---

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:

- Regression model for number of followers is trained based on training dataset and applied for all datasets
- Training is limited to 1000 playlists in each of 6 threads calculated in paralllel. This allows validation and training datasets sizes to be roughly equivalent. Below is the script for training.

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([
'Unnamed: 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='gz
ip')
dfTrain = pd.read_csv(DATA_DIR + '/train_aug_playlists.csv.gz', compressi
on='gzip').drop(['Unnamed: 0'],axis=1)
# For validation, added dfValidate = pd.read_csv(DATA_DIR + '/validate_au
g_playlists.csv.gz', compression='gzip').drop(['Unnamed: 0'],axis=1)
# For test: dfTest = pd.read_csv(DATA_DIR + '/test_aug_playlists.csv.gz',
 compression='gzip').drop(['Unnamed: 0'],axis=1)

dfSim = pd.read_csv(DATA_DIR + '/simsong5.csv.gz', compression='gzip').dr
op(['Unnamed: 0'],axis=1)

def addSim(dfSim, cur_set, c_id, num) :
    dfCandidates = dfSim[(dfSim.songid == c_id) | (dfSim.simsongid == c_i
d)]
    t = dfCandidates.sort_values(by='count', ascending=False).values[0:nu
m, :]
    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.acoust
icness.mean(),
                         candidate_pl.instrumentalness.mean(), candidate_pl.l
iveness.mean(),
                         candidate_pl.valence.mean(), candidate_pl.duration.m
ean(), candidate_pl.key.max(),
```

```
                         candidate_pl.loudness.max(),
                         candidate_pl.tempo.max(), candidate_pl.time_signatur
e.max() ]# .iloc[0] -> first

def getNumfXy(dfPlSongsAgg) :
    y = dfPlSongsAgg.num_followers
    X = dfPlSongsAgg[['mean_danceability','mean_energy','mean_speechines
s','mean_acousticness',
                              'mean_instrumentalness', 'mean_liveness',
'mean_valence',
                 'mean_duration', 'max_key', 'max_loudness', 'max_tempo',
 'max_time_signature']].values
    return (X, y)

def getNumfRegr(dfPlSongsAgg) :
    (X, y) = getNumfXy(dfPlSongsAgg)
    return LinearRegression().fit(X, y)

def scoreNumfRegr(reg, dfValidate) :
    (X, y) = getNumfXy(dfValidate)
    return reg.score()

def getSongsFrom(dfSongMatch, n) :
    return dfSongMatch.sample(n, random_state=0)['track_id'].values.tolis
t()

def getSimSongs(dfSim, song_set, start_num, from_i, n) :
    size = len(song_set)
    while True :
        if size > n or from_i > (size - 1):
            break
        addSim(dfSim, song_set, song_set[from_i], start_num)
        from_i +=1
    song_set = song_set[0: n]
    assert len(set(song_set)) == len(song_set) # no dups expected
    return song_set

cluster_columns = ['mean_danceability','mean_energy','mean_speechiness',
'mean_acousticness',
                          'mean_instrumentalness', 'mean_liveness', 'mean
_valence',
             'mean_duration', 'max_key', 'max_loudness', 'max_tempo', 'ma
x_time_signature']

dfTrain.dropna(inplace=True)
reg = getNumfRegr(dfTrain) # training
dfTrain=shuffle(dfTrain)
# For validation: dfValidation=shuffle(dfTrain)
# For test: dfTest=shuffle(dfTrain)

# start_num : how many similar songs to fetch on each level
def generate_playlists(dfPlSongsAgg, dfPidPosPl, name, reg, try_clusters,
 try_startNum, song_name) :
    num_top_pl = 5 # choose top 5 playlists
    train_song_id = 0 # first song is used to continue playlist
    toCluster = dfPlSongsAgg[cluster_columns].values
    for n_clusters in try_clusters :
```

```
        kmeans = KMeans(n_clusters=n_clusters, random_state=0, n_jobs = -
1).fit(toCluster)
        print("Found", n_clusters, "clusters")
        train_pl = dfPlSongsAgg.playlist_id.values
        for start_num in try_startNum :
            print("start_num", start_num)
            with gzip.open(DATA_DIR + "/result_" + name + "_" + str(n_clu
sters) + "_" + str(start_num) + ".csv.gz",
                           'wt', newline='') as fz:
                writer = csv.writer(fz, delimiter=',')
                writer.writerow(['playlist_id', 'n_clusters', 'start_num'
, 'metric', 'match', 'distance', 'numf',
                                 'diff'])
                for pl_id in train_pl :
                    print("Running playlist", pl_id)
                    try :
                        train_numf = dfPlSongsAgg[dfPlSongsAgg.playlist_i
d == pl_id].num_followers.values
                        target_n = int(dfPlSongsAgg[dfPlSongsAgg.playlist
_id == pl_id].sum_num_tracks.values)
                        (train_agg_info, _) = getNumfXy(dfPlSongsAgg[dfPl
SongsAgg.playlist_id == pl_id])
                        train_song_set = df[df.playlist_id == pl_id].trac
k_id.values
                        root_name = dfAugSongs[dfAugSongs.id == train_son
g_set[train_song_id]].name.values[0]
                        fromsim_n = int(target_n / 2) # metaparam
                        root_id = dfAugSongs[dfAugSongs.name == root_name
].id.values[0]
                        song_set = [train_song_set[0]]
                        addSim(dfSim, song_set, root_id, start_num)
                        # choose start_num - loop to fromsim_n
                        getSimSongs(dfSim, song_set, start_num, 0, fromsi
m_n)
                        # find song aug data and create playlist
                        candidate_pl = dfAugSongs.iloc[song_set, :]
                        candidate_pl_agg = getPlAgg(candidate_pl)
                        p_label = kmeans.predict([candidate_pl_agg])[0]
                        dfPlaylistMatch = dfPlSongsAgg[kmeans.labels_ ==
p_label]
                        dfPlaylistMatchTop = dfPlaylistMatch.sort_values(
by='num_followers', ascending=False).head(
                            num_top_pl)
                        dfSongMatch = pd.merge(dfPidPosPl, dfPlaylistMatc
hTop, left_on='playlist_id', right_on='playlist_id',
                                               how='left').dropna()
                        from_cluster = target_n - fromsim_n
                        if from_cluster > dfSongMatch.shape[0] : # not en
ough songs
                            song_from_pl = getSongsFrom(dfSongMatch, dfSo
ngMatch.shape[0])
                            getSimSongs(dfSim, song_from_pl, start_num, 0
, from_cluster)
                        else :
                            song_from_pl = getSongsFrom(dfSongMatch, from
_cluster)
                        song_play = [*song_set, *song_from_pl]
```

```
                        song_info = dfAugSongs.iloc[song_play, :]
                        # metrics
                        metric_match = list(set(train_song_set) & set(son
g_play))
                        metric_match_n = len(metric_match)
                        song_agg = np.array(getPlAgg(song_info)) # aggreg
ated to playlist
                        dist = (song_agg - train_agg_info)**2
                        dist = np.sum(dist, axis=1)
                        dist = np.sqrt(dist)[0]
                        song_numf = round(reg.predict(song_agg.reshape(1,
 -1))[0])
                        numf_diff = int(np.abs(train_numf - song_numf)[0
])
                        sum_metric = 1.0 / metric_match_n + dist + numf_d
iff
                        writer.writerow([pl_id, n_clusters, start_num, ro
und(sum_metric, 2), metric_match_n,
                                         round(dist, 2), song_numf, numf_
diff])
                    except  Exception as e:
                        print("Ex playlist", pl_id, str(e))

kT = int(dfTrain.shape[0] / 6) # dfTrain was changed to dfValidation / df
Test for validation / test
# For validation, no limit was used as number of records was roughly the
 same
limit = 1000
n_cl = [2,10,50,100]
def func1() :
    print("starting 1")
    dfT = dfTrain.iloc[0 : kT, :].sample(limit, random_state=0)
    dfP = pd.merge(df, dfT, left_on='playlist_id', right_on='playlist_id'
, how='left').dropna()
    print("running 1")
    generate_playlists(dfT, dfP, "train1", reg, try_clusters=n_cl, try_st
artNum=[10])
    print("done 1")
def func2() :
    print("starting 2")
    dfT = dfTrain.iloc[kT : 2*kT, :].sample(limit, random_state=0)
    dfP = pd.merge(df, dfT, left_on='playlist_id', right_on='playlist_id'
, how='left').dropna()
    print("running 2")
    generate_playlists(dfT, dfP, "train2", reg, try_clusters=n_cl, try_st
artNum=[10])
    print("done 2")
def func3() :
    print("starting 3")
    dfT = dfTrain.iloc[2*kT : 3*kT, :].sample(limit, random_state=0)
    dfP = pd.merge(df, dfT, left_on='playlist_id', right_on='playlist_id'
, how='left').dropna()
    print("running 3")
    generate_playlists(dfT, dfP, "train3", reg, try_clusters=n_cl, try_st
artNum=[10])
    print("done 3")
def func4() :
```

```
    print("starting 4")
    dfT = dfTrain.iloc[3*kT : 4*kT, :].sample(limit, random_state=0)
    dfP = pd.merge(df, dfT, left_on='playlist_id', right_on='playlist_id'
, how='left').dropna()
    print("running 4")
    generate_playlists(dfT, dfP, "train4", reg, try_clusters=n_cl, try_st
artNum=[10])
    print("done 4")
def func5() :
    print("starting 5")
    dfT = dfTrain.iloc[4*kT : 5*kT, :].sample(limit, random_state=0)
    dfP = pd.merge(df, dfT, left_on='playlist_id', right_on='playlist_id'
, how='left').dropna()
    print("running 5")
    generate_playlists(dfT, dfP, "train5", reg, try_clusters=n_cl, try_st
artNum=[10])
    print("done 5")
def func6() :
    print("starting 6")
    dfT = dfTrain.iloc[5*kT :, :].sample(limit, random_state=0)
    dfP = pd.merge(df, dfT, left_on='playlist_id', right_on='playlist_id'
, how='left').dropna()
    print("running 6")
    generate_playlists(dfT, dfP, "train6", reg, try_clusters=n_cl, try_st
artNum=[10])
    print("done 6")

p1 = Process(target=func1)
p1.start()
p2 = Process(target=func2)
p2.start()
p3 = Process(target=func3)
p3.start()
p4 = Process(target=func4)
p4.start()
p5 = Process(target=func5)
p5.start()
p6 = Process(target=func6)
p6.start()
p1.join()
p2.join()
p3.join()
p4.join()
p5.join()
p6.join()

print("Training done")
```