add 10 songs, go through them to find similar songs for each in order) - until there are enough or none can be taken
- Algorithm fills 50% of playlist from similar songs and 50% from clusters. If there were only a few similar songs for the first phase - similar songs are also taken from the list of songs from clusters

It would be great to run through several metaparam variations but processing time is too high to try that out. Only variation of number of clusters was done. There was not enough time to find out if splitting similar songs / playlists should be done not 50 / 50 but in different proportion. After training and validation, it looks like best number of clusters are 2 and 10. On average, 10 clusters seems to be best.

`Execution`

- It turned out that playlist generation is pretty slow - running through 500 playlists takes an hour. Hence, code splits (after shuffling) the input dataset into "batches" and runs analysis in parallel
- Each run (train / validation / test) saves results into separate compressed csv files - by batch and by cluster size metaparameter

`Programmatically`, code is very similar for train / validation and test. Major differences:

In [ ]:
```
#!/usr/bin/env python
# coding: utf-

# Training script

import sys
import datetime
import numpy as np
import pandas as pd
import string
from sklearn.cluster import KMeans
from sklearn.linear_model import LinearRegression
import gzip
import csv
from multiprocessing import Process
from sklearn.utils import shuffle

DATA_DIR="./data/data"

df = pd.read_csv(DATA_DIR + '/pidpos.csv.gz', compression='gzip').drop(['Unname
d: 0'],axis=1)
dfAugSongs = pd.read_csv(DATA_DIR + '/full_aug_songs.csv.gz', compression='gzip
')
dfPlaylists = pd.read_csv(DATA_DIR + '/playlists.csv.gz', compression='gzip')
dfTrain = pd.read_csv(DATA_DIR + '/train_aug_playlists.csv.gz', compression='gz
ip').drop(['Unnamed: 0'],axis=1)
# For validation, added dfValidate = pd.read_csv(DATA_DIR + '/validate_aug_play
lists.csv.gz', compression='gzip').drop(['Unnamed: 0'],axis=1)
# For test: dfTest = pd.read_csv(DATA_DIR + '/test_aug_playlists.csv.gz', compr
ession='gzip').drop(['Unnamed: 0'],axis=1)

dfSim = pd.read_csv(DATA_DIR + '/simsong5.csv.gz', compression='gzip').drop(['U
nnamed: 0'],axis=1)

def addSim(dfSim, cur_set, c_id, num) :
    dfCandidates = dfSim[(dfSim.songid == c_id) | (dfSim.simsongid == c_id)]
    t = dfCandidates.sort_values(by='count', ascending=False).values[0:num, :]
    for i in t :
        id = i[1] if i[0] == c_id else i[0]
        if id in cur_set :
            continue
        cur_set.append(id)
    return cur_set

def getPlAgg(candidate_pl) :
     return [ candidate_pl.danceability.mean(), candidate_pl.energy.mean(),
                    candidate_pl.speechiness.mean(), candidate_pl.acousticness
.mean(),
                    candidate_pl.instrumentalness.mean(), candidate_pl.livenes
s.mean(),
                    candidate_pl.valence.mean(), candidate_pl.duration.mean(),
candidate_pl.key.max(),
                    candidate_pl.loudness.max(),
                    candidate_pl.tempo.max(), candidate_pl.time_signature.max(
) ]# .iloc[0] -> first

def getNumfXy(dfPlSongsAgg) :
    y = dfPlSongsAgg.num_followers
    X = dfPlSongsAgg[['mean_danceability','mean_energy','mean_speechiness','mea
n_acousticness',
                                'mean_instrumentalness', 'mean_liveness', 'mean_v
alence',
                'mean_duration', 'max_key', 'max_loudness', 'max_tempo', 'max
```