CA4015 Music Recommendation Task

Anthony Reidy, 18369643

Nov 28, 2021

CONTENTS

With the growth of music streaming platforms such as Spotify, and Apple music, never before has music has been so accessible. However, the plethora of songs and artists makes it difficult for user's to discover music that relates to their own taste. Recommendation systems seek to providers user's with personalized recommendations in an effort to enhance the user's experience of the platform. As streaming services generate most of their money through subscription plans, the adoption of such models can increase revenue by increased customer retention. The research literature of recommender systems is primarily concerned with increasing performance (i.e. low error (e.g. RMSE) and high precision, recall etc). Although, there are number of other variables that are important when deploying these models in production.

Diversity: There is often a diversity–accuracy dilemma in recommender systems. The tuning of parameters to reach the highest accuracy possible, often leads to recommendations that are highly similar to each other and lack diversity. (Hurley et al., 2011). However, users value recommendations from a diverse range of sources e.g. artists of different genres. Diversity in recommendation systems is used to avoid the overfitting problem as well as increasing the quality of user experiences. For instance, the recent success of TikTok is driven by it's ForYou page. In a recent press conference, Tiktok confirmed avoiding homogenous video streams was one of the KPI's when developing its algorithm (Perez, 2021) .

Labelling: The way in which recommendations are presented also affects user's likeness to utilise them. Beel et al. (2013) conducted a human behavioral study in which recommendations were labelled as either "Sponsored" or "Organic". Recommendation with the former had a lower click-through rate ((5.93%) when compared to the latter (8.86%). They were also recommendations with no label at all. Surprisingly, these performed the best in the study (9.87%).

Serendipity): Serendipity refers to the noveleness or uncertainty of the recommendations and is related to diversity. Again, there is often a trade-off between model performance and serendipity. The purpose of serendipity is two-fold; mitgates against lost interest due to uniform reccomendation sets and improves the underlying algorithm.

Bias: The biases in NLP (Blodgett et al., 2020, Garrido-Muñoz et al., 2021) and computer vision (Buo-lamwini & Gebru, 2018) models are well documented. Likewise, recommender systems can suffer from these same harms as their success often relies on their ability to exploit the collective behavior of users in order to deliver recommendations. Therefore, there is a risk of enforcing hegemonic viewpoints which may marginalize certain groups. However, some bias is desired such as gender bias when recommending textiles (Tsintzou et al, 2018).

Robustness against Fraud: Recommendation systems should be robust against fraud. An example of fraud could be bots designed to increase the perceived popularity of an item, which may cause an algorithm to recommend it more often.

Cold-start: The cold start problem occurs when new items are added that have either none or very little interactions. This is a major problem for collaborative filtering based methods as these algorithms rely on user interactions to make recommendations.

The type of algorithms that seek to solve to solve this use case are extremely diverse. Probabilistic models such as latent sematic model (Hofmann et al., 2004), Bayesian models (Sembium et al., 2018), Markov chain models (Rendle et al., 2010) and Spectral analysis techniques (Marple et al., 1989) all feature predominantly within the scientific work on recommendation systems. In this investigation, we propose a number of approaches for artist recommendation, namely weighted matrix factorization (SGD) and a softmax model. Precision, and recall, viewed in different contexts, will be the metrics used to evaluate the robustness of our approaches. Our recommendation systems attempts to give N number of recommendations to the users where N=15. We also utilise three novel graphs in an attempt to better understand the listening habits of the Last.FM userbase. Note, we provide a "quick links" section at the start of each notebook, to enable fast browsing when running the code. It also describes the general outline of the proceeding notebook.

In this notebook, we perform an initial data exploration to perform transformations & data sanitization checks; acquire rudimentary statistics of the datasets; create exploratory visualizations.

Quick links

- *Download Last.fm Dataset*

Download data

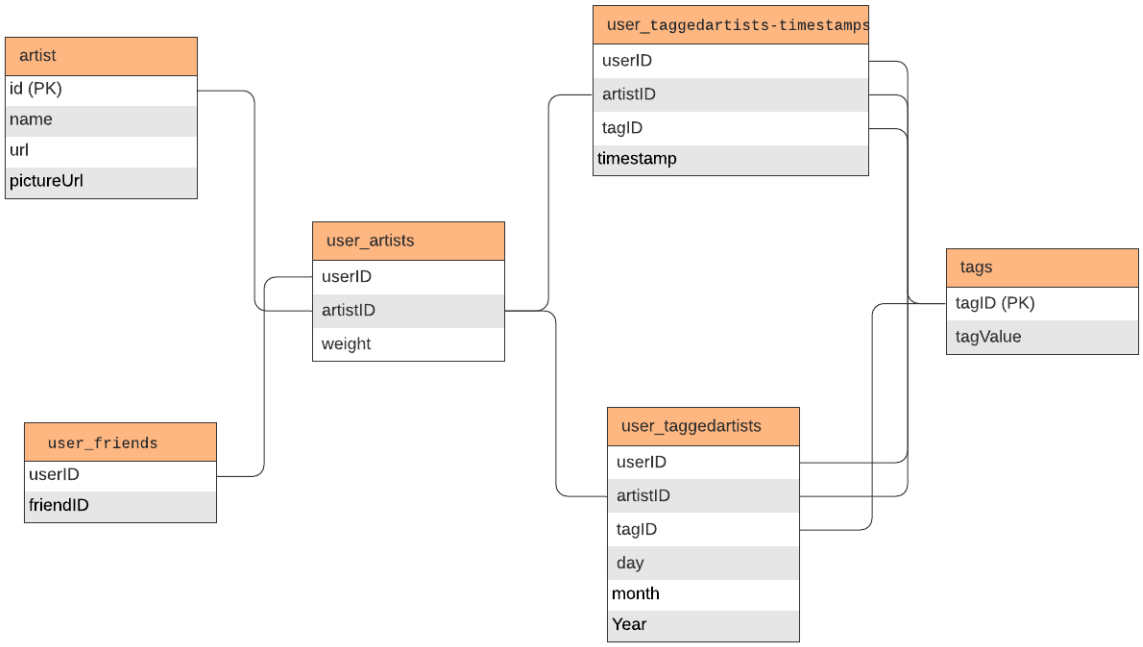
DO NOT RUN THE NEXT CELL AS THE DATA HAS ALREADY BEEN DOWNLOADED

Download the Last.FM dataset from the grouplens website into a *data* folder. This cell does not need to executed again, since the files are already there.

```
#!mkdir data && wget https://files.grouplens.org/datasets/hetrec2011/hetrec2011-
↪lastfm-2k.zip && unzip hetrec2011-lastfm-2k.zip -d data/.
```

Dataset

Fig. 1: ER diagram of the Last.fm dataset



.png

In this investigation, we utilise the last.fm dataset. This dataset, introduced at the 5th ACM Conference on Recommender Systems workshop, details music artist listenings, users specified semantic categories (tags) associated with artists, and social networking information (user -> user friendship relationships). Over 1800 users from the last.fm website were included with 17632 artists also being contained. Please refer to the above ER diagram for more context. The entities in the diagram represent the source .dat files. Unfortunately user's demographic data are not provided in the dataset. This dataset was curated in 2011.

Exploratory data analysis

Data Validation

Import relevant libaries and read in the files as pandas dataframes with the same table names specified in the ER. Minimal transformations are performed at this stage.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from matplotlib import cm
```

```
#from google.colab import drive
#import os
#drive.mount('/content/drive/')
#os.chdir("/content/drive/My Drive/DCU/fouth_year/advanced_machine_learning/music-
 ↪recommodation-system")
```

```
artist = pd.read_csv('data/artists.dat', sep='\t')
tags = pd.read_csv('data/tags.dat',encoding = 'unicode_escape', sep='\t')
user_friends = pd.read_csv('data/user_friends.dat', sep='\t')
user_artists = pd.read_csv('data/user_artists.dat', sep='\t')
user_taggedartists_timestamps = pd.read_csv('data/user_taggedartists-timestamps.dat',␣
 ↪sep='\t')
user_taggedartists = pd.read_csv('data/user_taggedartists.dat', sep='\t')
```

check for null_values and ensure expected schema contraints (e.g. that tagID is unique).

```
print(f'Does artist contain any null values? {artist.isnull().any().any()}')
print(f'Does tags contain any null values? {tags.isnull().any().any()}')
print(f'Does user_friends contain any null values? {user_friends.isnull().any().any()}
 ↪')
print(f'Does user_artists contain any null values? {user_artists.isnull().any().any()}
 ↪')
print(f'Does user_taggedartists_timestamps contain any null values? {user_
 ↪taggedartists_timestamps.isnull().any().any()}')
print(f'Does user_taggedartists contain any null values? {user_taggedartists.isnull().
 ↪any().any()}')
```

```
Does artist contain any null values? True
Does tags contain any null values? False
Does user_friends contain any null values? False
Does user_artists contain any null values? False
Does user_taggedartists_timestamps contain any null values? False
Does user_taggedartists contain any null values? False
```

```
print(artist.info())
#Rename id to artistID
artist.rename({'id':'artistID'}, axis=1,inplace=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17632 entries, 0 to 17631
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   id          17632 non-null  int64
 1   name        17632 non-null  object
 2   url         17632 non-null  object
 3   pictureURL  17188 non-null  object
dtypes: int64(1), object(3)
memory usage: 551.1+ KB
None
```

Not every artist has a pictureUrl attached. Combine the two user_taggedartist* dfs

```
user_taggedartists_merged = pd.merge(user_taggedartists_timestamps,user_taggedartists,
 ↪ on=['userID','artistID','tagID'], how='inner')
print(user_taggedartists_merged['artistID'].nunique())
```

```
12523
```

Validate the uniqueness of IDs and surrogate keys.

```
print(f'Is tagID unique?: {tags["tagID"].is_unique}')
print(f'Is artistID unique?: {artist["artistID"].is_unique}')
user_friends_ids  = [f'{a}_{b}' for a,b in user_friends[['userID', 'friendID']].
 ↪values.tolist()]
print(f'Is the combination of user and friend IDs unique in user_friends?:
 ↪{len(set(user_friends_ids)) == len(user_friends_ids)}')
user_artist_tags_ids  = [f'{a}_{b}_{c}' for a,b,c in user_taggedartists_merged[[
 ↪'userID', 'artistID', 'tagID']].values.tolist()]
print(f'is the combination of user artist and tag IDs unique in user_taggedartists_
 ↪merged?: {len(set(user_artist_tags_ids)) == len(user_artist_tags_ids)}')
```

```
Is tagID unique?: True
Is artistID unique?: True
Is the combination of user and friend IDs unique in user_friends?: True
```

```
is the combination of user artist and tag IDs unique in user_taggedartists_merged?:␣
 ↪True
```

Exploratory visualisation

First, we are interested in how the trends in the top 4 most 'mainstream' genres. (Tags and genres are used interchangeably in this investigation). We define mainstream in terms of the most number of users rather than the listening count. First, we flatten the present relational model into a flat dataframe for ease of use.

```
flatten_df = pd.merge(user_artists, user_taggedartists_merged, on=['userID', 'artistID
↪'], how='outer')
flatten_df = pd.merge(flatten_df, tags, on=['tagID'], how='inner')
flatten_df = pd.merge(flatten_df, artist, on=['artistID'], how='inner')
flatten_df.head(5)
```

```
   userID  artistID   weight  tagID     timestamp  day  month    year  \
0       2        52  11690.0   13.0  1.238537e+12  1.0    4.0  2009.0
1     283        52     68.0   13.0  1.222812e+12  1.0   10.0  2008.0
2     541        52    894.0   13.0  1.193872e+12  1.0   11.0  2007.0
3     697        52    443.0   13.0  1.170284e+12  1.0    2.0  2007.0
4      43        52      NaN   13.0  1.272665e+12  1.0    5.0  2010.0

   tagValue       name                                  url  \
0  chillout  Morcheeba  http://www.last.fm/music/Morcheeba
1  chillout  Morcheeba  http://www.last.fm/music/Morcheeba
2  chillout  Morcheeba  http://www.last.fm/music/Morcheeba
3  chillout  Morcheeba  http://www.last.fm/music/Morcheeba
4  chillout  Morcheeba  http://www.last.fm/music/Morcheeba

                                          pictureURL
0  http://userserve-ak.last.fm/serve/252/46005111...
1  http://userserve-ak.last.fm/serve/252/46005111...
2  http://userserve-ak.last.fm/serve/252/46005111...
3  http://userserve-ak.last.fm/serve/252/46005111...
4  http://userserve-ak.last.fm/serve/252/46005111...
```

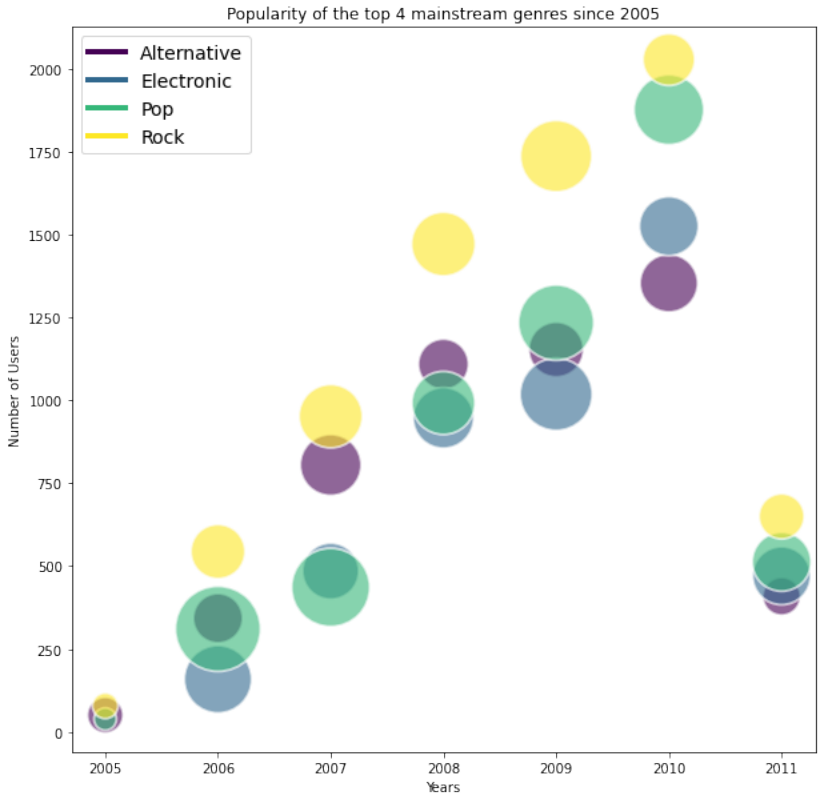
```
bubble_chart_df = flatten_df[flatten_df.year>= 2000]
top_ten_tags = bubble_chart_df['tagValue'].value_counts().head(4).index.tolist()
bubble_chart_df = bubble_chart_df.groupby(['year','tagValue']).agg({
    'tagValue':'count',
    'weight':'mean'
    }).rename({
        'tagValue':'number of users',
        'weight':'avg listening count'}, axis=1).reset_index()
bubble_chart_df = bubble_chart_df[bubble_chart_df['tagValue'].isin(top_ten_tags)]
```

```
def create_bubble_chart(flattend_df):
    """
    Arguments:
        flattend_df: a df containing the number of users and average
            listing count grouped by year and tag value.
    Returns:
        A bubble chart show casing  popularity trends of genres since 2005
    """

    # Conert the categorical variable to numeric
    flattend_df['tagValue_cat']=pd.Categorical(flattend_df['tagValue'])
    tag_dict = dict(zip(flattend_df['tagValue'], flattend_df['tagValue']))

    # Set the figure size
    plt.figure(figsize=(10, 10))

    # bars for legend
    custom_lines = [Line2D([0], [0], color=(0.267004, 0.004874, 0.329415, 1), lw=4),
                Line2D([0], [0], color=(0.190631, 0.407061, 0.556089, 1), lw=4),
```

(continues on next page)

(continued from previous page)

```
                Line2D([0], [0], color=(0.20803 , 0.718701, 0.472873, 1      ), lw=4),
                Line2D([0], [0], color= (0.993248, 0.906157, 0.143936, 1), lw=4)]

    # Bubblechart
    plt.scatter(
        x = flattend_df['year'],
        y = flattend_df['number of users'],
        s=flattend_df['avg listening count'],
        c=flattend_df['tagValue_cat'].cat.codes,
        cmap= cm.get_cmap('viridis', 4),
        alpha=0.6,
        edgecolors="white",
        linewidth=2);

    # Add titles (main and on axis)
    plt.xlabel("Years")
    plt.ylabel("Number of Users")
    plt.title("Popularity of the top 4 mainstream genres since 2005")
    plt.legend(custom_lines, ['Alternative', 'Electronic', 'Pop', 'Rock'], loc="upper␣
↪left", prop={'size': 14})

create_bubble_chart(bubble_chart_df)
```

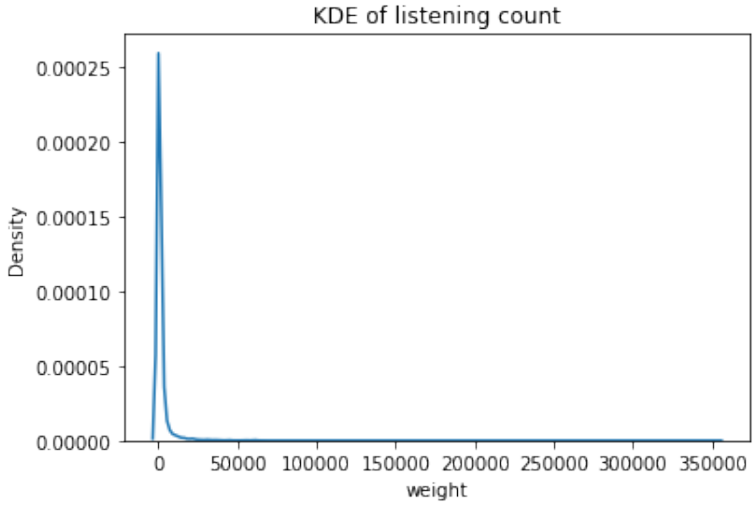


In the bubble chart above, each bubble represents a mainstream genre, its area is proportional to its average listening count in a given year, the colour indicates its genre and the horizontal and vertical positions relate to year (since 2005) and the amount of users, respectively. It is evident that the average listening count and the total number of users was comparatively small in 2005. Last.fm had a smaller user base in 2005 (842 users) with it's number of customers increasing in every year until 2010 (23790 users). The falloff at 2011 is most likely due to the fact that the data was not collected at the end of

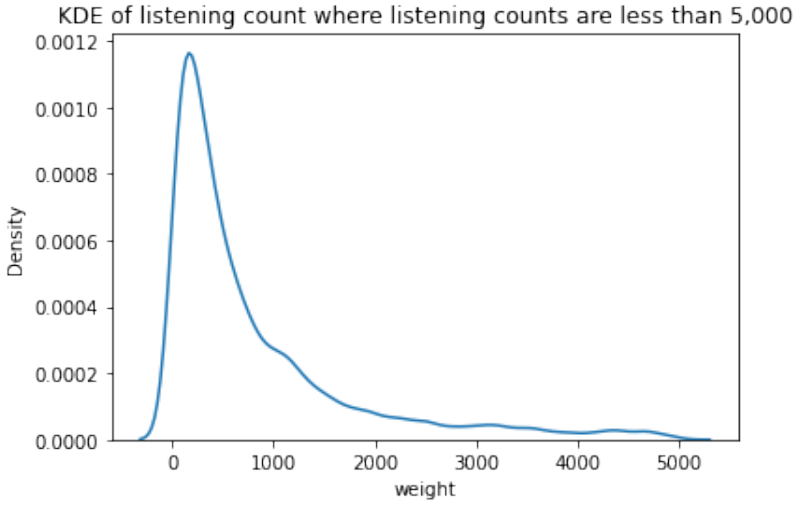
2011. The number of user's a genre has garnered does not necessarily correlate with average listening time. For example, in 2009 Electronic has the highest average listen time but the lowest amount of listeners. This indicates this genre in 2009 was skewed by 'hardcore listeners' who listened longer than the average user in the other examined genres. Rock had the most number of users for 5 out of 7 years. Pop had the top three highest average listing count overall, in 2007, 2006 and 2009, respectively. Finally, we observe the relative plateauing of the alternative genres in the terms of active listeners from 2007 to 2009.

Next, we plot the distribution of listening counts. We observe the number of listens is extremely right-skewed. A majority of users play artists a few hundred times.

```
sns.kdeplot(data=flatten_df, x="weight")
plt.title("KDE of listening count")
```

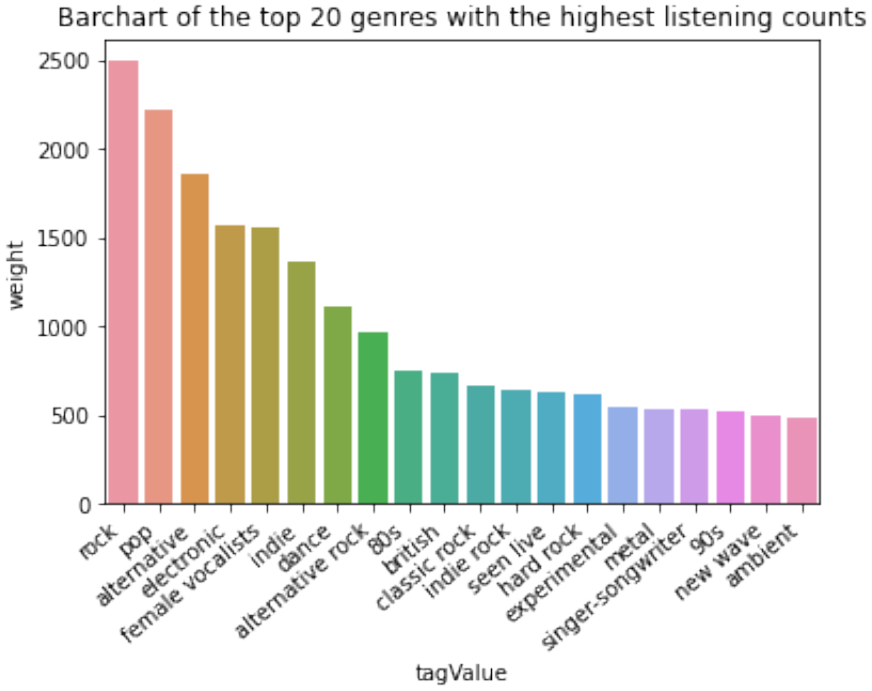



```
sns.kdeplot(data=flatten_df[flatten_df.weight < 5000], x="weight")
plt.title("KDE of listening count where listening counts are less than 5,000")
```



We are also interested in plotting the plotting the most and least number of tags. We observe pop and rock claim the top two spots. As tags are semantical categories by the users, we see a lot of non-sensical tags in the second plot such as "merge". We hypothesis that these may be the title of a song.

```
ax = sns.barplot(x="tagValue", y="weight", data=flatten_df[['tagValue','weight']].
 ↪groupby('tagValue').count().reset_index().sort_values('weight', ascending=False).
 ↪head(20))
ax.set_xticklabels(ax.get_xticklabels(), rotation=40, ha="right")
plt.title("Barchart of the top 20 genres with the highest listening counts")
plt.show()
```

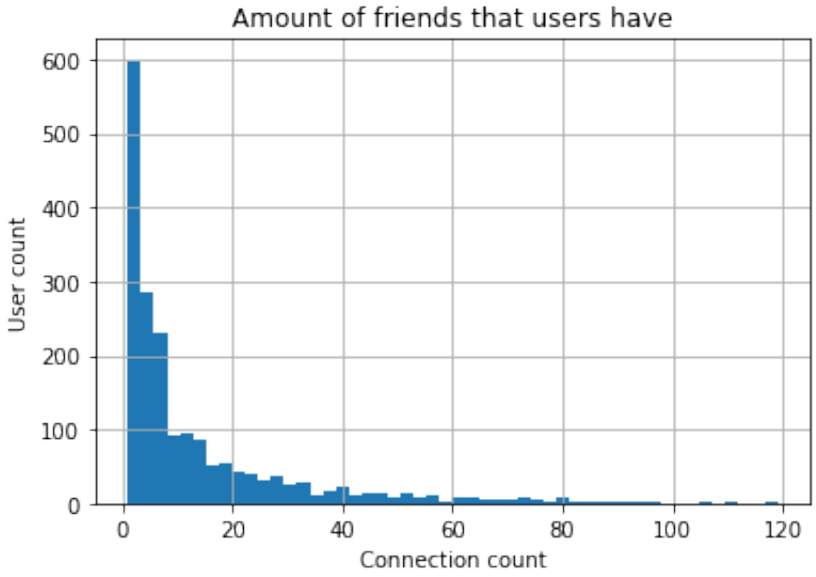



```
print(flatten_df[['tagValue','weight']].groupby('tagValue').count().reset_index().
 ↪sort_values('weight').head(20))
```

```
                                    tagValue  weight
8539                              talib kweli       0
6940                              pra acalmar       0
5311                                macy gray       0
3595                               glitch-hop       0
8154                              sonra dinle       0
5310                             macumba music       0
8158                      sophie ellis-bextor       0
5309                           mactonight fav       0
5308                       mactonight essential       0
3599                       global communication       0
5307                              machine head       0
6025                             nature sounds       0
3602                          glorious tracks       0
2294                                   dejeto       0
8168                                soul-pop       0
5304                                    m0sh       0
8173                            sound collage       0
1301                              bomfunk mcs       0
8174  sound collage psychedelia electronic folk       0
8177                            sound of 2011       0
```

Finally, we conduct a basic visualisation of the friendship relationships. We are interested in the number of connections a user has. We intend to explore this further in the graph analysis notebook. We observe that most users have a relative low number of friends (less than 20). However, some have a high amount. We hypothesis that these are likely music critics, influencers or DJs.

```
user_friends.groupby("userID")["friendID"].nunique().hist(bins=50)
plt.title("Amount of friends that users have")
plt.ylabel("User count")
plt.xlabel("Connection count")
plt.show()
plt.show()
```



Data Cleaning

Through the course of our visualisation development, we noticed a number of irregularities. According to the data, User 43 tagged Freemasons with the tag 'dance' in 1956. User 133 tagged Claudia Leitte with the tag 'axe' in 1957. This was before Lastfm was created, and before either artist started their careers! Thus, these values are removed from further analysis. In the graph analysis notebook, we are only interested in unique relationships i.e. the dataframe should **not** contain two values detailing the same relationship (user_25 -> user_51, user_51 ->user_25). Networkx's Graph object demands this requirement for undirected graphs. In addition, 5,499 artists **did not** have any listening counts attached so these were not included in our investigation.

```
flatten_df = flatten_df[flatten_df.year >=2005]
unique_friendships = list()
for friend1, friend2 in zip(user_friends['userID'].values, user_friends['friendID'].
 ↪values):
    if (friend1,friend2) not in unique_friendships and (friend2,friend1) not in␣
 ↪unique_friendships:
        unique_friendships.append((friend1,friend2))
unique_friendships_df = pd.DataFrame(unique_friendships, columns =['friend1_id',
 ↪'friend2_id'])
```

Save dataframes

```
unique_friendships_df.to_csv('data/unqiue_friendships.csv', index=False)
flatten_df.to_csv('data/flatten_data.csv', index=False)
```

CHAPTER

ONE

COLLABORATIVE FILTERING

A distinction is often made between two forms of data collection for recommendation systems. Explicit feedback relies on the user giving explicit signals about their preferences i.e. review ratings. Where as, implicit feedback refers to non-explicit signals of preference e.g. user watch-time. Traditionally, recommender systems can be split into three types:

- **Collaborative filtering (CF)**: CF produces recommendations based on the knowledge of users' attitudes towards items, that is, it uses the "wisdom of the crowd" to recommend items.
- **Content-based (CB)**: CB recommender systems focus on the attributes of the items to recommend other items similar to what the user likes, based on their previous actions or explicit feedback.
- **Hybrid recommendation systems**: Hybrid methods are a combination of CB recommending and CF methods

In many applications, content-based features are not easy to extract, and thus, collaborative filtering approaches are preferred. Thus, we will only explore collaborative filtering methods from now on.

CF methods typically fall into three types, memory-based, model-based and more recently deep-learning based (Su & Khoshgoftaar, 2009, He et al., 2017). Neighbour-based CF and item-based/user-based top-N recommendations are typical examples of memory-based systems that utilises user rating data to compute the similarity between users or items. As mentioned previously, common model-based approaches include Bayesian networks, latent semantic models and markov decision processes. In this investigation, we will utilise a weighted matrix factorization approach. Later on, we will generalize the matrix factorization algorithm via a non-linear neural architecture (a softmax model).

However, there are a number of limitations to our approaches such as the inability to model the order of interactions. For instance, Markov chain algorithms (Rendle et al., 2010) can not only encode the same information as traditional CF methods but also the order in which user's interacted with the items. Furthermore, the sparsity of the frequency matrix (*described later on*), makes computations prohibitly expensive in real-world settings, without some optimization.

1.1 Quick Links:

1.2 Setup

The next few code cells details the initial preparatory steps needed for the development of our collaborative filtering models, namely importing the required libraries; scaling the ids of users and artists;constructing a indicator variable for presence of user-artist interaction;finding the most assigned tag of an artist.

```
from __future__ import print_function
import numpy as np
import pandas as pd
import collections
from IPython import display
from matplotlib import pyplot as plt
import sklearn
import sklearn.manifold
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
tf.logging.set_verbosity(tf.logging.ERROR)

# Add some convenience functions to Pandas DataFrame.
pd.options.display.max_rows = 10
pd.options.display.float_format = '{:.3f}'.format

# Install Altair and activate its colab renderer.
print("Installing Altair...")
!pip install git+git://github.com/altair-viz/altair.git
import altair as alt
alt.data_transformers.enable('default', max_rows=None)
alt.renderers.enable('colab')
print("Done installing Altair.")
```

```
# NEEDED FOR GOOGLE COLAB
# from google.colab import auth
#from google.colab import drive
# import gspread
# from oauth2client.client import GoogleCredentials

# drive.mount('/content/drive/')
# os.chdir("/content/drive/My Drive/DCU/fouth_year/advanced_machine_learning/music-
↪recommodation-system")
```

Helper functions

```
def calculate_sparsity(M):
    """
    Computes sparsity of frequency matrix
    """
    matrix_size = len((M['userID'].unique())) * len((M['artistID'].unique()))  #␣
↪Number of possible interactions in the matrix
    num_plays = len(M['weight']) # Number of weights
    sparsity = (float(num_plays/matrix_size))
    return sparsity
```

```
def build_music_sparse_tensor(music_df):
  """
  Args:
    ratings_df: a pd.DataFrame with `userID`, `artistID` and `weight` columns.
```

(continues on next page)

(continued from previous page)

```
    num_rows: an integer representing the number of rows in the frequency matrix
    num_rows: an integer representing the number of columns in the frequency matrix
  Returns:
    a tf.SparseTensor representing the feedback matrix.
  """
  indices = music_df[['userID', 'artistID']].values
  values = music_df['weight'].values
  return tf.SparseTensor(
      indices=indices,
      values=values,
      dense_shape=[num_users, num_artist])
```

```
def preproces_ids(music_df):
  """
  Args:
    ratings_df: a pd.DataFrame with `userID`, `artistID` and `weight` columns.
  Returns:
    a pd.DataFrame where userIDs and artistIDs now start at 1
      and end at n and m (defined above), respectively
    two dictionary preserving the orginal ids.
  """
  unique_user_ids_list = sorted(music_df['userID'].unique())
  print(unique_user_ids_list[0])

  unique_user_ids = dict(zip(range(0, len(unique_user_ids_list) ),unique_user_ids_
↪list))
  unique_user_ids_switched = dict(zip(unique_user_ids_list, range(0, len(unique_user_
↪ids) )))

  unique_artist_ids_list = sorted(music_df['artistID'].unique())
  unique_artist_ids = dict(zip(range(0, len(unique_artist_ids_list) ),unique_artist_
↪ids_list))
  unique_artist_ids_switched = dict(zip(unique_artist_ids_list, range(0, len(unique_
↪artist_ids_list) )))

  music_df['userID'] = music_df['userID'].map(unique_user_ids_switched)
  music_df['artistID'] = music_df['artistID'].map(unique_artist_ids_switched)

  return music_df, unique_user_ids, unique_artist_ids
```

```
def split_dataframe(df, holdout_fraction=0.1):
  """Splits a DataFrame into training and test sets.
  Args:
    df: a dataframe.
    holdout_fraction: fraction of dataframe rows to use in the test set.
  Returns:
    train: dataframe for training
    test: dataframe for testing
  """
  test = df.sample(frac=holdout_fraction, replace=False)
  train = df[~df.index.isin(test.index)]
  return train, test
```

Traditional recommender system development relies on explicit feedback. Many models were designed to tackle this issue as a regression problem. For instance, the input of the model would be a matrix F_{nm} denoting user's (m) preference of items (n) on a scale. In the classic movie ratings example, this preference would be users giving a 1-to-5 star rating to

different movies.

This dataset contains implicit feedback: that is, observed logs of user interactions with items, in this instance user's listening counts to artists. However, implicit feedback does not signal negativity, in the same way as a 1-star rating would. In our data, a user could listen to song of an artist a limited number of times. But that does not necessarily mean that the particular user has an aversion to that artist i.e. it could be part of a curated playlist by another user. Therefore, we decide to construct a binary matrix, which has a value of one if the observation is observed (i.e. a listening count has been logged between an artist and a user). **Note**, a 0 is **not used** to describe unobserved artist-user interactions. This is for optimization reasons, explained below.

```
user_artists = pd.read_csv('data/user_artists.dat', sep='\t')
user_artists['weight'] = 1
artists = pd.read_csv('data/artists.dat', sep='\t')
artists.rename({'id':'artistID'}, inplace=True, axis=1)
```

```
user_taggedartists = pd.read_csv(r'data/user_taggedartists-timestamps.dat', sep='\t')
user_taggedartists_years = pd.read_csv(r'data/user_taggedartists.dat', sep='\t')
tags = pd.read_csv(open('data/tags.dat', errors='replace'), sep='\t')
user_taggedartists = pd.merge(user_taggedartists, tags, on=['tagID'])
```

```
num_users = user_artists.userID.nunique()
num_artist = artists.artistID.nunique()
collab_filter_df = user_artists
```

Here, we calculate the top 10 tags by popularity. Then, we assign it to a artist, if the artist has a top 10 tag. If an artist's tags are not in the top 10, we input 'N/A'. Note, the next cell can take several mintues to compute.

```
top_10_tags = user_taggedartists['tagValue'].value_counts().index[0:10]
user_taggedartists['top10TagValue'] = None
for index, row in user_taggedartists.iterrows():
  if row['tagValue'] in top_10_tags:
    user_taggedartists.iloc[index, -1] = row['tagValue']
user_taggedartists.fillna('N/A',inplace=True)
```

```
artists = pd.merge(user_taggedartists, artists, on=['artistID'], how='right')[[
 ↪'artistID','name','top10TagValue','tagValue']].fillna('N/A')
artists.groupby(['artistID','name','top10TagValue']).agg(lambda x:x.value_counts().
 ↪index[0]).reset_index()
artists = artists.drop_duplicates(subset=['artistID'])
assert artists.artistID.nunique() == num_artist
```

```
artists.rename({'tagValue':'mostCommonGenre'},axis=1, inplace=True)
```

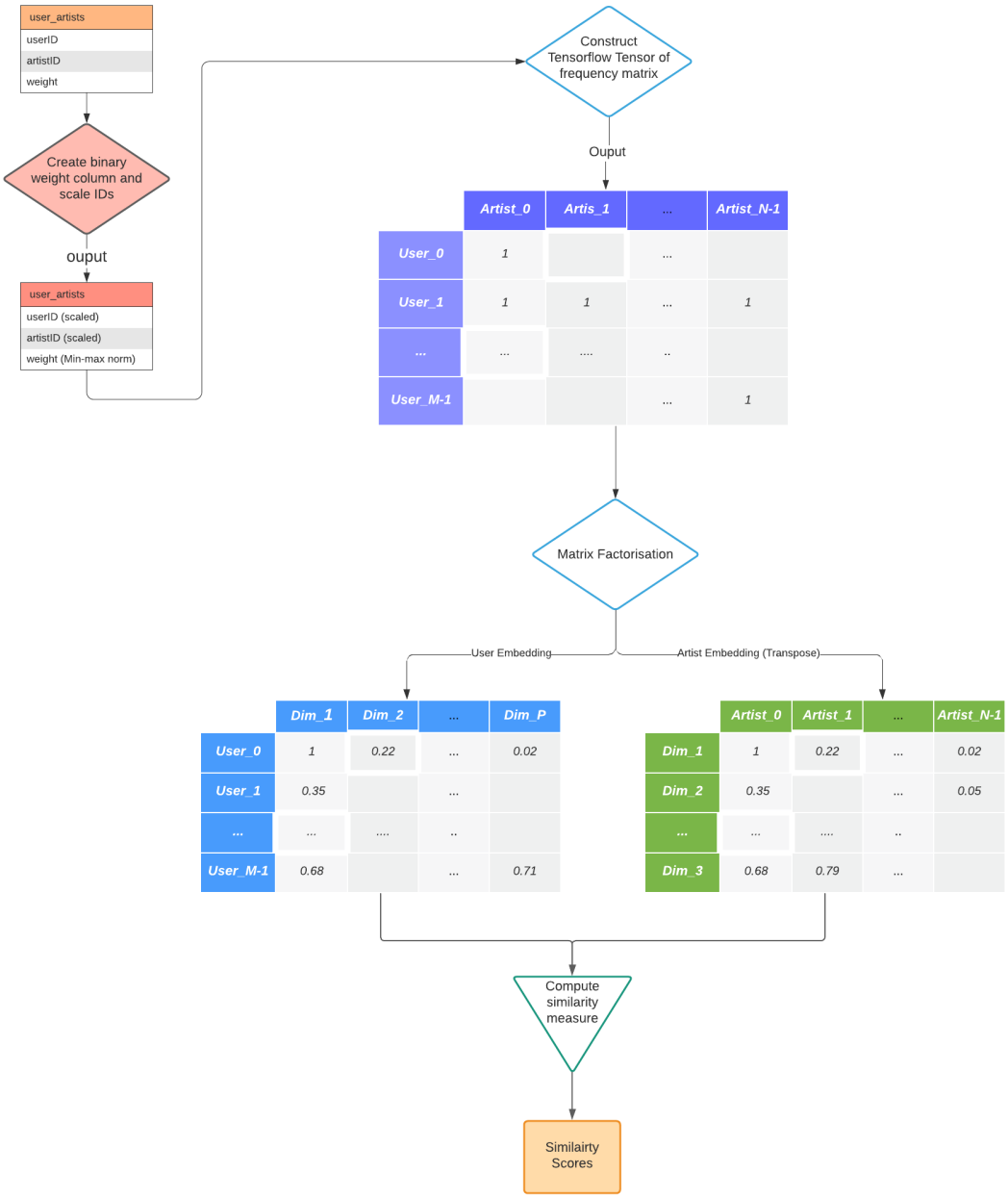
We require two matrices or embeddings to compute a similarity measure (one for quires and one for items), but how do we get these two embeddings?

CHAPTER

TWO

MATRIX FACTORISATION

Fig. 2.1: Data flow chart for the Matrix Factorisation methods



First, we need to contsruct the feedback matrix $F \in R^{m \times n}$, where m is the number of users and n is the number of artists. The goal is to two generate two lower-dimensional matrices U_{mp} and V_{np} (with $p << m$ and $p << n$), representing latent user and artist components, so that: $$F \approx UV^\top$$

First,we attempt to build the frequency matrix for both training and testing data. tf.SparseTensor is used for efficient representation. Three separate arguments are used to represent a tensor, namely `indices, values, dense_shape`, where a value $A_{ij} = a$ is encoded by setting `indices[k] = [i, j]` and `values[k] = a`. The last tensor `dense_shape` is used to specify the shape of the full underlying matrix. Note, as the indices arguments represent row and columns indices, some pre-processing needs to be performed on artist and user IDs. The IDs should start from 0 and end at $m - 1$ and $n - 1$ for users and artists respectively. Presently, userIDs start at 2. Two dictionaries, `orginal_artist_ids`, `orginal_user_ids` will preserve the original ids for analysis purposes later on. Assertions and print statements are used to ensure the validity of the transformations.

```
colab_filter_df, orginal_user_ids, orginal_artist_ids =  preproces_ids(collab_filter_
↪df)
```

```
colab_filter_df.describe()
```

```
         userID  artistID    weight
count 92834.000 92834.000 92834.000
mean    944.222  3235.737     1.000
std     546.751  4197.217     0.000
min       0.000     0.000     1.000
25%     470.000   430.000     1.000
50%     944.000  1237.000     1.000
75%    1416.000  4266.000     1.000
max    1891.000 17631.000     1.000
```

Next, we calculate the number of unique artists, userids and sparisty of our proposed frequency matrix, before splitting into training and test subsets. Quite a sparse matrix indeed!

```
print(f'Number of unqiue users are: {collab_filter_df["userID"].nunique()}')
print(f'Number of unqiue artists are: {collab_filter_df["artistID"].nunique()}')
print(f'Sparsity of our frequency matrix: {calculate_sparsity(collab_filter_df)}')
```

```
Number of unqiue users are: 1892
Number of unqiue artists are: 17632
Sparsity of our frequency matrix: 0.002782815119924182
```

```
collab_filter_df.to_csv('data/test_user_artists.csv',index=False)
```

```
frequency_m_train, frequency_m_test = split_dataframe(colab_filter_df)
frequency_m_train_tensor  = build_music_sparse_tensor(frequency_m_train)
frequency_m_test_tensor  = build_music_sparse_tensor(frequency_m_test)
```

```
assert num_users  == frequency_m_train_tensor.shape.as_list()[0]
assert num_artist == frequency_m_train_tensor.shape.as_list()[1]
assert num_users == frequency_m_test_tensor.shape.as_list()[0]
assert num_artist == frequency_m_test_tensor.shape.as_list()[1]
```

2.1 Training a Matrix factorization model

Per the definition above, $UV^\top$ *approximates* F. The Mean Squared Error is used to measure this approximation error. In the notation below, k is used to represent the **set** of observed listening counts, and K is the **number** of observed listening counts.

$$\text{MSE}(F, UV^\top) = \frac{1}{K} \sum_{(i,j) \in k} (F_{ij} - (UV^\top)_{ij})^2$$

However, rather than computing the full prediction matrix, $UV^\top$ and gathering the entries in the embeddings (corresponding to the observed listening counts) , we only gather the embeddings of the observers pairs and compute their dot products. Thereby, we reduce the complexity from $O(NM)$ to $O(Kp)$ where p is the embedding dimension. Stochastic gradient descent (SGD) is used to minimize the loss (objective) function. The SDG algorithim cycles through the observed listening binary and caulates the prediction according to the following equation.

$$e_{ui} = F_{ui} - U_i V_j$$

Then it updates the user and artist as embeddings as shown in the following equations.

$$U_i \leftarrow U_i + \alpha(e_{ui} V_j - \beta U_i)$$

$$V_j \leftarrow V_j + \alpha(e_{ui} U_j - \beta V_j)$$

where α denotes the learning rate. The algorithim continues untill convergence is found.

Other matrix factorization algorithms functions are also commonly used such as Alternating Least Squares (Takács and Tikk, 2012). A modified version of the aforementioned function known as Weighted Alternating Least Squares (WALS) is slower than SDG but can be parallelised. For the purposes of this investigation, we are not particularly concerned with training times/latency requirements so we proceed with SDG.

We also decide to add regularization to our model, to avoid *overfitting*. Overfitting occurs when the model tries to fit the training dataset to hard and does not generalize well to unseen or future data. In the context of artist recommendation, fitting the observed listening counts often emphasizes learning high similarity (between artists with many listeners), but a good embedding representation also requires learning low similarity (between artists with few listeners).

First, we define the two classes `train_matrix_norm` and `build_matrix_norm` class. The `build_matrix_norm` class computes the necessary pre-processing steps before we train the model such as specifying the loss metric to optimise and the loss components(e.g. gravity loss for the regularized model) and the initial artist and user embeddings. `train_matrix_norm` simply trains the models and outputs figures detailing the the loss metrics and components. The methods `build_vanilla()` and `build_reg_model()` computes the necessary pre-processing steps for the non-regularized and regularized model.

```
### Training a Matrix Factorization model
class train_matrix_norm(object):
  """Simple class that represents a matrix normalisation model"""
  def __init__(self, embedding_vars, loss, metrics=None):
    """Initializes a Matrix normalisation model
    Args:
      embedding_vars: A dictionary of tf.Variables.
      loss: A float Tensor. The loss to optimize.
      metrics: optional list of dictionaries of Tensors. The metrics in each
        dictionary will be plotted in a separate figure during training.
    """
    self._embedding_vars = embedding_vars
    self._loss = loss
    self._metrics = metrics
```

(continues on next page)

(continued from previous page)

```
    self._embeddings = {k: None for k in embedding_vars}
    self._session = None


  @property
  def embeddings(self):
    """The embeddings dictionary."""
    return self._embeddings


  def train(self, num_iterations=100, learning_rate=1.0, plot_results=True,
            optimizer=tf.train.GradientDescentOptimizer):
    """Trains the model.
    Args:
      iterations: number of iterations to run.
      learning_rate: optimizer learning rate.
      plot_results: whether to plot the results at the end of training.
      optimizer: the optimizer to use. Default to SDG
    Returns:
      The metrics dictionary evaluated at the last iteration.
    """
    with self._loss.graph.as_default():
      opt = optimizer(learning_rate)
      train_op = opt.minimize(self._loss)
      local_init_op = tf.group(
          tf.variables_initializer(opt.variables()),
          tf.local_variables_initializer())
      if self._session is None:
        self._session = tf.Session()
        with self._session.as_default():
          self._session.run(tf.global_variables_initializer())
          self._session.run(tf.tables_initializer())
          tf.train.start_queue_runners()

    with self._session.as_default():
      local_init_op.run()
      iterations = []
      metrics = self._metrics or ({},)
      metrics_vals = [collections.defaultdict(list) for _ in self._metrics]

      # Train and append results.
      for i in range(num_iterations + 1):
        _, results = self._session.run((train_op, metrics))
        if (i % 10 == 0) or i == num_iterations:
          print("\r iteration %d: " % i + ", ".join(
                ["%s=%f" % (k, v) for r in results for k, v in r.items()]),
                end='')
          iterations.append(i)
          for metric_val, result in zip(metrics_vals, results):
            for k, v in result.items():
              metric_val[k].append(v)

      for k, v in self._embedding_vars.items():
        self._embeddings[k] = v.eval()

      if plot_results:
        # Plot the metrics.
```

(continues on next page)

(continued from previous page)

```
        num_subplots = len(metrics)+1
        fig = plt.figure()
        fig.set_size_inches(num_subplots*10, 8)
        for i, metric_vals in enumerate(metrics_vals):
          ax = fig.add_subplot(1, num_subplots, i+1)
          for k, v in metric_vals.items():
            ax.plot(iterations, v, label=k)
          ax.set_xlim([1, num_iterations])
          ax.legend()
      return results

class build_matrix_norm():
  """Simple class that represents a matrix normalisation model"""
  def __init__(self, listens, embedding_dim=3, regularization_coeff=.1, gravity_
 ↪coeff=1.,
    init_stddev=0.1):
    """Initializes a Matrix normalisation model
    Args:
      listens: the DataFrame of artist listening counts.
      embedding_dim: The dimension of the embedding space.
      regularization_coeff: The regularization coefficient lambda.
      gravity_coeff: The gravity regularization coefficient lambda_g.
    Returns:
      A train_matrix_norm object that uses a regularized loss.
  """
    self._embedding_vars = embedding_vars
    self._loss = loss
    self._metrics = metrics
    self._embeddings = {k: None for k in embedding_vars}
    self._session = None

  def sparse_mean_square_error(sparse_listens, user_embeddings, artist_embeddings):
    """
    Args:
      sparse_listens: A SparseTensor rating matrix, of dense_shape [N, M]
      user_embeddings: A dense Tensor U of shape [N, k] where k is the embedding
        dimension, such that U_i is the embedding of user i.
      artist_embeddings: A dense Tensor V of shape [M, k] where k is the embedding
        dimension, such that V_j is the embedding of movie j.
    Returns:
      A scalar Tensor representing the MSE between the true ratings and the
        model's predictions.
    """
    predictions = tf.gather_nd(
        tf.matmul(user_embeddings, artist_embeddings, transpose_b=True),
        sparse_listens.indices)
    loss = tf.losses.mean_squared_error(sparse_listens.values, predictions)
    return loss

  def gravity(U, V):
    """Creates a gravity loss given two embedding matrices."""
    return 1. / (U.shape[0].value*V.shape[0].value) * tf.reduce_sum(
        tf.matmul(U, U, transpose_a=True) * tf.matmul(V, V, transpose_a=True))

  def build_vanilla(embedding_dim=3, init_stddev=1.):
    """performs the necessary preprocessing steps for the regularized model.  """
    # Initialize the embeddings using a normal distribution.
```

(continues on next page)

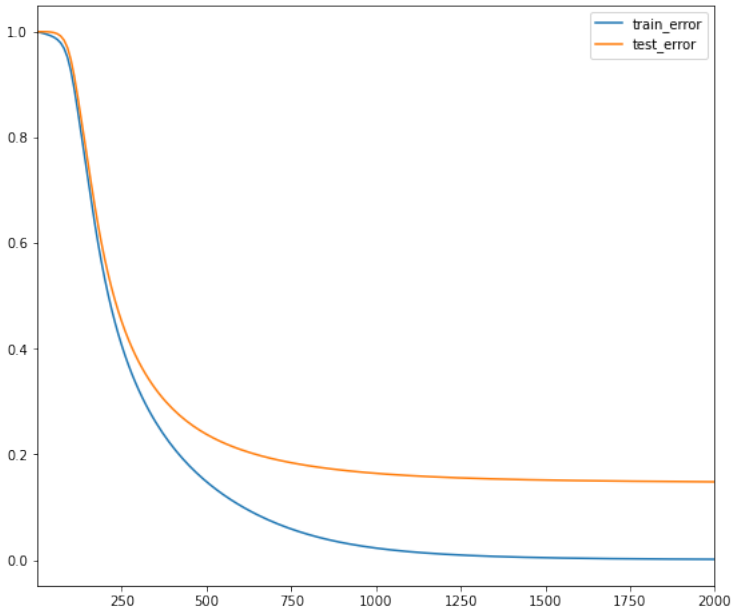
(continued from previous page)

```
    U = tf.Variable(tf.random.normal(
      [frequency_m_train_tensor.dense_shape[0], embedding_dim], stddev=init_stddev))
    V = tf.Variable(tf.random.normal(
      [frequency_m_train_tensor.dense_shape[1], embedding_dim], stddev=init_stddev))

    embeddings = {"userID": U, "artistID": V}
    error_train = build_matrix_norm.sparse_mean_square_error(frequency_m_train_tensor,
↪ U, V)
    error_test = build_matrix_norm.sparse_mean_square_error(frequency_m_test_tensor,␣
↪U, V)
    metrics = {
        'train_error': error_train,
        'test_error': error_test
    }
    return train_matrix_norm(embeddings, error_train, [metrics])


 def build_reg_model(embedding_dim=3, regularization_coeff=.1, gravity_coeff=1.,
 init_stddev=0.1
 ):
    """performs the necessary preprocessing steps for the regularized model.  """
    U = tf.Variable(tf.random.normal(
      [frequency_m_train_tensor.dense_shape[0], embedding_dim], stddev=init_stddev))
    V = tf.Variable(tf.random.normal(
      [frequency_m_train_tensor.dense_shape[1], embedding_dim], stddev=init_stddev))

    embeddings = {"userID": U, "artistID": V}

    error_train = build_matrix_norm.sparse_mean_square_error(frequency_m_train_tensor,
↪ U, V)
    error_test = build_matrix_norm.sparse_mean_square_error(frequency_m_test_tensor,␣
↪U, V)
    gravity_loss = gravity_coeff * build_matrix_norm.gravity(U, V)
    regularization_loss = regularization_coeff * (
      tf.reduce_sum(U*U)/U.shape[0].value + tf.reduce_sum(V*V)/V.shape[0].value)
    total_loss = error_train + regularization_loss + gravity_loss
    losses = {
      'train_error_observed': error_train,
      'test_error_observed': error_test,
    }
    loss_components = {
      'observed_loss': error_train,
      'regularization_loss': regularization_loss,
      'gravity_loss': gravity_loss,
    }
    #embeddings = {"userID": U, "artistID": V}

    return train_matrix_norm(embeddings, total_loss, [losses, loss_components])
```

2.2 Vanilla Model (non-regularized)

```
vanilla_model = build_matrix_norm.build_vanilla(embedding_dim=35,init_stddev=.05)
vanilla_model.train(num_iterations=2000, learning_rate=20.)
```

```
 iteration 2000: train_error=0.001319, test_error=0.148018
```



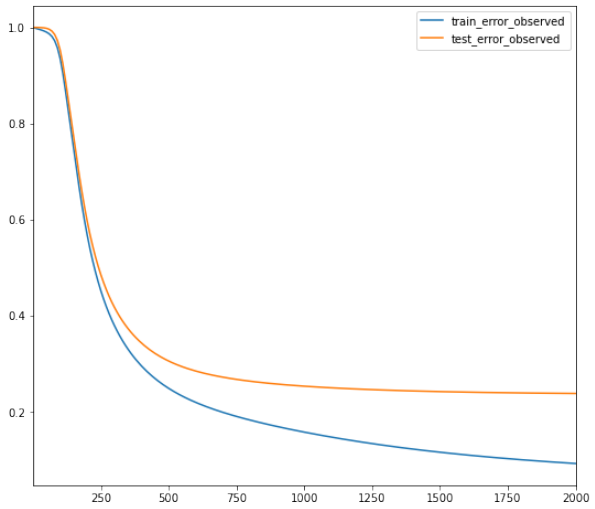
2.3 Regularized moodel

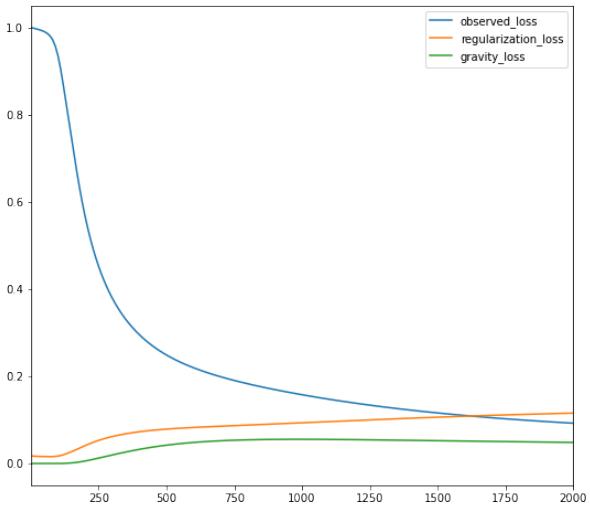
```
reg_model = build_matrix_norm.build_reg_model(regularization_coeff=0.1, gravity_
↪coeff=1.0, embedding_dim=35,init_stddev=.05)
```

```
reg_model.train(num_iterations=2000, learning_rate=20.)
```

```
 iteration 2000: train_error_observed=0.092527, test_error_observed=0.238303,␣
↪observed_loss=0.092527, regularization_loss=0.115726, gravity_loss=0.048498
```

```
[{'train_error_observed': 0.09252745, 'test_error_observed': 0.238303},
 {'observed_loss': 0.09252745,
  'regularization_loss': 0.115726285,
  'gravity_loss': 0.04849844}]
```





In both models, we observe a steep loss in train error and test as the model progress. Although, the regularized model has a higher MSE, both on the training and test set. It must be noted that the quality of recommendation is improved when regularization is added, which is proven when the `artist_neighbors()` function is utilized. In addition, we observe in the end evaluation section, that the the performance of the model is improved when regularization is added. The test error decreases similarity to the test error, although it plateaus around the 1000 epoch mark. As expected, the the additional loss generated by the regularization functions increases over epochs. We add the following regularisation terms to our model.

- Regularization of the model parameters. This is a common ℓ_2 regularization term on the embedding matrices, given by $r(U, V) = \frac{1}{N}\sum_i \|U_i\|^2 + \frac{1}{M}\sum_j \|V_j\|^2$.
- A global prior that pushes the prediction of any pair towards zero, called the *gravity* term. This is given by $g(U, V) = \frac{1}{MN}\sum_{i=1}^{N}\sum_{j=1}^{M}\langle U_i, V_j\rangle^2$

These terms modifies the "global" loss (as in, the sum of the network loss and the regularization loss) in order to drive the optimization algorithm in desired directions i.e. prevent overfitting.

2.4 Evaluating the embeddings

We will use two similairty meausres to inspect the robustness of our system:

- **Dot product**: score of artist j $\langle u, V_j\rangle$.
- **Cosine angle**: score of artist j $\frac{\langle u, V_j\rangle}{\|u\|\|V_j\|}$.

```
DOT = 'dot'
COSINE = 'cosine'
def compute_scores(query_embedding, item_embeddings, measure=DOT):
  """Computes the scores of the candidates given a query.
  Args:
    query_embedding: a vector of shape [k], representing the query embedding.
    item_embeddings: a matrix of shape [N, k], such that row i is the embedding
      of item i.
    measure: a string specifying the similarity measure to be used. Can be
      either DOT or COSINE.
  Returns:
    scores: a vector of shape [N], such that scores[i] is the score of item i.
  """
```

(continues on next page)

(continued from previous page)

```
  u = query_embedding
  V = item_embeddings
  if measure == COSINE:
    V = V / np.linalg.norm(V, axis=1, keepdims=True)
    u = u / np.linalg.norm(u)
  scores = u.dot(V.T)
  return scores
```

```
def user_recommendations(model,user_id, k=15, measure=DOT, exclude_rated=False):
    scores = compute_scores(
        model.embeddings["userID"][user_id], model.embeddings["artistID"], measure)
    score_key = measure + ' score'
    df = pd.DataFrame({
        'score': list(scores),
        'name': artists.sort_values('artistID', ascending=True)['name'],
        'most assigned tag':artists.sort_values('artistID', ascending=True)[
↪'mostCommonGenre']
    })
    return df.sort_values(['score'], ascending=False).head(k)


def artist_neighbors(model, title_substring, measure=DOT, k=6):
  # Search for artist ids that match the given substring.
  inv_artist_id_mapping = {v: k for k, v in orginal_artist_ids.items()}
  ids =  artists[artists['name'].str.contains(title_substring)].artistID.values
  titles = artists[artists.artistID.isin(ids)]['name'].values
  if len(titles) == 0:
    raise ValueError("Found no artists with name %s" % title_substring)
  print("Nearest neighbors of : %s." % titles[0])
  if len(titles) > 1:
    print("[Found more than one matching artist. Other candidates: {}]".format(
        ", ".join(titles[1:])))
  artists_id_orginal = ids[0]
  asrtists_id_mapped = inv_artist_id_mapping[ids[0]]
  scores = compute_scores(
      model.embeddings["artistID"][asrtists_id_mapped], model.embeddings["artistID"],
      measure)
  score_key = measure + ' score'
  df = pd.DataFrame({
      score_key: list(scores),
      'name': artists.sort_values('artistID', ascending=True)['name'],
      'most assigned tag':artists.sort_values('artistID', ascending=True)[
↪'mostCommonGenre']
  })
  return df.sort_values([score_key], ascending=False).head(k)
```

Here, we find the most similar artists to the band the cure. We also include the most assigned tag associated with an artist. The reccomdations are conistent with our domain knowedge of bands similar to the cure.

```
artist_neighbors(vanilla_model, "The Cure", DOT)
```

```
Nearest neighbors of : The Cure.
```

```
       dot score             name most assigned tag
9437       0.545         The Cure          chillout
```

(continues on next page)

(continued from previous page)

```
2239       0.535        Daft Punk          chillout
10365      0.534  The Cranberries       atmospheric
17472      0.530      The Killers          chillout
17278      0.529    Kings of Leon          chillout
3259       0.528          Coldplay          chillout
```

```
artist_neighbors(vanilla_model, "The Cure", COSINE)
```

```
Nearest neighbors of : The Cure.
```

```
       cosine score         name most assigned tag
9437          1.000     The Cure          chillout
16680         0.970  The Beatles          chillout
12363         0.967         Muse          chillout
8273          0.966    Radiohead          chillout
43413         0.959  David Bowie          chillout
17472         0.959  The Killers          chillout
```

```
artist_neighbors(reg_model, "The Cure", DOT)
```

```
Nearest neighbors of : The Cure.
```

```
       dot score         name most assigned tag
12363      3.274         Muse          chillout
16680      3.250  The Beatles          chillout
18364      3.226      Nirvana               pop
9437       3.224     The Cure          chillout
40639      3.184        Oasis               pop
3259       3.160     Coldplay          chillout
```

```
artist_neighbors(reg_model, "The Cure", COSINE)
```

```
Nearest neighbors of : The Cure.
```

```
       cosine score                   name most assigned tag
9437          1.000               The Cure          chillout
4936          0.967           Depeche Mode          chillout
40639         0.966                  Oasis               pop
33564         0.959  The Smashing Pumpkins       atmospheric
18364         0.955                Nirvana               pop
32942         0.954             The Smiths             groove
```

We observe that dot product tends to recommends more popular artists such as Nirvana and The Beatles, where as Cosine Similarity recommends more obscure artists. This is likely due to the fact that the norm of the embedding in matrix factorization is often correlated with prevalence. The regularised model seems to output better reccomodations as the varation of the most assigned tag attribute is less when compared to the vanilla model. In addition, Marilyn Manson was recommended by the vanilla model in our intial run. We argue that these artists are most dis-similar! However, this observation is subject to change when you run the model, as we initialize the embedddings with a random gaussian generator.

```
def artist_embedding_norm(models):
  """Visualizes the norm and number of ratings of the artist embeddings.
```

(continues on next page)

(continued from previous page)

```
  Args:
    model: A train_matrix_norm object.
  """
  if not isinstance(models, list):
    models = [models]
    df = pd.DataFrame({
          'name': artists.sort_values('artistID', ascending=True)['name'].values,
        'number of user-artist interactions': user_artists[['artistID','userID']].
↪sort_values('artistID', ascending=True).groupby('artistID').count()['userID'].
↪values,
    })
  charts = []
  brush = alt.selection_interval()
  for i, model in enumerate(models):
    norm_key = 'norm'+str(i)
    df[norm_key] = np.linalg.norm(model.embeddings["artistID"], axis=1)
    nearest = alt.selection(
        type='single', encodings=['x', 'y'], on='mouseover', nearest=True,
        empty='none')
    base = alt.Chart().mark_circle().encode(
        x='number of user-artist interactions',
        y=norm_key,
        color=alt.condition(brush, alt.value('#4c78a8'), alt.value('lightgray'))
    ).properties(
        selection=nearest).add_selection(brush)
    text = alt.Chart().mark_text(align='center', dx=5, dy=-5).encode(
        x='number of user-artist interactions', y=norm_key,
        text=alt.condition(nearest, 'name', alt.value('')))
    charts.append(alt.layer(base, text))
  return alt.hconcat(*charts, data=df)

artist_embedding_norm(reg_model)
```

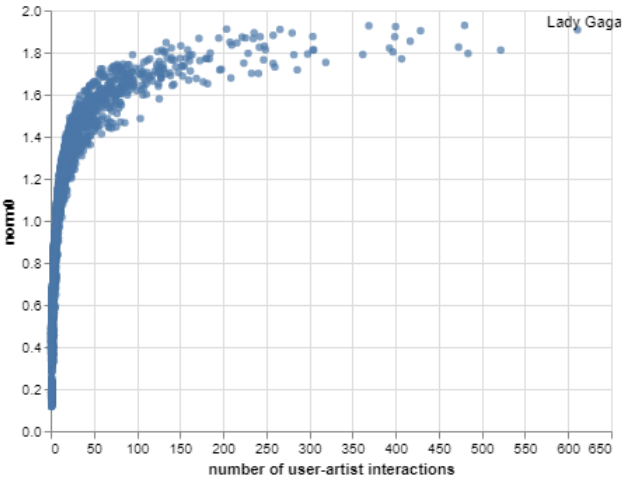


Fig. 2.2: Norm of artist embedding vs popularity

```
def visualize_movie_embeddings(data, x, y):
  genre_filter = alt.selection_multi(fields=['top10TagValue'])
  genre_chart = alt.Chart().mark_bar().encode(
```

(continues on next page)

(continued from previous page)

```
      x="count()",
      y=alt.Y('top10TagValue'),
      color=alt.condition(
          genre_filter,
          alt.Color("top10TagValue:N"),
          alt.value('lightgray'))
  ).properties(height=300, selection=genre_filter)
  nearest = alt.selection(
      type='single', encodings=['x', 'y'], on='mouseover', nearest=True,
      empty='none')
  base = alt.Chart().mark_circle().encode(
      x=x,
      y=y,
      color=alt.condition(genre_filter, "top10TagValue", alt.value("whitesmoke")),
  ).properties(
      width=600,
      height=600,
      selection=nearest)
  text = alt.Chart().mark_text(align='left', dx=5, dy=-5).encode(
      x=x,
      y=y,
      text=alt.condition(nearest, 'name', alt.value('')))
  return alt.hconcat(alt.layer(base, text), genre_chart, data=data)

def tsne_movie_embeddings(model):
  """Visualizes the movie embeddings, projected using t-SNE with Cosine measure.
  Args:
    model: A MFModel object.
  """
  tsne = sklearn.manifold.TSNE(
      n_components=2, perplexity=40, metric='cosine', early_exaggeration=10.0,
      init='pca', verbose=True, n_iter=400)

  print('Running t-SNE...')
  V_proj = tsne.fit_transform(model.embeddings["artistID"])
  artists.loc[:,'x'] = V_proj[:, 0]
  artists.loc[:,'y'] = V_proj[:, 1]
  return visualize_movie_embeddings(artists, 'x', 'y')
```

T-distributed stochastic neighbor embedding (t-SNE) is a dimensionality reduction algorithm useful for visualizing high dimensional data. We use this algorithim to visualise our embeddings of the regualrised model. Due to the large number of user submitted semantic categories, we decide to color-code the top 15 tags, with the rest being labelled as 'N/A'. Although the sea of orange, indicating'N/A', makes it difficult to interrupt these results, the regularised model seems to adequaltly cluster artists of a similar genre in it's embeddings.

```
tsne_movie_embeddings(reg_model)
```

```
def m_embedding_norm(models):
  """Visualizes the norm and number of ratings of the movie embeddings.
  Args:
    model: A MFModel object.
  """
  if not isinstance(models, list):
    models = [models]
    df = pd.DataFrame({
          'title': artists.sort_values('artistID', ascending=True)['name'].values,
```

(continues on next page)

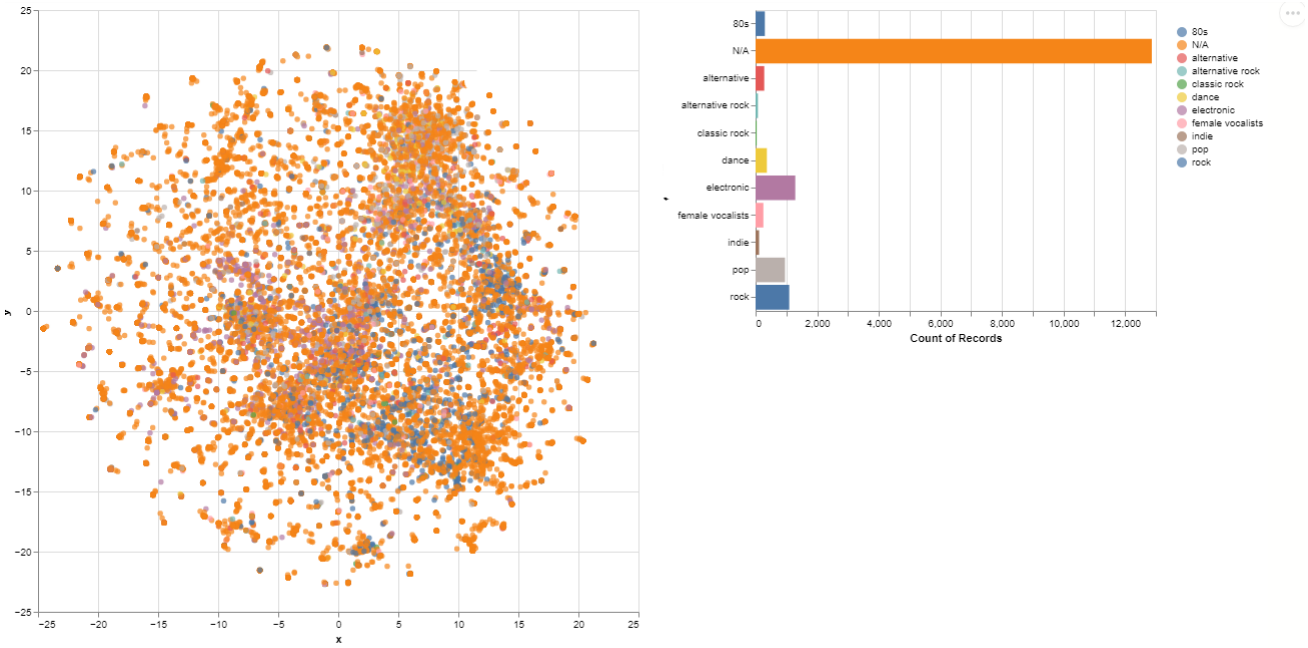


Fig. 2.3: T-SNE visualisation of artist embeddings, color coded by the top 15 tags.

(continued from previous page)

```
        'num_ratings': user_artists[['artistID','userID']].sort_values('artistID',␣
↪ascending=True).groupby('artistID').count()['userID'].values,
    })
  charts = []
  brush = alt.selection_interval()
  for i, model in enumerate(models):
    norm_key = 'norm'+str(i)
    df[norm_key] = np.linalg.norm(model.embeddings["artistID"], axis=1)
    nearest = alt.selection(
        type='single', encodings=['x', 'y'], on='mouseover', nearest=True,
        empty='none')
    base = alt.Chart().mark_circle().encode(
        x='num_ratings',
        y=norm_key,
        color=alt.condition(brush, alt.value('#4c78a8'), alt.value('lightgray'))
    ).properties(
        selection=nearest).add_selection(brush)
    text = alt.Chart().mark_text(align='center', dx=5, dy=-5).encode(
        x='num_ratings', y=norm_key,
        text=alt.condition(nearest, 'title', alt.value('')))
    charts.append(alt.layer(base, text))
  return alt.hconcat(*charts, data=df)
```

2.5 Demo

You can find the most similar artist to a specified artist (that is contained in Last.FM) using the `artist_neighbours()` function. Similarily, you can find the top 10 recommendations of a particular userID [0 to 1891] using the `user_recommendations()` function. The first argument specifies the desired model, second argument the userID and third the top-k recommendations. Fourth argument represents the similarity measure, either DOT or COSINE (default = DOT, not a string).

```
user_recommendations(reg_model, 234, 10, COSINE)
```

	score	name	most assigned tag
126490	0.913	Street Bulldogs	N/A
126582	0.899	Validuaté	N/A
126539	0.895	Menstruação Anarquika	N/A
126400	0.889	The Vibrators	punk
126491	0.887	Bandas Gaúchas - www.DownsMtv.com	N/A
121134	0.883	7Seconds	80s
126536	0.878	S.O.D.	thrash metal
126404	0.840	Cachorro Grande	rock
126540	0.831	The Exploited	uk
126492	0.826	Pata de Elefante	rock

To further demonstrate the robustness of the system and measure the serendipity of our model, we incorporate the top artists that we listen to on Spotify (i.e. an unknown user). **Note**, these artists have to also be in the Last.FM dataset. The recommendation system should output similar artists based on it's artist embeddings. The *Spotipy* library is used to interact with Spotify's API. The similarity measure used is the Dot product. Due to the short lived nature of the spotify token and the fact you have to sign into a pop-up to retrieve the authentication token, we simply list our top 5 artists manually. If we did not, jupyter book will stall when attempting to build as it is waiting for our response. However, we provide the code used to retrieve the short-lived token for verification purposes.

```
"""
import spotipy
from spotipy.oauth2 import SpotifyOAuth
client_id = <insert_your_client_id>
client_secret = <insert your client secret>
redirect_url = '<insert your redirect uri>
scope = "user-top-read user-read-playback-state streaming ugc-image-upload playlist-
 ↪modify-public"

authenticate_manager = spotipy.oauth2.SpotifyOAuth(client_id = client_id,client_
 ↪secret = client_secret,redirect_uri =redirect_url,scope =scope,show_dialog = True)
sp = spotipy.Spotify(auth_manager=authenticate_manager)

artists_long = sp.current_user_top_artists(limit=5, time_range="long_term")
"""
top_5_artists =[
                'Coldplay',
                'Paramore',
                'Arctic Monkeys',
                'Lily Allen',
                'Miley Cyrus'
]
spotify_reccomdations_df = pd.DataFrame()
for artist in top_5_artists:
  similar_artist_df = artist_neighbors(reg_model, artist)[['name','dot score']]
  spotify_reccomdations_df = pd.concat([spotify_reccomdations_df, similar_artist_df])
```

(continues on next page)

(continued from previous page)

```
spotify_reccomdations_df.sort_values('dot score', ascending=False).head(10)
```

```
                name  dot score
3259         Coldplay      3.663
12363            Muse      3.573
17832       Green Day      3.534
37842        Paramore      3.514
17278  Kings of Leon       3.512
30355     Linkin Park      3.508
17472     The Killers      3.505
24447      Lily Allen      3.458
30355     Linkin Park      3.456
6543        Lady Gaga      3.443
```

We believe these recommodations are good as when our model was given an artist in the top five, it actually recommended **other** artits in the top five.

2.6 Evaluation Code

This is the code needed to produce the in-depth model comparison. This is explained in-depth later on.

```
## create holdout test set for each user (15 items)
user_artists = pd.read_csv('data/user_artists.dat', sep='\t')
user_ids = []
holdout_artits = []
for user_id in user_artists.userID.unique():
  top_15_artists = user_artists[user_artists.userID == user_id].sort_values(by='weight
↪').head(15).artistID.tolist()
  if len(top_15_artists) == 15:
    holdout_artits.append(top_15_artists)
    user_ids.append(user_id)
holdout_df = pd.DataFrame(data={'userID':user_ids,'holdout_artists':holdout_artits})

holdout_df.to_csv('data/evaluation/test-set.csv',index=False)
```

```
## Finding the models vanilla, regualrised predection for each user.
def get_top_15_model_predictions(model, measure):
  """Computes the top 15 predictions for a given model
  Args:
    model: the name of the model
    measure: a string specifying the similarity measure to be used. Can be
      either DOT or COSINE.
  Returns:
    predicted_df a dataframe containing userIDs, their top 15 artists by the model,␣
↪and the correspnding scores.
  """
  artist_name_id_dict = dict(zip(artists['name'], artists['artistID']))
  user_ids = []
  predicted_artists = []
  scores_list = []
  for new_user_id, orginal_user_id in orginal_user_ids.items():
    top_15_names = user_recommendations(model, new_user_id, k=15,measure=measure )[
↪'name'].values
```

(continues on next page)

(continued from previous page)

```
    top_15_scores = user_recommendations(model, new_user_id, k=15, measure=measure )[
↪'score'].values.tolist()
    artist_ids = []
    for name in top_15_names:
      artist_ids.append(artist_name_id_dict[name])
    predicted_artists.append(artist_ids)
    user_ids.append(orginal_user_id)
    scores_list.append(top_15_scores)
  predicted_df = pd.DataFrame(data={'userID':user_ids,'predictions_artists':predicted_
↪artists, 'score':scores_list })
  return predicted_df
```

```
# save the recommended artits into dfs and save them to data/evaluation folder
vanilla_dot_pred= get_top_15_model_predictions(vanilla_model, measure=DOT)
vanilla_cos_pred = get_top_15_model_predictions(vanilla_model, measure=COSINE)
reg_dot_pred= get_top_15_model_predictions(reg_model, measure=DOT)
reg_cos_pred = get_top_15_model_predictions(reg_model, measure=COSINE)

vanilla_dot_pred.to_csv('data/evaluation/vannila_dot_pred.csv',index=False)
vanilla_cos_pred.to_csv('data/evaluation/vanila_cos_pred.csv',index=False)
reg_dot_pred.to_csv('data/evaluation/reg_dot_pred.csv',index=False)
reg_cos_pred.to_csv('data/evaluation/reg_cos_pred.csv',index=False)
```

CHAPTER

THREE

SOFTMAX MODEL

From the devlopment and the research required to implement the matrix factorisation algorithims, we notice some limitations:

1. Only user and artist interactions are used in our matrix factoriation model. The inner product of the subsequent user and artist embeddings may be not enough to capture and represent the complex relations in the user and artists.
2. The matrix factorisation methods suffer heavily from the cold start problem due to the absense of an embedding for new artists.
3. When dot product is used as the similarity measure, there is tendency for it to recommend artist with the most amount of interactions, which may fail for users with niche listening habits.
4. We can not query a user that is not present in the training set.
5. Adding side features is difficult to do in matrix factorisation methods.

Several of these limitations motivate us to develop a non-linear generalization of factorization techniques. We choose a softmax deep learning model as we can easily incorporate query features and item features. This model treats the problem as a multiclass prediction problem in which:

- Input is a query vector
- Ouput is the probability vector with size equal to the number of artists present in the Last.fm dataset. This probability represents the likehood of user to have listened to each artist.

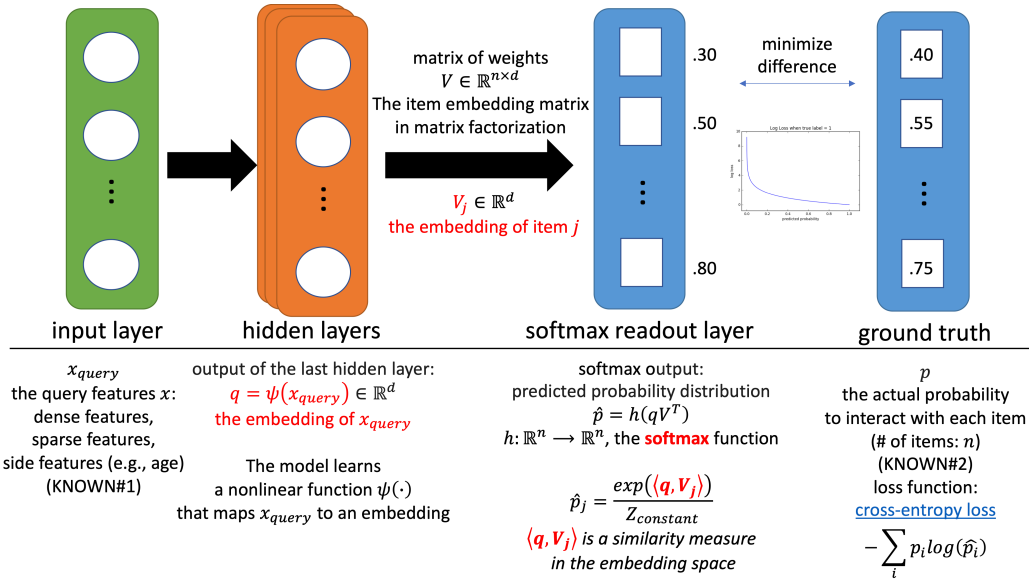


Fig. 3.1: Workflow of softmax model

3.1 Quick Links

3.2 Setup

The next code cells detail the initial preparatory steps needed for the development of our softmax model, namely importing the required libraries; scaling the ids of users and artists;finding the most assigned tag of an artist;

```
from __future__ import print_function
import os
import numpy as np
import pandas as pd
import collections
from mpl_toolkits.mplot3d import Axes3D
from IPython import display
from matplotlib import pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import sklearn
import sklearn.manifold
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
tf.logging.set_verbosity(tf.logging.ERROR)

# Install Altair and activate its colab renderer.
print("Installing Altair...")
!pip install git+git://github.com/altair-viz/altair.git
import altair as alt
alt.data_transformers.enable('default', max_rows=None)
alt.renderers.enable('colab')
print("Done installing Altair.")
```

```
# from google.colab import drive
# !pip install --upgrade -q gspread

# from google.colab import auth
# import gspread
# from oauth2client.client import GoogleCredentials
# drive.mount('/content/drive/')
# os.chdir("/content/drive/My Drive/DCU/fouth_year/advanced_machine_learning/music-
↪recommodation-system")
```

```
Mounted at /content/drive/
```

```
def split_dataframe(df, holdout_fraction=0.1):
  """Splits a DataFrame into training and test sets.
  Args:
    df: a dataframe.
    holdout_fraction: fraction of dataframe rows to use in the test set.
  Returns:
    train: dataframe for training
    test: dataframe for testing
  """
  test = df.sample(frac=holdout_fraction, replace=False)
  train = df[~df.index.isin(test.index)]
  return train, test


DOT = 'dot'
COSINE = 'cosine'
def compute_scores(query_embedding, item_embeddings, measure=DOT):
  """Computes the scores of the candidates given a query.
  Args:
    query_embedding: a vector of shape [k], representing the query embedding.
    item_embeddings: a matrix of shape [N, k], such that row i is the embedding
      of item i.
    measure: a string specifying the similarity measure to be used. Can be
      either DOT or COSINE.
  Returns:
    scores: a vector of shape [N], such that scores[i] is the score of item i.
  """
  u = query_embedding
  V = item_embeddings
  if measure == COSINE:
    V = V / np.linalg.norm(V, axis=1, keepdims=True)
    u = u / np.linalg.norm(u)
  scores = u.dot(V.T)
  return scores
```

```
artists = pd.read_csv('data/artists.dat', sep='\t')
artist_index = dict(zip(sorted(artists['id'].unique()), [str(num) for num in  range(0,
 ↪artists['id'].nunique())]))
artists = artists.replace({"id": artist_index})
artists = artists.astype(str)
artists.rename({'id':'artistID'}, inplace=True, axis=1)

user_artists = pd.read_csv('data/user_artists.dat', sep='\t')
user_index = dict(zip(sorted(user_artists['userID'].unique()), [str(num) for num in␣
 ↪range(0,user_artists['userID'].nunique())]))
user_artists['weight'] = 1
user_artists = user_artists.replace({"artistID": artist_index,"userID":user_index})
user_artists = user_artists.astype(str)
```

```
user_taggedartists = pd.read_csv(r'data/user_taggedartists-timestamps.dat', sep='\t')
user_taggedartists = user_taggedartists.replace({"artistID": artist_index,"userID
 ↪":user_index})
user_taggedartists = user_taggedartists.astype(str)
user_taggedartists_years = pd.read_csv(r'data/user_taggedartists.dat', sep='\t')
user_taggedartists_years = user_taggedartists_years.astype(str)

tags = pd.read_csv(open('data/tags.dat', errors='replace'), sep='\t')
```

(continues on next page)

(continued from previous page)

```
tags = tags.astype(str)
user_taggedartists = pd.merge(user_taggedartists, tags, on=['tagID'])
```

```
top_15_tags = user_taggedartists['tagValue'].value_counts().index[0:15]
user_taggedartists['top15TagValue'] = None
for index, row in user_taggedartists.iterrows():
  if row['tagValue'] in top_15_tags:
    user_taggedartists.iloc[index, -1] = row['tagValue']

user_taggedartists.fillna('N/A',inplace=True)
```

```
artists = pd.merge(user_taggedartists, artists, on=['artistID'], how='right')[[
 ↪'artistID','name','top15TagValue','tagValue']].fillna('N/A')
artists.groupby(['artistID','name','top15TagValue']).agg(lambda x:x.value_counts().
 ↪index[0]).reset_index()
artists = artists.drop_duplicates(subset=['artistID'])
artists.rename({'tagValue':'mostCommonGenre'},axis=1, inplace=True)
```

3.3 Softmax Model

Our softmax model predicts whether a user has listened to a particular artist or not. The model will take as input a feature vector x representing the list of artists the user has rated i.e. `rated_artists`. The `make_batch()` function generates a batch where each batch contains the following variables

- artist: A tensor of strings of the artist ids that the user rated.
- tag: A tensor of strings of the tags of those artists
- year: A tensor of strings of the years where a user has annoted a artist.

Note, if an artist does not have any tags attached, we input a value of unkown. Likewise, if a user-arist interaction does not have a year attached, we input a value of 2011, the year in which the dataset was created.

The next code cells detail the pre-processings required to create two dictionaries (`year_dict` and `tags_dict`) which hold the data required to implement the tag and year batch features.

```
user_taggedartists_years_unique = user_taggedartists_years.drop_duplicates(subset=[
 ↪'artistID','year'])[['artistID','year']]
user_taggedartists_years_unique = user_taggedartists_years_unique.groupby('artistID')[
 ↪'year'].apply(list).reset_index()
years_dict = dict(zip(user_taggedartists_years_unique['artistID'], user_taggedartists_
 ↪years_unique['year']))
unknown_years = set(artists['artistID'].values).difference(set(years_dict.keys()))
for  unknown_year in unknown_years:
  years_dict[unknown_year] = []
```

```
user_taggedartists_tags = user_taggedartists.drop_duplicates(subset=['artistID',
 ↪'tagValue'])[['artistID','tagValue']]
user_taggedartists_tags = user_taggedartists_tags.groupby('artistID')['tagValue'].
 ↪apply(list).reset_index()
tags_dict = dict(zip(user_taggedartists_tags['artistID'], user_taggedartists_tags[
 ↪'tagValue']))
unkown_tags = set(artists['artistID'].values).difference(set(tags_dict.keys()))
for  unknown_tag in unkown_tags:
  tags_dict[unknown_tag] = []
```

```
rated_artists = (user_artists[["userID", "artistID"]]
                 .groupby("userID", as_index=False)
                 .aggregate(lambda x: list(x)))
```

```
def make_batch(ratings, batch_size):
  """Creates a batch of examples.
  Args:
    ratings: A DataFrame of ratings such that examples["artistID"] is a list of
      artists rated by a user.
    batch_size: The batch size.
  """
  def pad(x, fill):
    return pd.DataFrame.from_dict(x).fillna(fill).values

  artist = []
  year = []
  tag = []
  label = []
  for movie_ids in ratings["artistID"].values:
    artist.append(movie_ids)
    tag.append([str(x) for movie_id in movie_ids for x in tags_dict[movie_id]])
    year.append([str(x) for movie_id in movie_ids for x in years_dict[movie_id]])
    label.append([int(movie_id) for movie_id in movie_ids])

  features = {
      "artistID": pad(artist, ""),
      "year": pad(year, ""),
      "tag": pad(tag, ""),
      "label": pad(label, -1)
  }
  batch = (
      tf.data.Dataset.from_tensor_slices(features)
      .shuffle(1000)
      .repeat()
      .batch(batch_size)
      .make_one_shot_iterator()
      .get_next())
  return batch

def select_random(x):
  """Selectes a random elements from each row of x."""
  def to_float(x):
    return tf.cast(x, tf.float32)
  def to_int(x):
    return tf.cast(x, tf.int64)
  batch_size = tf.shape(x)[0]
  rn = tf.range(batch_size)
  nnz = to_float(tf.count_nonzero(x >= 0, axis=1))
  rnd = tf.random_uniform([batch_size])
  ids = tf.stack([to_int(rn), to_int(nnz * rnd)], axis=1)
  return to_int(tf.gather_nd(x, ids))
```

3.3.1 Define the loss function

As is illustrated in the Figure 3, the softmax model maps the input features x to a user embedding $\psi(x) \in \mathbb{R}^d$, where d is the embedding dimension. We then mutiply this vector by a artist embedding matrix $V \in \mathbb{R}^{n \times d}$ (where n is the number of artists). The final output is the softmax of this product.

$$\hat{p}(x) = \text{softmax}(\psi(x)V^\top).$$

Similar to our matrix factorisation approach, we one-hot encode the user-artist interactions (p). These represent the ground truth i.e. $p = 1$. The loss between the model produced propaility and the target label is the cross-entropy between $\hat{p}(x)$ and p. The loss function the loss function compares two probability vectors (the ground truth and the output of the model).

```
def softmax_loss(user_embeddings, artist_embeddings, labels):
  """Returns the cross-entropy loss of the softmax model.
  Args:
    user_embeddings: A tensor of shape [batch_size, embedding_dim].
    artist_embeddings: A tensor of shape [num_artists, embedding_dim].
    labels: A tensor of [batch_size], such that labels[i] is the target label
      for example i.
  Returns:
    The mean cross-entropy loss.
  """
  # Verify that the embddings have compatible dimensions
  user_emb_dim = user_embeddings.shape[1].value
  movie_emb_dim = artist_embeddings.shape[1].value
  if user_emb_dim != movie_emb_dim:
    raise ValueError(
        "The user embedding dimension %d should match the movie embedding "
        "dimension % d" % (user_emb_dim, movie_emb_dim))

  logits = tf.matmul(user_embeddings, artist_embeddings, transpose_b=True)
  loss = tf.reduce_mean(tf.nn.sparse_softmax_cross_entropy_with_logits(
      logits=logits, labels=labels))
  return loss
```

3.3.2 Build a softmax model

We now build a softmax model using the `build_softmax_model()` function and `train_softmax_model` class. The `build_softmax_model` function concatenates the input emeddings (artistID, tag and year) to form the input layer. We choose 35 hidden layers in our softmax model. By adding hidden layers and non-linear activation functions, we hope that the model can capture more complex relationships in the last.fm data. The last hidden layer in the model is then multiplied by the artist embeddings to obtain the logits layer. For the target label, we will use a randomly-sampled artistID from the list of artist the user rated.

```
class train_softmax_model(object):
  """Simple class that represents a trains the softmax model"""
  def __init__(self, embedding_vars, loss, metrics=None):
    """Initializes a Matrix normalisation model
    Args:
      embedding_vars: A dictionary of tf.Variables.
      loss: A float Tensor. The loss to optimize.
      metrics: optional list of dictionaries of Tensors. The metrics in each
        dictionary will be plotted in a separate figure during training.
    """
```

(continues on next page)

(continued from previous page)

```
    self._embedding_vars = embedding_vars
    self._loss = loss
    self._metrics = metrics
    self._embeddings = {k: None for k in embedding_vars}
    self._session = None


  @property
  def embeddings(self):
    """The embeddings dictionary."""
    return self._embeddings


  def train(self, num_iterations=100, learning_rate=1.0, plot_results=True,
            optimizer=tf.train.GradientDescentOptimizer):
    """Trains the model.
    Args:
      iterations: number of iterations to run.
      learning_rate: optimizer learning rate.
      plot_results: whether to plot the results at the end of training.
      optimizer: the optimizer to use. Default to SDG
    Returns:
      The metrics dictionary evaluated at the last iteration.
    """
    with self._loss.graph.as_default():
      opt = optimizer(learning_rate)
      train_op = opt.minimize(self._loss)
      local_init_op = tf.group(
          tf.variables_initializer(opt.variables()),
          tf.local_variables_initializer())
      if self._session is None:
        self._session = tf.Session()
        with self._session.as_default():
          self._session.run(tf.global_variables_initializer())
          self._session.run(tf.tables_initializer())
          tf.train.start_queue_runners()

    with self._session.as_default():
      local_init_op.run()
      iterations = []
      metrics = self._metrics or ({},)
      metrics_vals = [collections.defaultdict(list) for _ in self._metrics]

      # Train and append results.
      for i in range(num_iterations + 1):
        _, results = self._session.run((train_op, metrics))
        if (i % 10 == 0) or i == num_iterations:
          print("\r iteration %d: " % i + ", ".join(
                ["%s=%f" % (k, v) for r in results for k, v in r.items()]),
                end='')
          iterations.append(i)
          for metric_val, result in zip(metrics_vals, results):
            for k, v in result.items():
              metric_val[k].append(v)

      for k, v in self._embedding_vars.items():
        self._embeddings[k] = v.eval()
```

(continues on next page)

(continued from previous page)

```
    if plot_results:
      # Plot the metrics.
      num_subplots = len(metrics)+1
      fig = plt.figure()
      fig.set_size_inches(num_subplots*10, 8)
      for i, metric_vals in enumerate(metrics_vals):
        ax = fig.add_subplot(1, num_subplots, i+1)
        for k, v in metric_vals.items():
          ax.plot(iterations, v, label=k)
        ax.set_xlim([1, num_iterations])
        ax.legend()
    return results
```

```
def build_softmax_model(rated_artists, embedding_cols, hidden_dims):
  """Builds a Softmax model for MovieLens.
  Args:
    rated_artists: DataFrame of traing examples.
    embedding_cols: A dictionary mapping feature names (string) to embedding
      column objects. This will be used in tf.feature_column.input_layer() to
      create the input layer.
    hidden_dims: int list of the dimensions of the hidden layers.
  Returns:
    A train_softmax_model object.
  """
  def create_network(features):
    """Maps input features dictionary to user embeddings.
    Args:
      features: A dictionary of input string tensors.
    Returns:
      outputs: A tensor of shape [batch_size, embedding_dim].
    """
    # Create a bag-of-words embedding for each sparse feature.
    inputs = tf.feature_column.input_layer(features, embedding_cols)
    # Hidden layers.
    input_dim = inputs.shape[1].value
    for i, output_dim in enumerate(hidden_dims):
      w = tf.get_variable(
          "hidden%d_w_" % i, shape=[input_dim, output_dim],
          initializer=tf.truncated_normal_initializer(
              stddev=1./np.sqrt(output_dim))) / 10.
      outputs = tf.matmul(inputs, w)
      input_dim = output_dim
      inputs = outputs
    return outputs

  train_rated_artists, test_rated_artists = split_dataframe(rated_artists)
  train_batch = make_batch(train_rated_artists, 200)
  test_batch = make_batch(test_rated_artists, 100)

  with tf.variable_scope("model", reuse=False):
    # Train
    train_user_embeddings = create_network(train_batch)
    train_labels = select_random(train_batch["label"])
  with tf.variable_scope("model", reuse=True):
    # Test
```

(continues on next page)

(continued from previous page)

```
    test_user_embeddings = create_network(test_batch)
    test_labels = select_random(test_batch["label"])
    artist_embeddings = tf.get_variable(
        "input_layer/artistID_embedding/embedding_weights")

  test_loss = softmax_loss(
      test_user_embeddings, artist_embeddings, test_labels)
  train_loss = softmax_loss(
      train_user_embeddings, artist_embeddings, train_labels)
  _, test_precision_at_10 = tf.metrics.precision_at_k(
      labels=test_labels,
      predictions=tf.matmul(test_user_embeddings, artist_embeddings, transpose_
↪b=True),
      k=10)

  metrics = (
      {"train_loss": train_loss, "test_loss": test_loss},
      {"test_precision_at_10": test_precision_at_10}
  )
  embeddings = {"artistID": artist_embeddings}
  return train_softmax_model(embeddings, train_loss, metrics)
```

3.3.3 Train the Softmax model

We now train the softmax model. There are four hyper paramterics that we can tune, namely

- learning rate
- number of iterations
- input embedding dimensions (the `input_dims` argument)
- number of hidden layers and size of each layer (the `hidden_dims` argument)

```
# Create feature embedding columns
def make_embedding_col(key, embedding_dim):
  if key == 'artistID':
    unique_keys = artists['artistID'].values
  elif key == 'tag':
    unique_keys = user_taggedartists['tagValue'].values
  elif key =='year':
    unique_keys = user_taggedartists_years['year'].values

  categorical_col = tf.feature_column.categorical_column_with_vocabulary_list(
      key=key, vocabulary_list=list(set(unique_keys)), num_oov_buckets=0)

  return tf.feature_column.embedding_column(
      categorical_column=categorical_col, dimension=embedding_dim,
      combiner='mean')

with tf.Graph().as_default():
  softmax_model = build_softmax_model(
      rated_artists,
      embedding_cols=[
          make_embedding_col("artistID", 35),
          make_embedding_col("tag", 3),
```

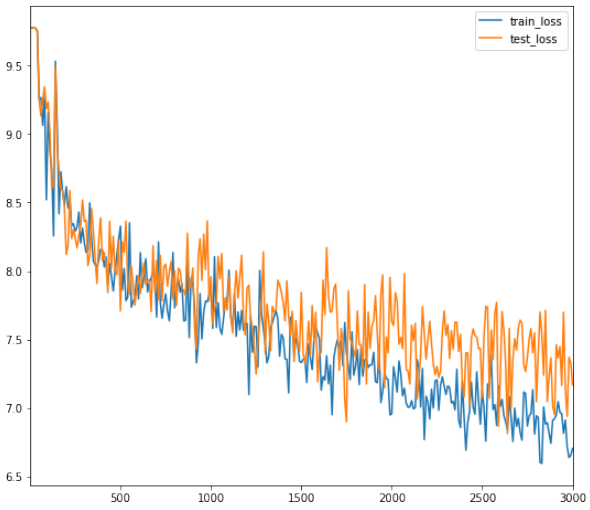
(continues on next page)

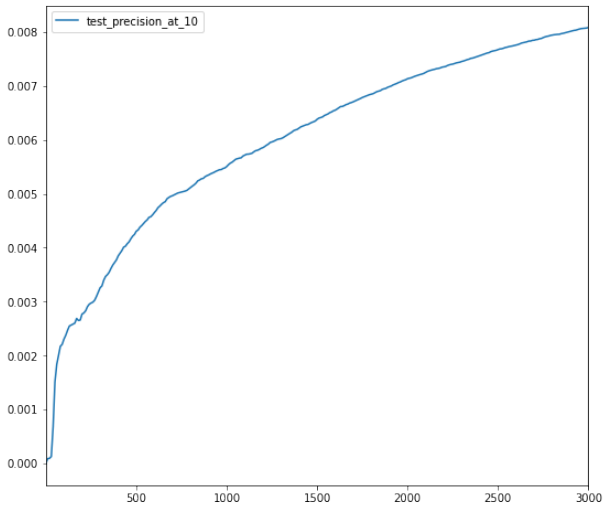
(continued from previous page)

```
            make_embedding_col("year", 2),
        ],
        hidden_dims=[35])

softmax_model.train(
    learning_rate=8., num_iterations=3000, optimizer=tf.train.AdagradOptimizer)
```

```
 iteration 3000: train_loss=6.705453, test_loss=7.169943, test_precision_at_10=0.
↪008087
```

```
({'test_loss': 7.169943, 'train_loss': 6.705453},
 {'test_precision_at_10': 0.008086637787404198})
```





Comapred to the matrix factorisation methods, we observe a more ragged decrease in the softmax loss function as the model progresses compared to the MSE of the matrix factorisation methods.

3.3.4 Evaluate the embeddings

Like the previous section, we evaluate the artist embeddings. We use the same similairty meausres as before, to inspect the robustness of the softmax model.

```
def artist_neighbors(model, title_substring, measure=DOT, k=6):
  # Search for artist ids that match the given substring.
  ids =  artists[artists['name'].str.contains(title_substring)].artistID.values
  titles = artists[artists.artistID.isin(ids)]['name'].values
  if len(titles) == 0:
    raise ValueError("Found no artists with name %s" % title_substring)
  print("Nearest neighbors of : %s." % titles[0])
  if len(titles) > 1:
    print("[Found more than one matching artist. Other candidates: {}]".format(
        ", ".join(titles[1:])))
  scores = compute_scores(
      model.embeddings["artistID"][int(ids)], model.embeddings["artistID"],
      measure)
  score_key = measure + ' score'
  df = pd.DataFrame({
```

(continues on next page)

(continued from previous page)

```
      score_key: list(scores),
      'name': artists.sort_values('artistID', ascending=True)['name'],
      'most assigned tag':artists.sort_values('artistID', ascending=True)[
↪'mostCommonGenre']
  })
  return df.sort_values([score_key], ascending=False).head(k)
```

```
artist_neighbors(softmax_model, "The Cure", DOT)
```

```
Nearest neighbors of : The Cure.
```

```
        dot score                name most assigned tag
161729  32.578003             Firebug              ska
62469   31.128336             Erasure       electronic
161803  30.389132  Peter Pan Speedrock        rock hard
161906  29.331894         Torment Day       electronic
161368  29.005989   Perrey & Kingsley              N/A
161913  28.973671               Пурген             punk
```

```
artist_neighbors(softmax_model, "The Cure", COSINE)
```

```
Nearest neighbors of : The Cure.
```

```
        cosine score              name          most assigned tag
161729      1.000000           Firebug                        ska
162859      0.920241    Taylor Lautner                        N/A
165317      0.895609              Rojo                        N/A
161368      0.880020  Perrey & Kingsley                        N/A
395         0.872602          Behemoth                death metal
164005      0.861260     Chris Crocker  very very guilty pleasure
```

We belive that matrix factorisation models' embeddings actually produced better results than the softmax model. These artists have little in common to The Cure. We hypothesis that adding user-submitted tags may cause the model to learn random noise since users are not domain experts. The may be noise or "contradictions" within the tag data. DNNs aslo require more data in general, when comapred to traditional machine learning methods such as weighted matrix factorisation. The limited user-artist interaction supply (1892 instances in the `rated_artists` df) may not be enough.

```
rated_artists.head(5)
```

```
  userID                                           artistID
0      0  [45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 5...
1      1  [95, 96, 97, 98, 99, 100, 101, 102, 103, 104, ...
2     10  [66, 183, 185, 224, 282, 294, 327, 338, 371, 3...
3    100  [597, 610, 735, 739, 744, 746, 747, 763, 769, ...
4   1000  [49, 50, 58, 59, 61, 65, 83, 251, 282, 283, 28...
```

```
def visualize_movie_embeddings(data, x, y):
  genre_filter = alt.selection_multi(fields=['top15TagValue'])
  genre_chart = alt.Chart().mark_bar().encode(
      x="count()",
      y=alt.Y('top15TagValue'),
      color=alt.condition(
          genre_filter,
```

(continues on next page)

(continued from previous page)

```
          alt.Color("top15tagValue:N"),
          alt.value('lightgray'))
  ).properties(height=300, selection=genre_filter)
  nearest = alt.selection(
      type='single', encodings=['x', 'y'], on='mouseover', nearest=True,
      empty='none')
  base = alt.Chart().mark_circle().encode(
      x=x,
      y=y,
      color=alt.condition(genre_filter, "top15TagValue", alt.value("whitesmoke")),
  ).properties(
      width=600,
      height=600,
      selection=nearest)
  text = alt.Chart().mark_text(align='left', dx=5, dy=-5).encode(
      x=x,
      y=y,
      text=alt.condition(nearest, 'name', alt.value('')))
  return alt.hconcat(alt.layer(base, text), genre_chart, data=data)

def tsne_movie_embeddings(model):
  """Visualizes the movie embeddings, projected using t-SNE with Cosine measure.
  Args:
    model: A MFModel object.
  """
  tsne = sklearn.manifold.TSNE(
      n_components=2, perplexity=40, metric='cosine', early_exaggeration=10.0,
      init='pca', verbose=True, n_iter=400)

  print('Running t-SNE...')
  V_proj = tsne.fit_transform(model.embeddings["artistID"])
  artists.loc[:,'x'] = V_proj[:, 0]
  artists.loc[:,'y'] = V_proj[:, 1]
  return visualize_movie_embeddings(artists, 'x', 'y')
tsne_movie_embeddings(softmax_model)
```

```
def artist_embedding_norm(models):
  """Visualizes the norm and number of ratings of the artist embeddings.
  Args:
    model: A train_matrix_norm object.
  """
  if not isinstance(models, list):
    models = [models]
    df = pd.DataFrame({
          'name': artists.sort_values('artistID', ascending=True)['name'].values,
        'number of user-artist interactions': user_artists[['artistID','userID']].
↪sort_values('artistID', ascending=True).groupby('artistID').count()['userID'].
↪values,
    })
  charts = []
  brush = alt.selection_interval()
  for i, model in enumerate(models):
    norm_key = 'norm'+str(i)
    df[norm_key] = np.linalg.norm(model.embeddings["artistID"], axis=1)
    nearest = alt.selection(
        type='single', encodings=['x', 'y'], on='mouseover', nearest=True,
```

(continues on next page)

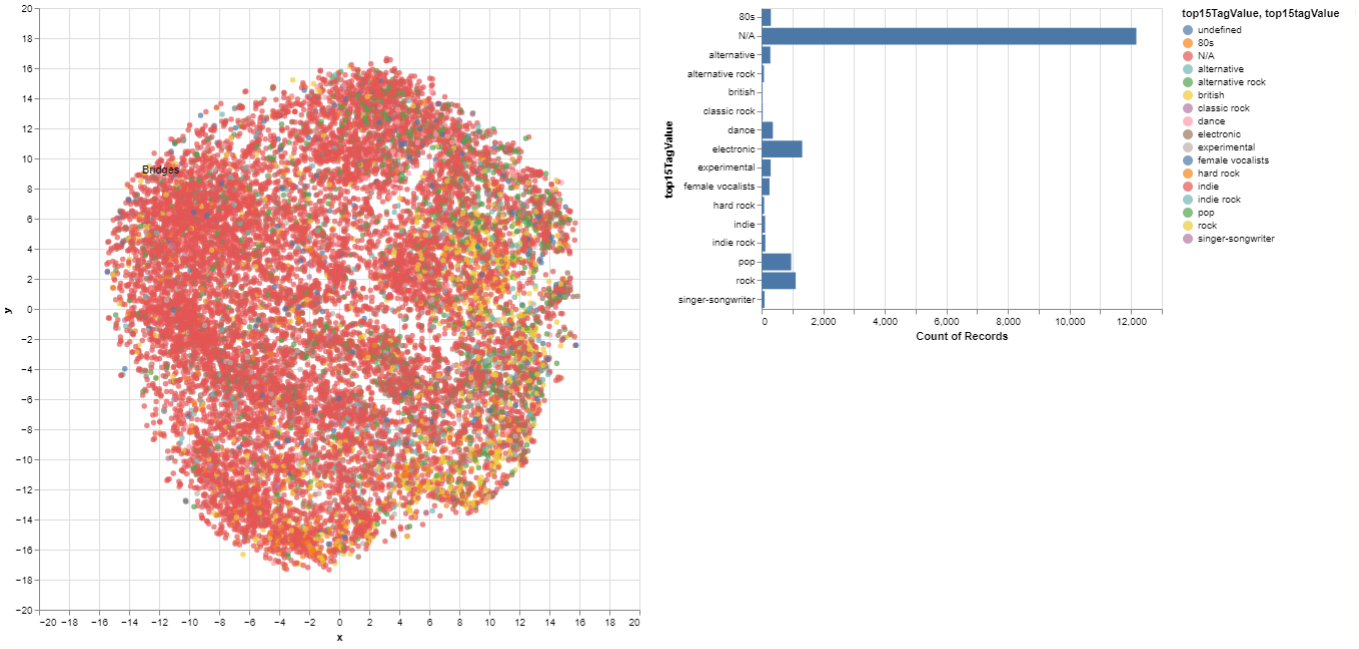


Fig. 3.2: T-SNE projection of softmax model weights

(continued from previous page)

```
        empty='none')
    base = alt.Chart().mark_circle().encode(
        x='number of user-artist interactions',
        y=norm_key,
        color=alt.condition(brush, alt.value('#4c78a8'), alt.value('lightgray'))
    ).properties(
        selection=nearest).add_selection(brush)
    text = alt.Chart().mark_text(align='center', dx=5, dy=-5).encode(
        x='number of user-artist interactions', y=norm_key,
        text=alt.condition(nearest, 'name', alt.value('')))
    charts.append(alt.layer(base, text))
  return alt.hconcat(*charts, data=df)

artist_embedding_norm(softmax_model)
```

Additionally, when the artist embeddings are visualised they do not demonstrate much clustering unlike the matrix factorisation methods. The norm of the embeddings does not also correlate with the popularity of an artist.

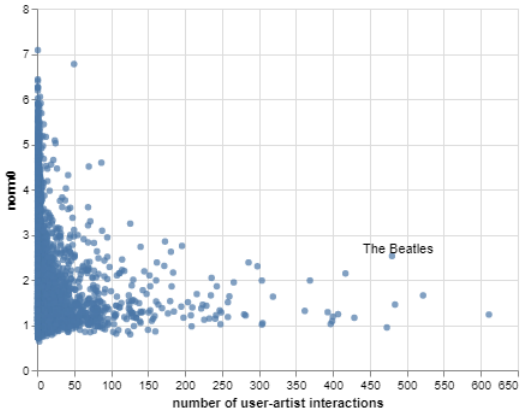


Fig. 3.3: Norm of softmax weights (artist embeddings) vs popularity

3.4 Demo

You can find the most similar artist to a specified artist using the `artist_neighbours()` function.

```
artist_neighbors(softmax_model, 'The Cure', COSINE)
```

```
Nearest neighbors of : The Cure.
```

	cosine score	name	most assigned tag
161729	1.000000	Firebug	ska
162859	0.920241	Taylor Lautner	N/A
165317	0.895609	Rojo	N/A
161368	0.880020	Perrey & Kingsley	N/A
395	0.872602	Behemoth	death metal
164005	0.861260	Chris Crocker	very very guilty pleasure

Overall. our softmax model generates poor reccomendations. We tried experimenting with different learning rates and hidden layers but we did not observe any increased performance. Therfore, we decide not to proceed with evaluating this model with the other models already specified in this report.

CHAPTER

FOUR

EVALUATION

One of the key requirements in developing a recommendation algorithm is to measure it's effectiveness. Typical metrics include mean squared error, root mean squared error, precision and recall. It must be noted that evaluating performance on a fixed test dataset will be biased as it is impossible to accurately predict the reactions of real users to the recommendations. Hence, any metric computed on static data will be some-what imprecise. Of course, in real-world settings, this is overcome by online A/B tests.

In this investigation, we set a small amount of users as hold-out for evaluation, fitting a model to all of the remaining users and artists. From these 'evaluation users', 15 of their interactions data is set as a hold-out test set .Then , the top-k recommendations for each user are collected. These top-k artists are compared against the hold-out test items, seeing how well they do at ranking them near the top vs. how they rank the remainder of the items.

The fact that the data is implicit, motivates us to choose evaluation criteria that is akin to evaluating information retrieval systems. We view the problem as ranking or classification instead of regression task, with the models being evaluated not by how well they perform at predicting listening counts, but by how good they are at scoring the observed interactions higher than the non-observed interactions for each user. The metrics we will use include:

- *P@k ("precision-at-k")*: This metric calculates the proportion of the top-k recommendations that include items from the test set for a given user - i.e.

$$P@k = \frac{1}{k}\sum_{i=1}^{k}\begin{cases}1 & \text{if } r_i \in \tau \\ 0 & \text{otherwise}\end{cases}$$

Where r_i is the item ranked at position i by the model (sorting the predicted scores in descending order), and τ is the set of items that are in the test set for that user. Then, we calculate the mean and standard deviation of these values to allow for comparison between the models.

- R@k ("recall-at-k"): while P@k captures what a recommender system aims at being good at, it does not capture the fact that, the more test items there are, the higher the chances that they will be included in the top-K recommendations. The test set includes 15 items for each user. This metric examines the proportion of the test artists would have been retrieved with the top-k recommended list:

$$R@k = \frac{1}{|\tau|}\sum_{i=1}^{k}\begin{cases}1 & \text{if } r_i \in \tau \\ 0 & \text{otherwise}\end{cases}$$

Again, we take the mean and standard deviation of this metric to allow comparison.

- Hit@k ("Hit Rate at k"): This is a simple Binary metric that examines whether any of the top-k recommended artists were in the test set for a given user:

$$\max_{i=1\ldots k} = \begin{cases}1 & \text{if } r_i \in \tau \\ 0 & \text{otherwise}\end{cases}$$

Again, we take the mean and standard deviation of this metric to allow comparison.

A simple baseline is used. Each user is assigned the top 15 artists by user-artist interaction.

4.1 Quick links:

4.2 Setup

In this section, we import the required data files and libraries. We also define the functions required for each metric. Please view the function string for more in-depth explanation.

```
import pandas as pd
import ast
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

vanilla_dot_pred = pd.read_csv('data/evaluation/vannila_dot_pred.csv')
vanilla_cos_pred = pd.read_csv('data/evaluation/vanila_cos_pred.csv')
reg_dot_pred = pd.read_csv('data/evaluation/reg_dot_pred.csv')
reg_cos_pred = pd.read_csv('data/evaluation/reg_cos_pred.csv')
test_set = pd.read_csv('data/evaluation/test-set.csv')
user_taggedartists = pd.read_csv(r'data/user_taggedartists-timestamps.dat', sep='\t')
user_artists = pd.read_csv('data/user_artists.dat', sep='\t')
```

Constructing baseline

```
def construct_baseline(user_artists, test_set):
    """Constructs a simple baseline.
        Each user is assigned the top 15 artist by user-artist interaction"""
    top_artists = str(user_artists[['artistID','userID']].groupby('artistID').count().
 ↪reset_index().sort_values(["userID"], ascending=False).head(15)["artistID"].values.
 ↪tolist())
    baseline_df = pd.DataFrame({'userID':test_set.userID.unique(),'predictions_artists
 ↪':([top_artists] * len(test_set.userID.unique()))})
    return baseline_df
basline_model = construct_baseline(user_artists, test_set)
```

```
def calculate_precision(model_recommended_items,test_set, k, unique_users=test_set.
 ↪userID.unique()):
    """Calculate precision as defined above
    Args:
        model_recommended_items: a dataframe containing userIDs, their top 15 artists␣
 ↪by a defined model, and the correspnding scores.
        test_Set: the test set as defined above, contains 15 items
    Returns:
        the mean and standard deviation of the precision values of the users
    """
    precision_score_list = []
    model_recommended_items = model_recommended_items[model_recommended_items.userID.
 ↪isin(test_set.userID.values)]
    for curr_userID in unique_users:
        recommended_artistIDs =  ast.literal_eval(model_recommended_items[model_
 ↪recommended_items.userID == curr_userID]["predictions_artists"].values[0])[0:k]
        test_artist_ids = ast.literal_eval(test_set[test_set.userID == curr_userID][
 ↪"holdout_artists"].values[0])
        common_count = len(set(recommended_artistIDs).intersection(set(test_artist_
 ↪ids)))
        precision_score_list.append(common_count/k)

    return (np.mean(precision_score_list), np.std(precision_score_list))



def calculate_recall(model_recommended_items,test_set, k, unique_users=test_set.
 ↪userID.unique()):
    """Calculate recall as defined above
    Args:
        model_recommended_items: a dataframe containing userIDs, their top 15 artists␣
 ↪by a defined model, and the correspnding scores.
        test_Set: the test set as defined above, contains 15 items
    Returns:
        the mean and standard deviation of the recall values of the users
"""
    recall_score_list = []
    model_recommended_items = model_recommended_items[model_recommended_items.userID.
 ↪isin(test_set.userID.values)]
    for curr_userID in unique_users:
        recommended_artistIDs =  ast.literal_eval(model_recommended_items[model_
 ↪recommended_items.userID == curr_userID]["predictions_artists"].values[0])[0:k]
        test_artist_ids = ast.literal_eval(test_set[test_set.userID == curr_userID][
 ↪"holdout_artists"].values[0])
        common_count = len(set(recommended_artistIDs).intersection(set(test_artist_
 ↪ids)))
```

(continues on next page)

(continued from previous page)

```
        recall_score_list.append(common_count/ len(test_artist_ids))

    return (np.mean(recall_score_list), np.std(recall_score_list))

def calculate_hit_rate(model_recommended_items,test_set, k, unique_users=test_set.
↪userID.unique()):
    """Calculate hit-rate as defined above
    Args:
        model_recommended_items: a dataframe containing userIDs, their top 15 artists␣
↪by a defined model, and the correspnding scores.
        test_Set: the test set as defined above, contains 15 items
    Returns:
        the mean and standard deviation of the hit-rate values of the users
    """
    hit_rate_score_list = []
    model_recommended_items = model_recommended_items[model_recommended_items.userID.
↪isin(test_set.userID.values)]
    for curr_userID in unique_users:
        recommended_artistIDs =  ast.literal_eval(model_recommended_items[model_
↪recommended_items.userID == curr_userID]["predictions_artists"].values[0])[0:k]
        test_artist_ids = ast.literal_eval(test_set[test_set.userID == curr_userID][
↪"holdout_artists"].values[0])
        common_count = len(set(recommended_artistIDs).intersection(set(test_artist_
↪ids)))
        if common_count > 0:
            common_count = 1
        hit_rate_score_list.append(common_count)
    return (np.mean(hit_rate_score_list), np.std(hit_rate_score_list))
```

```
model_names_variable_dict = {
    "Baseline": basline_model,
    "SGD_Matrix_Fact_vanilla_dot": vanilla_dot_pred,
    "SGD_Matrix_Fact_vanilla_cosine":vanilla_cos_pred,
    "SGD_Matrix_Fact_regularized_dot":reg_dot_pred,
    "SGD_Matrix_Fact_regularized_cosine":reg_cos_pred,
#    "Softmax_dot": None,
 #   "Softmax_cosine":None
}
```

4.3 Overall Evaluation

Now, we construct the evaluation matrix. In this evaluation matrix, the index is the the various model configurations in the format of *<Model_name>_<Similairty_meausre>*. The columns specify the metrics used in the format *<metric_name>*at*<k_value>*__*<mean/standard_deviation>*. An individual evaluation matrix is computed for each value of k to keep it presentable.

```
# create evaluation matrix
def create_evaluation_matrix_at_k(k):
    """create evualtion matrix for a value of k"""
    evaluation_matrix = pd.DataFrame()
    for model_name, variable_name in model_names_variable_dict.items():
        temp_dict = {}
        temp_dict['model_name'] = model_name
```

(continues on next page)

(continued from previous page)

```
        temp_dict[f'p_at_{k}_mean'] =  [calculate_precision(variable_name, test_set,␣
↪k)[0]]
        temp_dict[f'r_at_{k}_mean'] =  [calculate_recall(variable_name, test_set,␣
↪k)[0]]
        temp_dict[f'hr_at_{k}_mean'] =  [calculate_hit_rate(variable_name, test_set,␣
↪k)[0]]
        temp_dict[f'p_at_{k}_stdv'] = [calculate_precision(variable_name, test_set,␣
↪k)[1]]
        temp_dict[f'r_at_{k}_stdv'] = [calculate_recall(variable_name, test_set,␣
↪k)[1]]
        temp_dict[f'hr_at_{k}_stdv'] = [calculate_hit_rate(variable_name, test_set,␣
↪k)[1]]
        temp_df = pd.DataFrame(temp_dict)
        evaluation_matrix = pd.concat([evaluation_matrix, temp_df])
    return evaluation_matrix
```

4.3.1 Evaluation matrix at k=5

```
create_evaluation_matrix_at_k(5)
```

```
                          model_name  p_at_5_mean  r_at_5_mean  hr_at_5_mean  \
0                           Baseline     0.031240     0.010413      0.141170
0         SGD_Matrix_Fact_vanilla_dot     0.033387     0.011129      0.144928
0      SGD_Matrix_Fact_vanilla_cosine     0.257112     0.085704      0.713366
0     SGD_Matrix_Fact_regularized_dot     0.082662     0.027554      0.341922
0  SGD_Matrix_Fact_regularized_cosine     0.268062     0.089354      0.758454

   p_at_5_stdv  r_at_5_stdv  hr_at_5_stdv
0     0.081260     0.027087      0.348197
0     0.087315     0.029105      0.352028
0     0.221936     0.073979      0.452189
0     0.126538     0.042179      0.474353
0     0.210525     0.070175      0.428020
```

4.3.2 Evaluation matrix at k=10

```
create_evaluation_matrix_at_k(10)
```

```
                          model_name  p_at_10_mean  r_at_10_mean  \
0                           Baseline      0.034836      0.023224
0         SGD_Matrix_Fact_vanilla_dot      0.033924      0.022616
0      SGD_Matrix_Fact_vanilla_cosine      0.218894      0.145930
0     SGD_Matrix_Fact_regularized_dot      0.084326      0.056218
0  SGD_Matrix_Fact_regularized_cosine      0.247343      0.164895

   hr_at_10_mean  p_at_10_stdv  r_at_10_stdv  hr_at_10_stdv
0       0.276973      0.063148      0.042099       0.447503
0       0.250671      0.067292      0.044861       0.433399
0       0.845947      0.160707      0.107138       0.360999
0       0.566828      0.092157      0.061438       0.495514
0       0.893720      0.160956      0.107304       0.308196
```

4.3.3 Evaluation matrix at k=15

```
create_evaluation_matrix_at_k(15)
```

```
                          model_name  p_at_15_mean  r_at_15_mean  \
0                           Baseline      0.035033      0.035033
0         SGD_Matrix_Fact_vanilla_dot      0.036965      0.036965
0      SGD_Matrix_Fact_vanilla_cosine      0.189873      0.189873
0     SGD_Matrix_Fact_regularized_dot      0.083342      0.083342
0  SGD_Matrix_Fact_regularized_cosine      0.227232      0.227232

   hr_at_15_mean  p_at_15_stdv  r_at_15_stdv  hr_at_15_stdv
0       0.380569      0.052979      0.052979       0.485527
0       0.348363      0.062706      0.062706       0.476452
0       0.908749      0.129153      0.129153       0.287965
0       0.703704      0.076630      0.076630       0.456623
0       0.941492      0.133888      0.133888       0.234701
```

From these metrics, the best-performing models seem to be those that use the cosine similarity measure rather than the dot product. Cosine similarity only cares about *angle difference*, where as dot product cares about *angle and the magnitude*. Since artists that are more mainstream, i.e. have more user interactions, tend to have a higher norms, we hypothesis that models that incorporate this metric become biased towards mainstream artists and therefore fail to recommended niche artists. On the other hand, the dot product is less expensive (in terms of complexity and implementation). It is safe to conclude that the addition of regularization on the SGD Matrix Factorization model provided it with a boost across all examined metrics. As we explained previously, regularization helps mitigate the "rich-get-richer" effect for popular artists. If our vanilla SGD matrix factorization formula was deployed into production, there is a risk users could become stuck in an "information confinement area", leading to poor user experience and lost revenue. Regularization helps to mitigate against the model paying too much attention towards training data (low bias & high variance) which can lead to poor generalization.

For the models that incorporate the cosine similarity, the precision value decreases as k increases. This is the desired result (rather than the inverse) as the objective of a good recommendation system is supply relevant recommendations near the top. We believe users are unlikely to look at more than the top five recommendations. It is interesting to note that the spread of precision values decreases as k increases for these models. This is an undesired result as they could be outliers biasing our mean value for earlier values of k and our subsequent conclusions.

For the models that incorporate the dot product, the precision metrics remains relatively similar. This indicates the same proportion of relevant recommendations are produced for different values of K. Therefore, we would prefer the models that incorporated cosine similarity in production.

Recall increase as K increases for most models. This is expected as there are more recommendations. For instance, the maximum recall value we can have @ 5 is 1/3, where @10 is 2/3. Comparing the model recall values and the maximum recall available, the ability of these models to recommend relevant recommendations out of the relevant set is relatively proportionate across all K-values. Although the standard deviation of recall for most models increases as k increases.

Hit rate is a simple metric, and increases as k increases due to the increasing number of recommendations the model is allowed to make. The standard deviation of this metric decrease as k increases.

4.4 Drill Down Deeper

As mentioned in the introduction section, just computing evaluation metrics across all users does not truly capture the effectiveness of a recommendation system. We decide to drill down deeper, viewing users in three separate ways based on their behaviour.

- *Diversity*: We are interested in how our systems perform for users with an uniform listening taste vs users with a varied taste. We attach a diversity score to each user.

$$D_i = \frac{1}{|J|} \sum_{j=1}^{|J|} tc_j$$

where J is the set of artists that user i listened to and tc_j represents the total number of unique tags (assigned by **all** users) belonging to artist j. Of course, there is a cavet to this formulation, in that user's who listen to more artists will generally have have a higher score. This score is then used to divide the users into four distinct groups.

- *"Following the crowd" users*: Users are divided into four groups of approximently equal size, depending on how mainstream their listening habitats are. First, we caulate the mainstreamness of an artist j:

$$M_j = \frac{1}{M} \sum_{i=1}^{M} \begin{cases} 1 & \text{if } u_i \in \tau \\ 0 & \text{otherwise} \end{cases}$$

where τ represents an artist's audience and M is the the total number of users on the Last.fm Website. User i's mainstream listening habitat (MH_i) is then calculated as follows:

$$MH_i = \frac{1}{J} \sum_{j=1}^{J} M_j$$

where J is the set of artists that user i listened to.

- *Activity on Last.fm*: Users are divided into four separate clusters of approximately equal size depending on their activity on the last.fm website. "Activity" is calculated for user i as follows:

$$A_i = \text{Total Number of plays}$$

For all semantic categories, quartile ranges Q_1, Q_2, Q_3 are utilised as cutting off points for their respective groups.

We now define helper functions to aid in this process.

```
def catch(artist, desired_dict):
    "Not every artistID has a tag "
    try:
        return desired_dict[artist]
    except KeyError:
        return 0

def calculate_diversity_users(user_artists, user_taggedartists):
    """Calculate diversity score for each user
    Returns:
        a dict containing user id and diversity score
    """
    user_artists = user_artists[user_artists.userID.isin(test_set.userID.unique())]
    artist_tag_count_dict = user_taggedartists[["artistID", "tagID"]].groupby(
↪"artistID")["tagID"].nunique().to_dict()
    user_diversity_dict = {}
    for curr_userID in test_set.userID.unique():
        J = user_artists[user_artists.userID == curr_userID]['artistID'].unique()
        j_len = len(J)
```

(continues on next page)

(continued from previous page)

```
        tc = sum([catch(artist,artist_tag_count_dict) for artist in J])
        user_diversity_dict[curr_userID] = tc/j_len
    return user_diversity_dict

def calculate_mainstream_users(user_artists):
    """Calculate mainstream score for each user
    Returns:
        a dict containing user id and mainstream score
    """
    user_artists = user_artists[user_artists.userID.isin(test_set.userID.unique())]
    M = len(test_set.userID.unique())
    user_mainstream_dict = {}
    artist_user_count_dict =  user_artists[["artistID", "userID"]].groupby("artistID
↪")["userID"].nunique().to_dict()
    artist_user_count_dict = {k: v / M for k, v in artist_user_count_dict.items()}
    for curr_userID in test_set.userID.unique():
        J = user_artists[user_artists.userID == curr_userID]['artistID'].unique()
        j_len = len(J)
        tc = sum([artist_user_count_dict[artist] for artist in J])
        user_mainstream_dict[curr_userID] = tc/j_len
    return user_mainstream_dict

def calculate_activity(user_artists):
    """Calculate activity score for each user
    Returns:
        a dict containing user id and activity score
    """
    user_artists = user_artists[user_artists.userID.isin(test_set.userID.unique())]
    user_activity_dict = user_artists[["userID", "weight"]].groupby("userID").sum().
↪to_dict()['weight']
    return user_activity_dict

def split_dict_into_four_groups(score_dict):
    """splits users IDS into four groups according to their score
            the 25th,50th,75th percentiles serve as splitting points
    Returns:
        four lists, one for each group
    """
    Q1,Q2,Q3 = np.percentile(sorted(score_dict.values()), [25,50,75])
    group_1_ids, group_2_ids, group_3_ids, group_4_ids = [], [] ,[], []
    for user_id, score in score_dict.items():
        if score < Q1:
            group_1_ids.append(user_id)
        elif score >= Q1  and score < Q2:
            group_2_ids.append(user_id)
        elif score >= Q2  and score < Q3:
            group_3_ids.append(user_id)
        elif score >= Q3:
            group_4_ids.append(user_id)
    return group_1_ids, group_2_ids, group_3_ids, group_4_ids
```

```
def visualize_category_evaluation(groups, metric, k):
    """Does pre-processing to create a dataframe containing
    Returns:
        four lists, one for each group
    """
```

(continues on next page)

(continued from previous page)

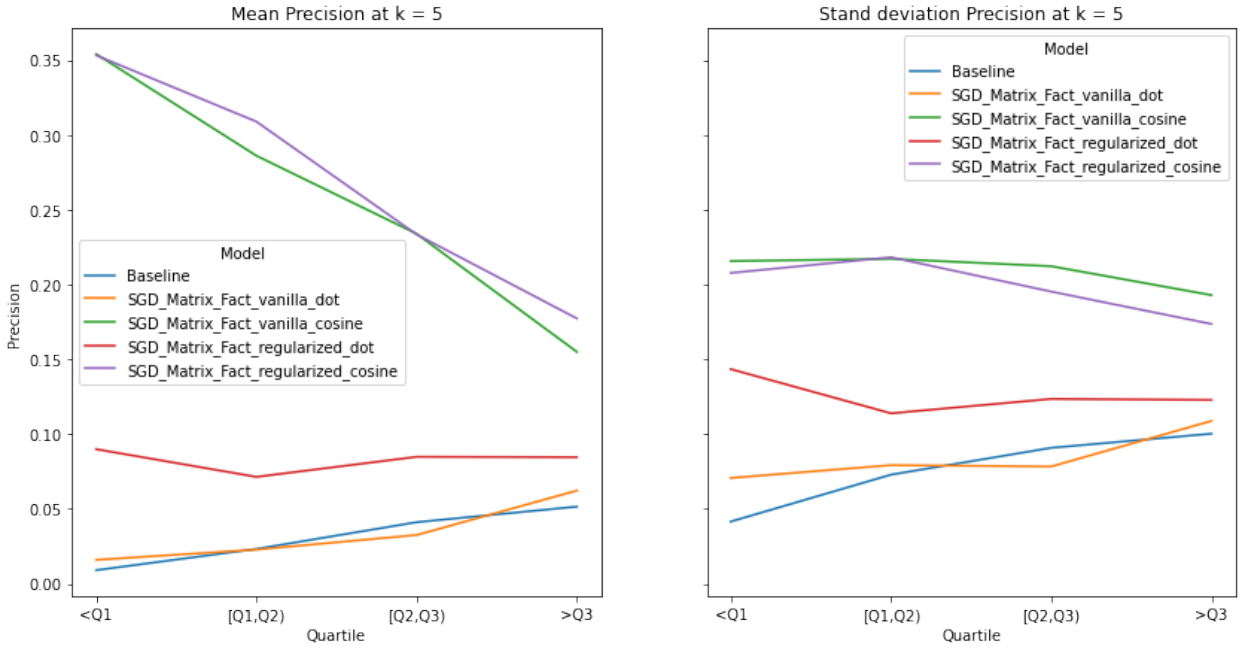
```
    metric_func_dict = {
        'Precision': calculate_precision,
        'Recall': calculate_recall,
        'Hit Rate': calculate_hit_rate
    }
    category_df = pd.DataFrame()
    for model_name, variable_name in model_names_variable_dict.items():
        model_df = pd.DataFrame()
        for group, qauntile in zip(groups, ['<Q1', '[Q1,Q2)','[Q2,Q3)', '>Q3']):
            temp_dict = {}
            temp_dict['Model'] = model_name
            temp_dict['Quartile'] = qauntile
            temp_dict[f'{metric}_at_{k}_mean'] =  [metric_func_dict[metric](variable_
↪name, test_set, k, group)[0]]
            temp_dict[f'{metric}_at_{k}_stdv'] = [metric_func_dict[metric](variable_
↪name, test_set, k, group)[1]]
            temp_df = pd.DataFrame(temp_dict)
            model_df = pd.concat([model_df, temp_df], ignore_index=True)
        category_df = pd.concat([category_df,model_df ], ignore_index=True)

    fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(14, 7), sharey=True)
    sns.lineplot(data=category_df, x="Quartile", y=f'{metric}_at_{k}_mean', hue="Model
↪", ax=ax1).set(title=f'Mean {metric} at k = {k}', ylabel=f'{metric}')
    sns.lineplot(data=category_df, x="Quartile", y=f'{metric}_at_{k}_stdv', hue="Model
↪", ax=ax2).set(title=f'Stand deviation {metric} at k = {k}', ylabel=f'{metric}')
```

4.4.1 Diversity

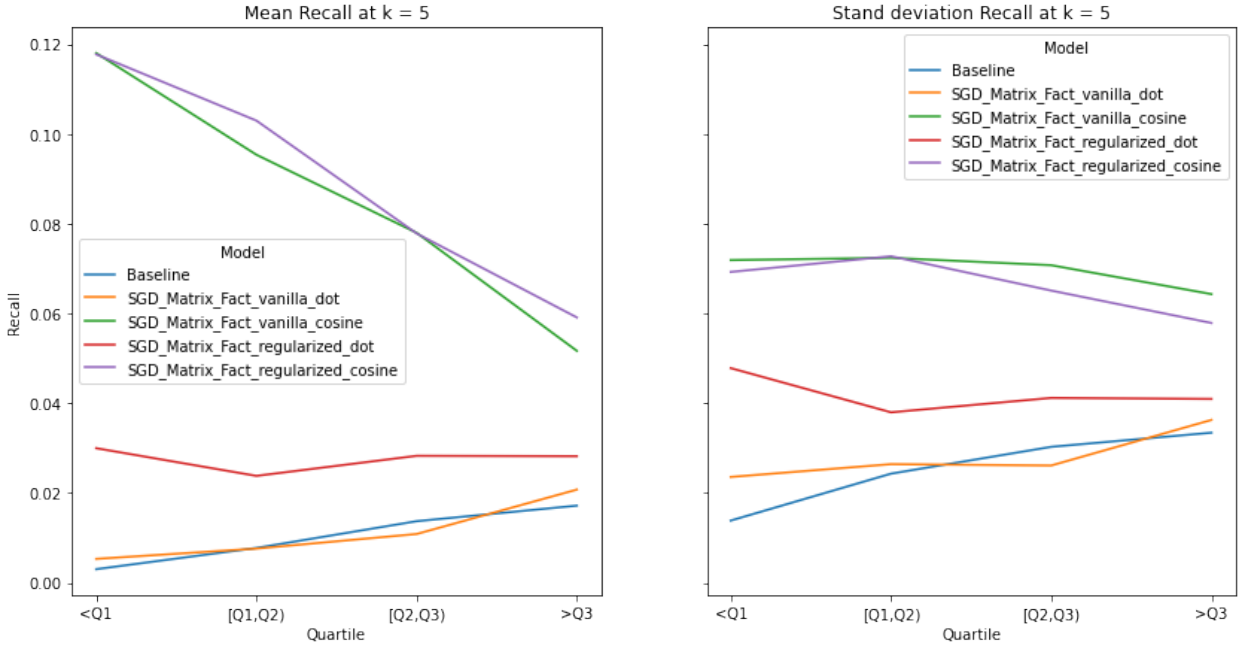
First, we plot the mean and standard deviation of the various evaluation metric values (described above), for each diversity group. **Note**, the group's (defined on the x axis) collective diversity score increases as we move along the axis. For instance, the group labelled as "<Q1" represents a cluster of user IDs, where all of the user's diversity scores are less than the 25th global percentile.

Diversity K=5

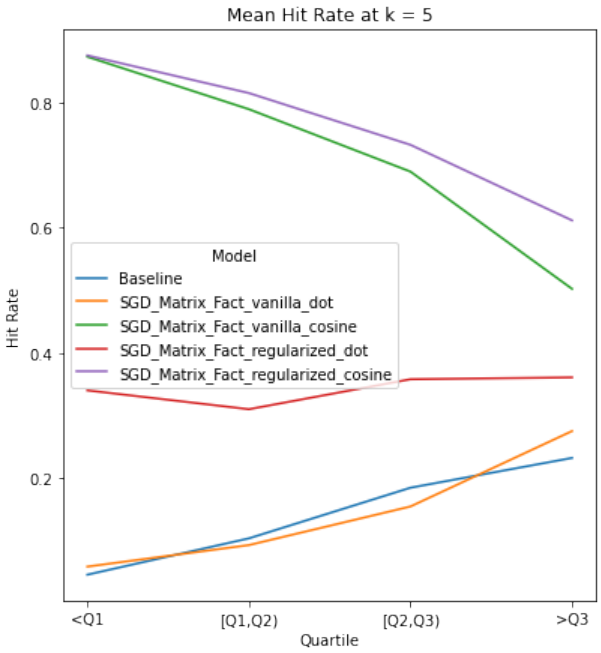
```
user_diversity_dict = calculate_diversity_users(user_artists, user_taggedartists)
visualize_category_evaluation(split_dict_into_four_groups(user_diversity_dict),
↪'Precision', k=5)
```

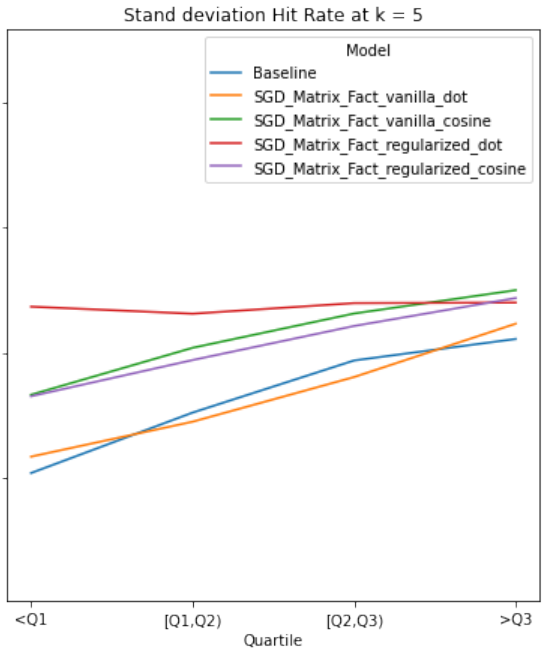



```
visualize_category_evaluation(split_dict_into_four_groups(user_diversity_dict),
↪'Recall', k=5)
```



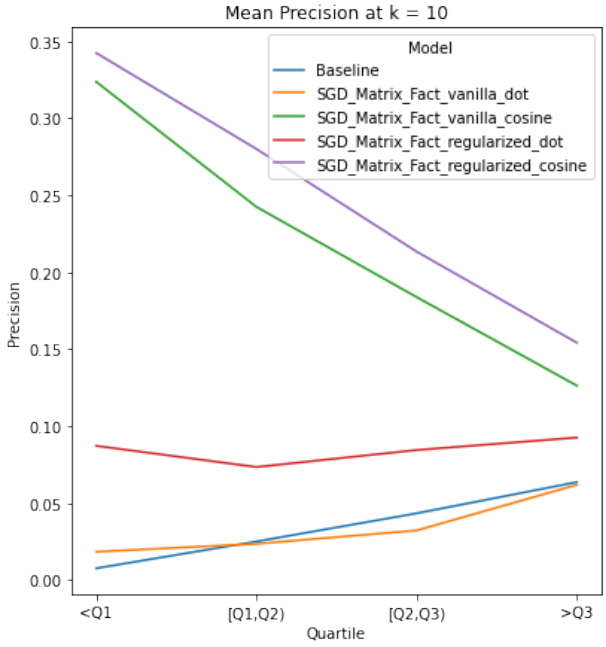

```
visualize_category_evaluation(split_dict_into_four_groups(user_diversity_dict), 'Hit␣
↪Rate', k=5)
```

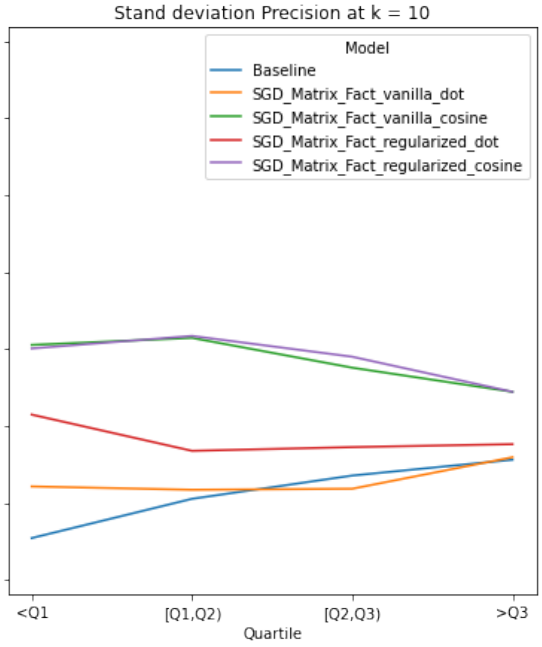




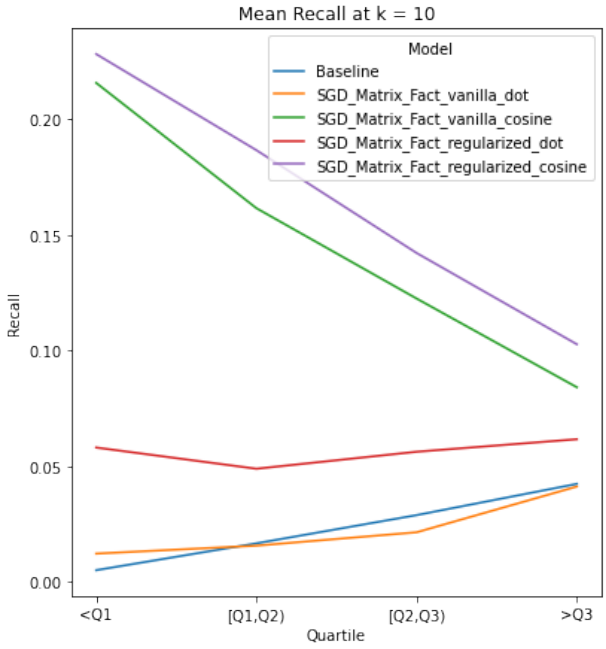
Diversity K=10

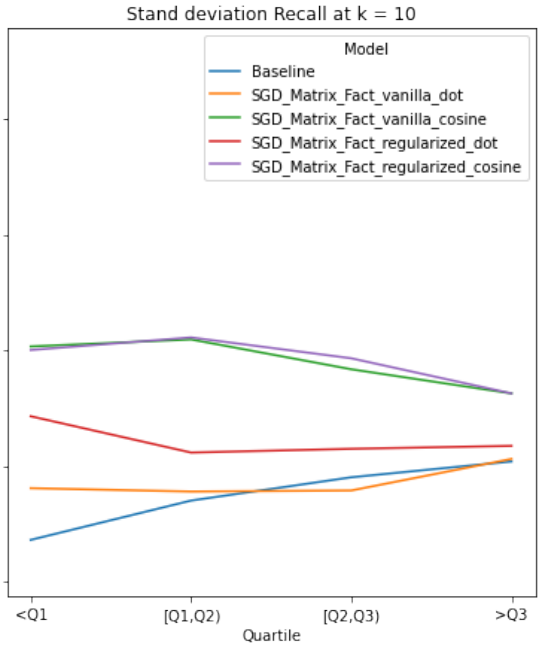
```
visualize_category_evaluation(split_dict_into_four_groups(user_diversity_dict),
 ↪'Precision', k=10)
```



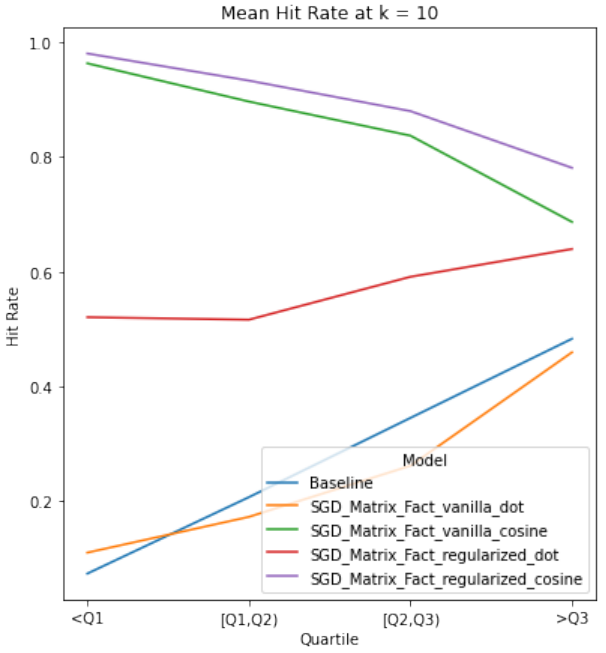


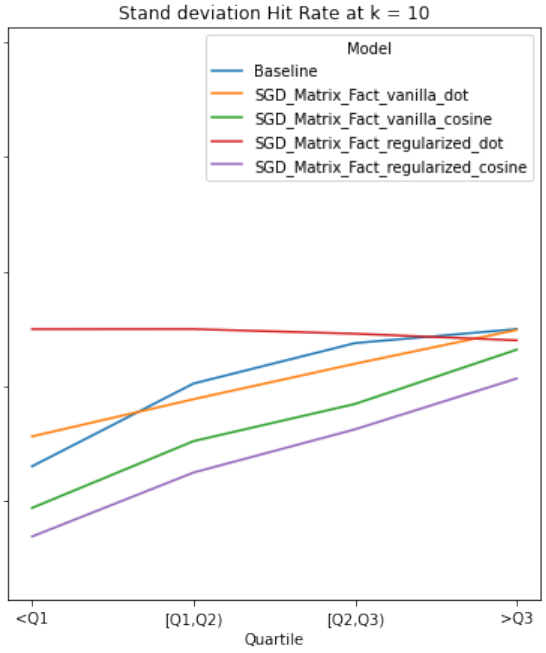

```
visualize_category_evaluation(split_dict_into_four_groups(user_diversity_dict),
 ↪'Recall', k=10)
```





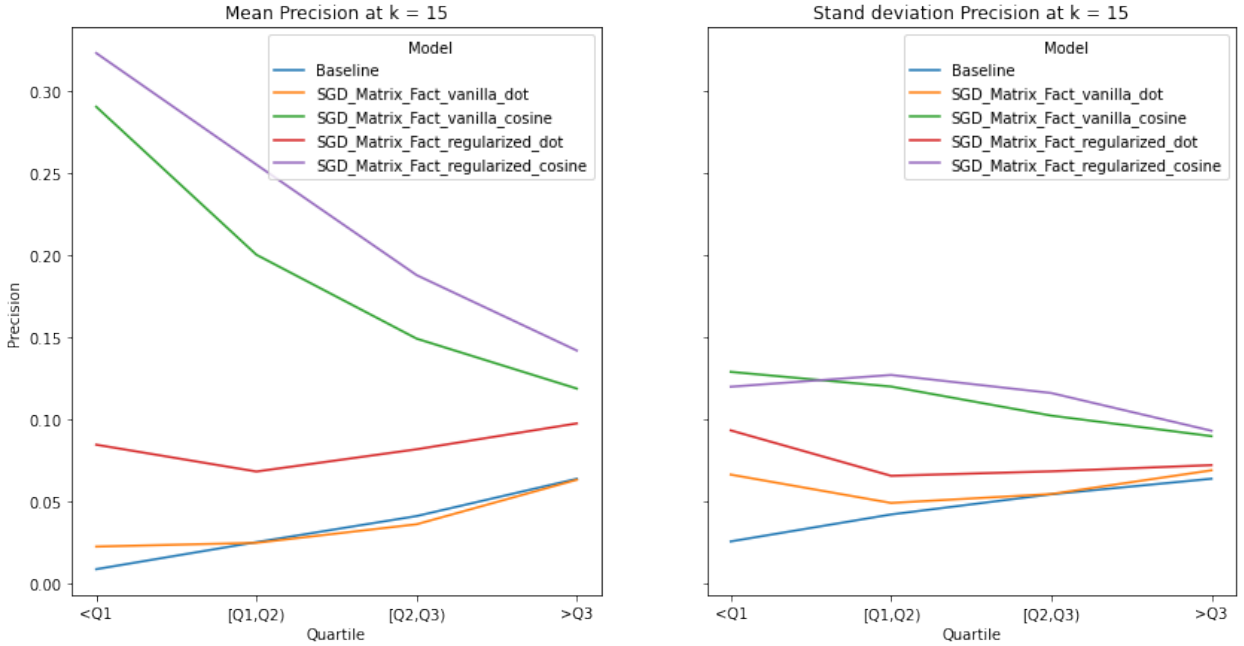

```
visualize_category_evaluation(split_dict_into_four_groups(user_diversity_dict), 'Hit␣
↪Rate', k=10)
```



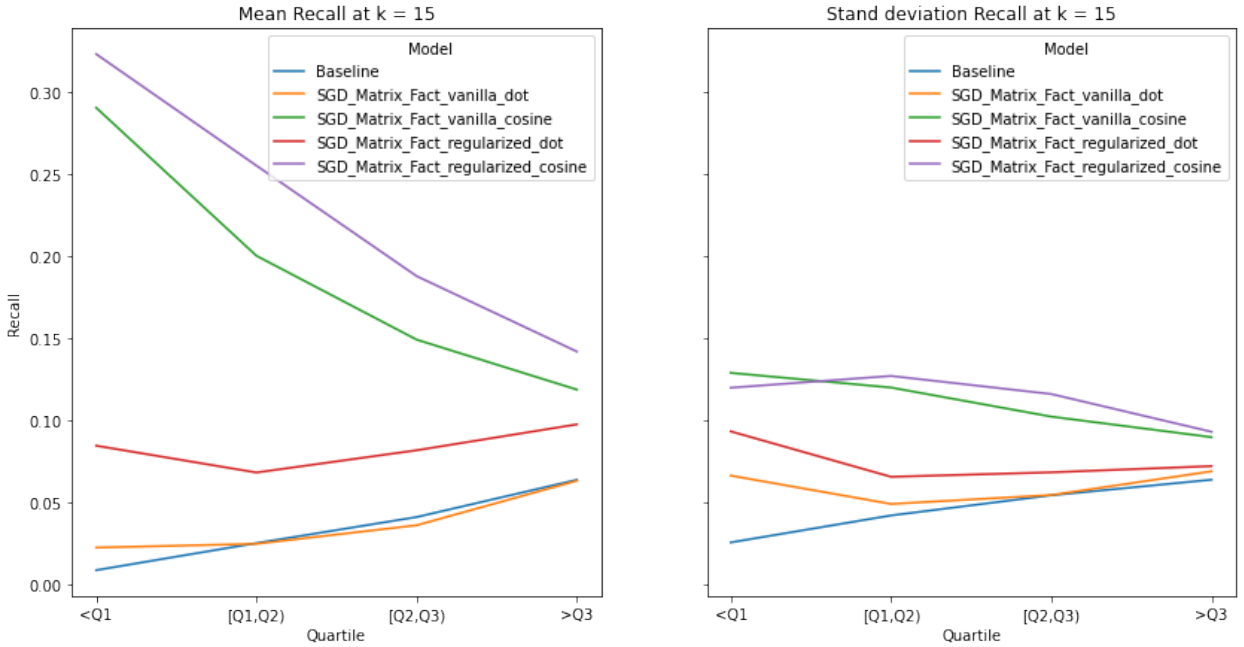


Diversity K=15

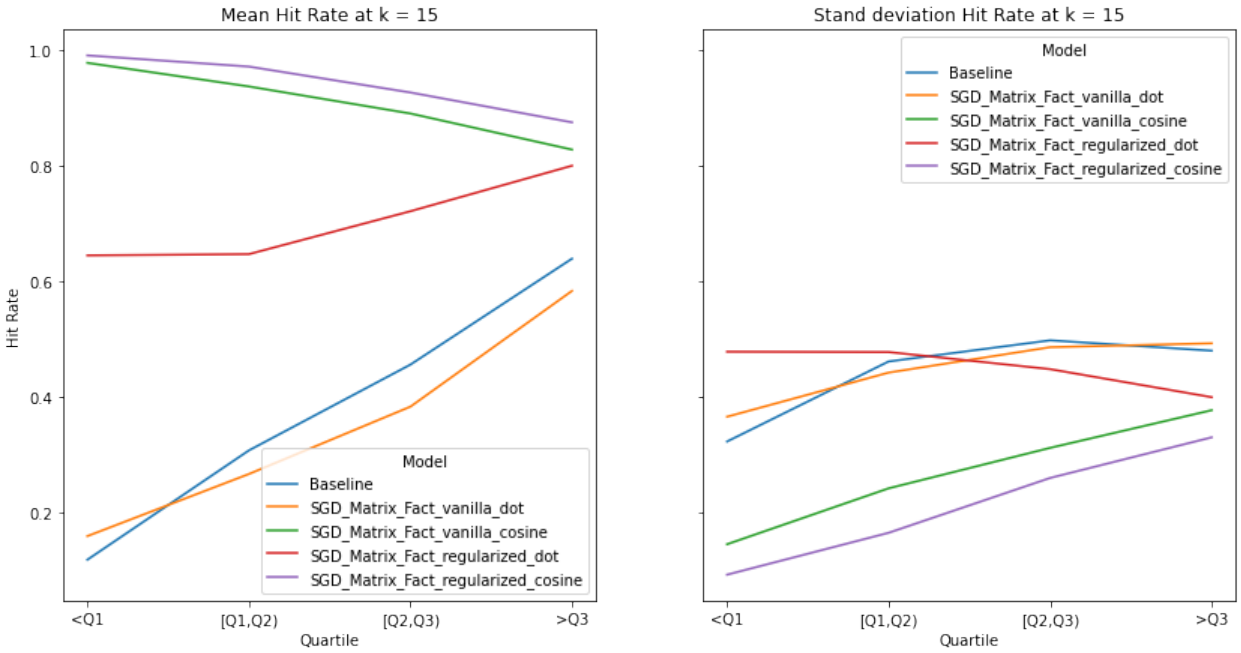
```
visualize_category_evaluation(split_dict_into_four_groups(user_diversity_dict),
↪'Precision', k=15)
```




```
visualize_category_evaluation(split_dict_into_four_groups(user_diversity_dict),
↪'Recall', k=15)
```




```
visualize_category_evaluation(split_dict_into_four_groups(user_diversity_dict), 'Hit␣
↪Rate', k=15)
```



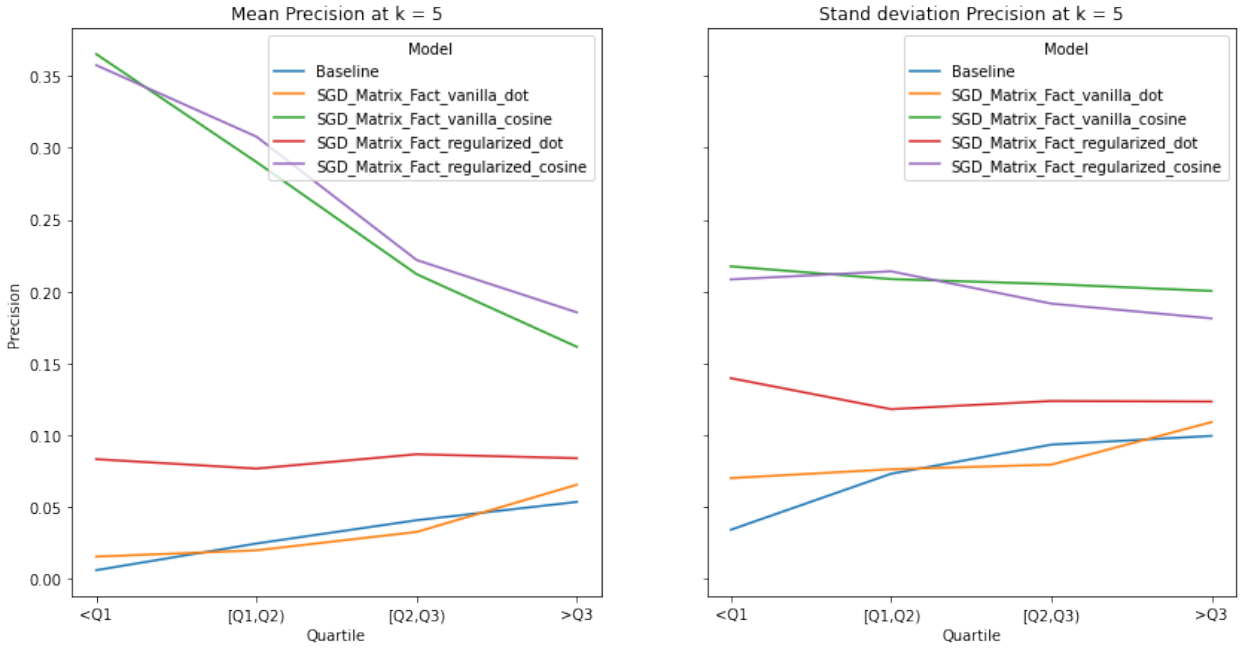
Music habits vary considerably across users. An ideal recommender system should be accurately meet the requirements of *all users*. There is a trend across all examined metrics that users with more uniform tastes are better served than those with a diverse set of tastes. We also notice similar observations as our overall evaluation, in terms of the trends of the precision, recall and hit-rate values for each value of K. The spread of the evaluation metrics either increases or stays the same as user's listening habitats become more diverse.

4.4.2 Mainstream

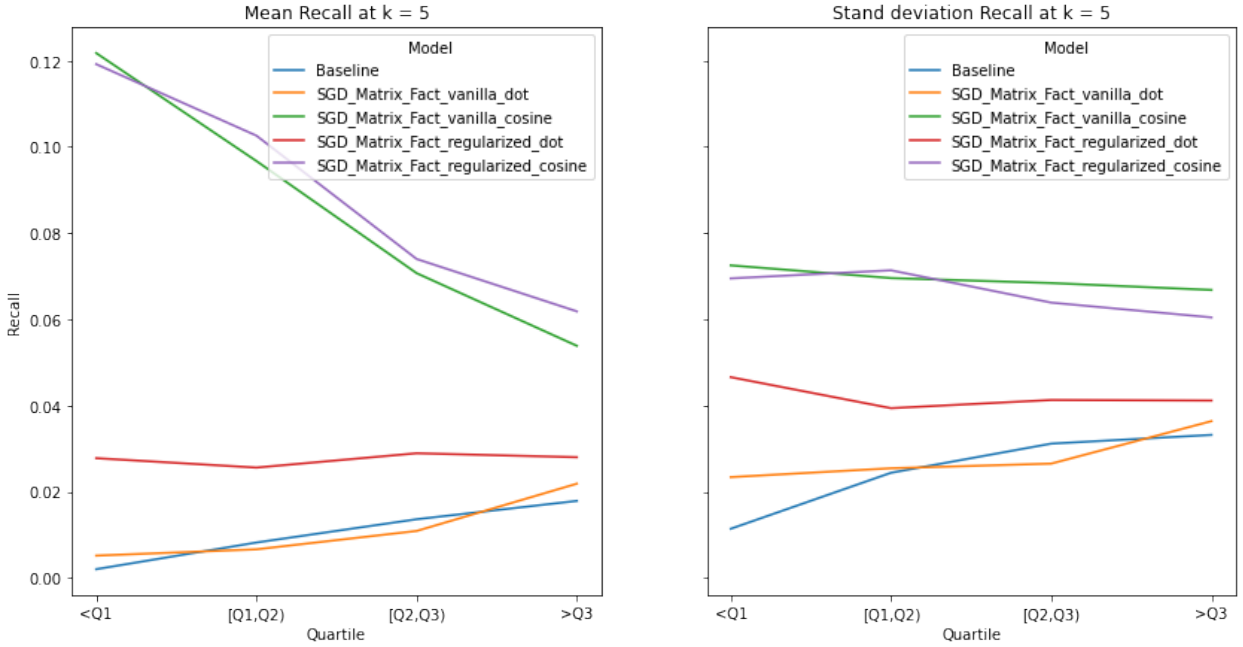
Next, we plot the mean and standard deviation of the various evaluation metric values (described above), for each group of varying 'mainstreamness'. **Note**, the group's (defined on the x axis) collective mainstream score increases as we move along the axis. For instance, the group labelled as "[Q1,Q2)" represents a group of user IDs, where all of the user's mainstream scores are less than the 50th global percentile, but greater than or equal to the 25th percentile.

Mainstream K = 5

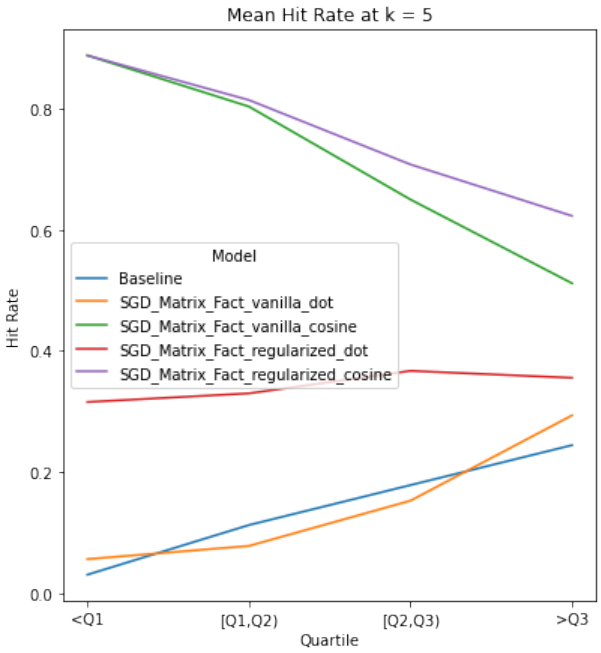
```
user_mainstream_dict = calculate_mainstream_users(user_artists)
visualize_category_evaluation(split_dict_into_four_groups(user_mainstream_dict),
 ↪'Precision', k=5)
```

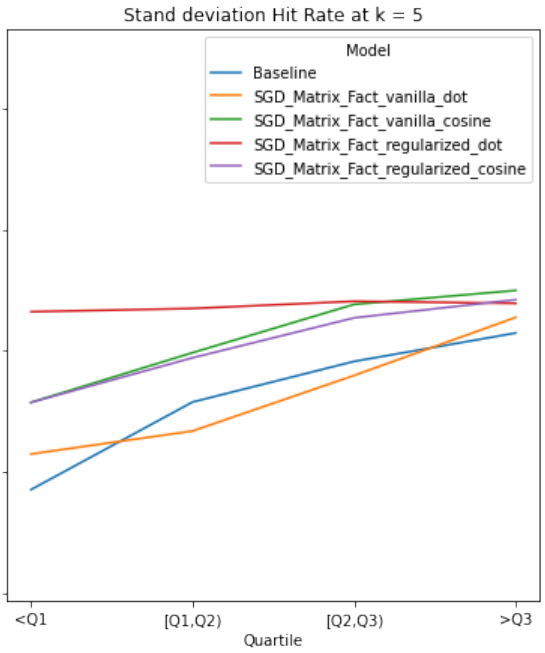



```
visualize_category_evaluation(split_dict_into_four_groups(user_mainstream_dict),
↪'Recall', k=5)
```



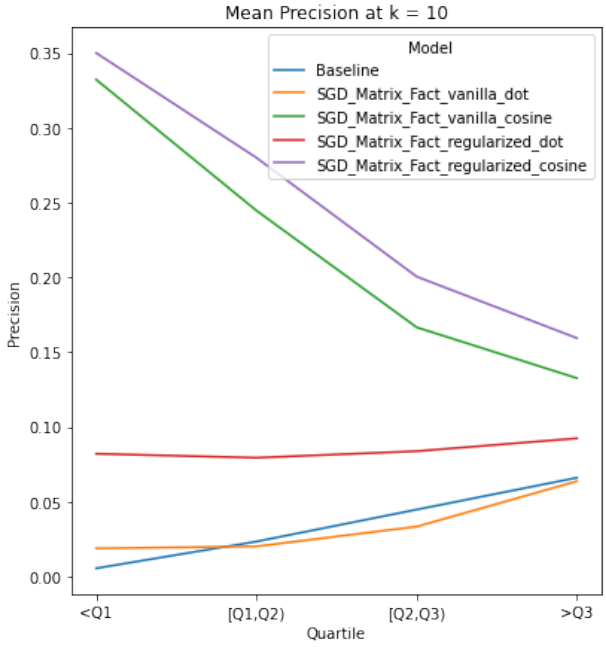

```
visualize_category_evaluation(split_dict_into_four_groups(user_mainstream_dict), 'Hit␣
↪Rate', k=5)
```

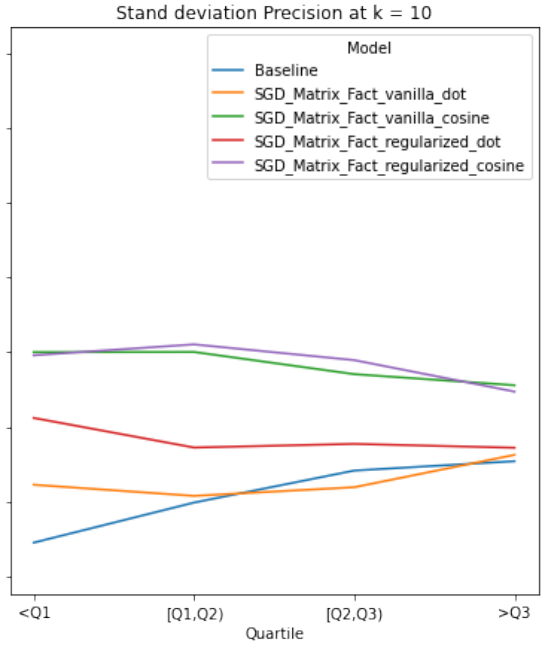




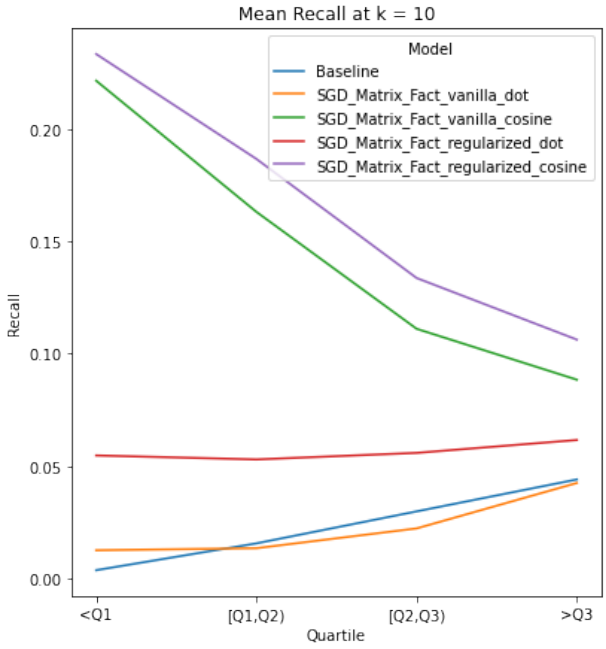
Mainstream K = 10

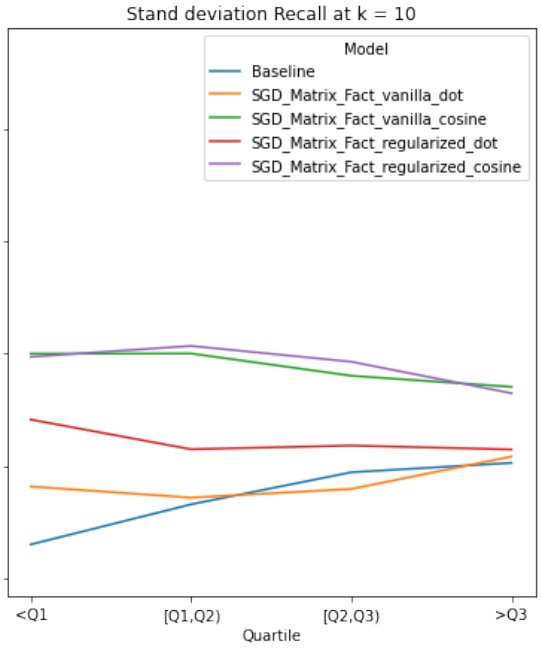
```
visualize_category_evaluation(split_dict_into_four_groups(user_mainstream_dict),
 ↪'Precision', k=10)
```



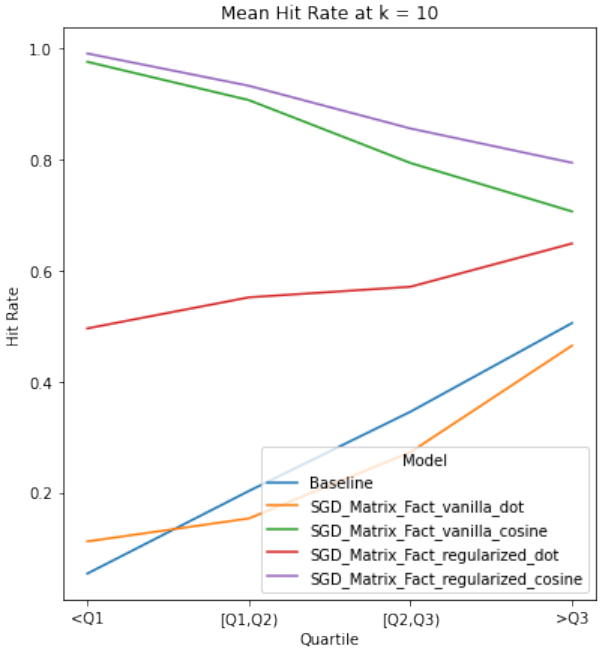


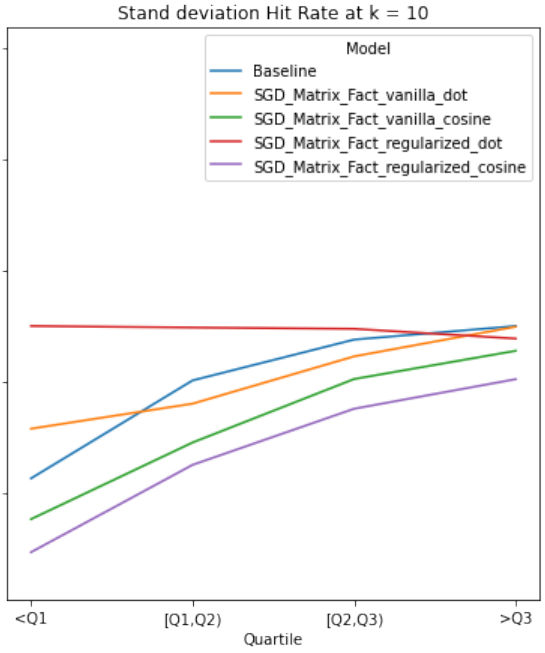

```
visualize_category_evaluation(split_dict_into_four_groups(user_mainstream_dict),
 ↪'Recall', k=10)
```





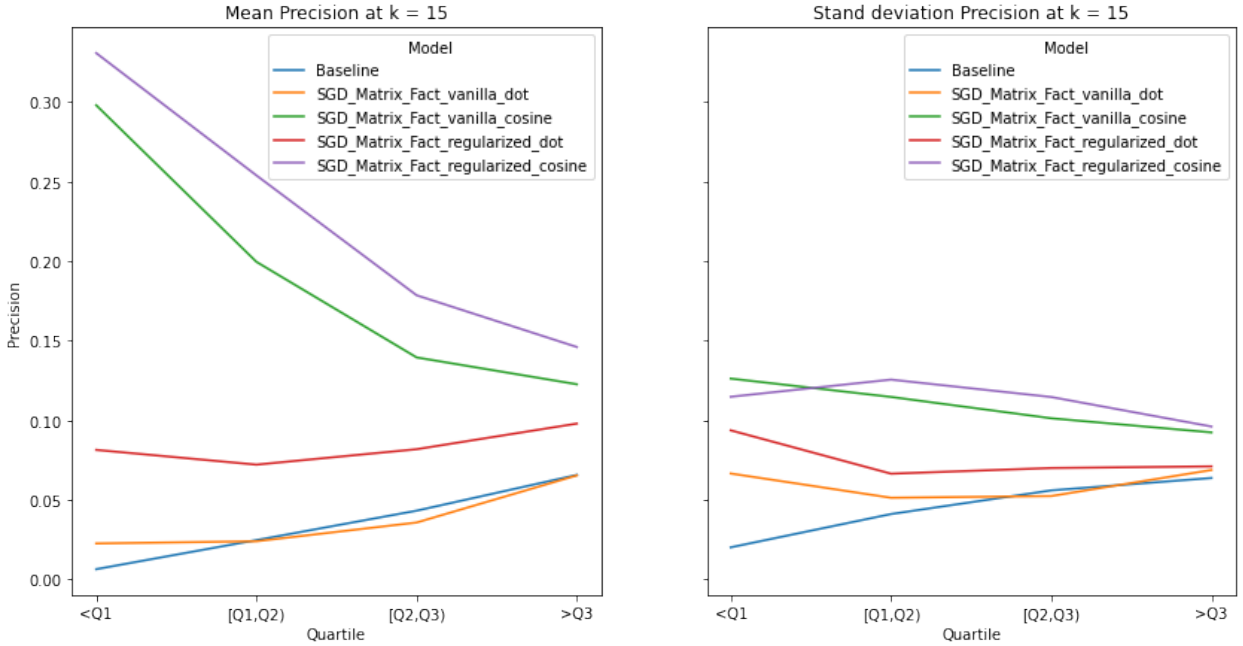

```
visualize_category_evaluation(split_dict_into_four_groups(user_mainstream_dict), 'Hit␣
↪Rate', k=10)
```



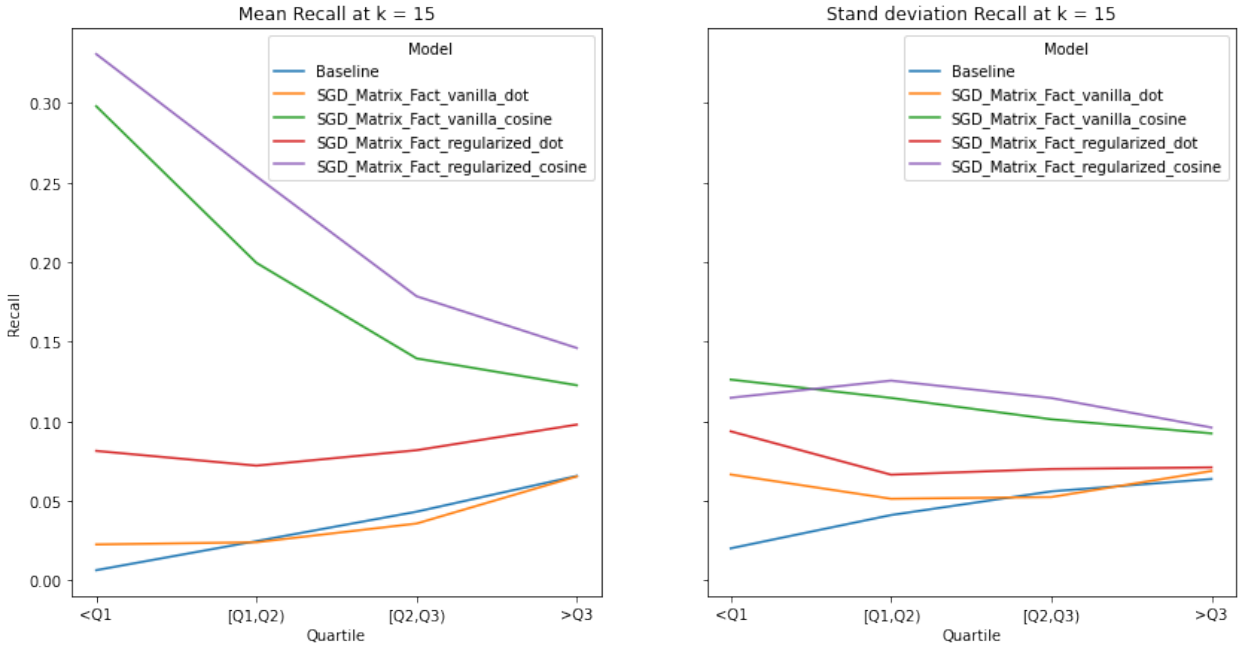


Mainstream K = 15

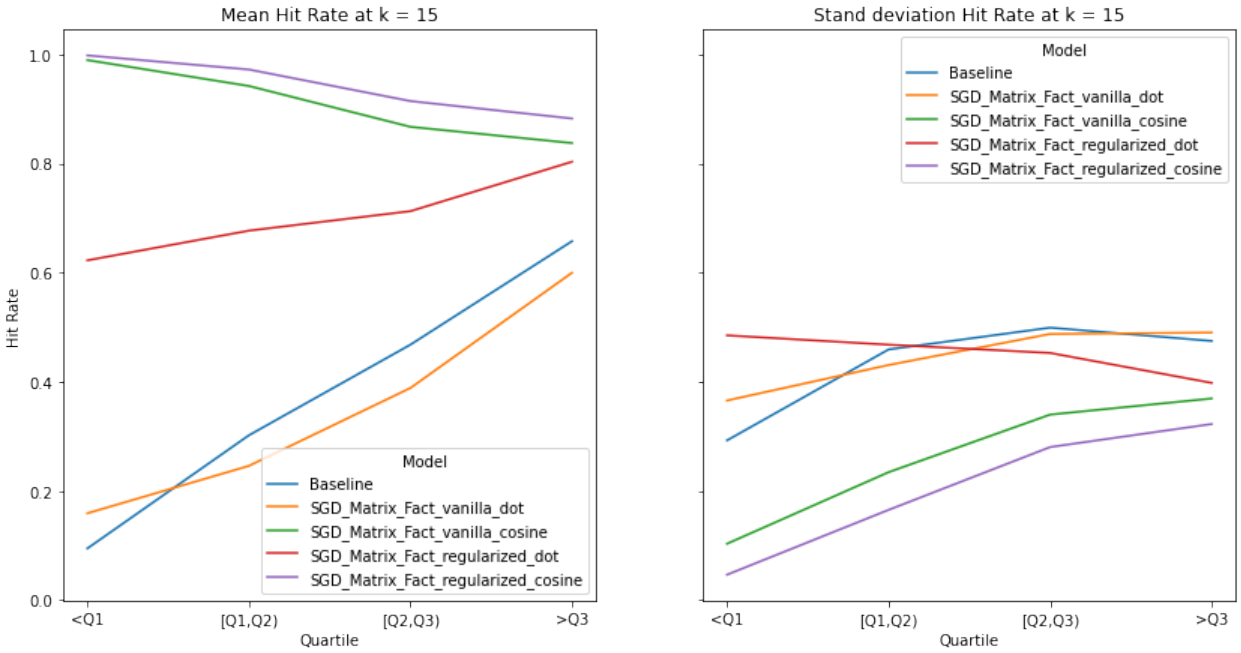
```
visualize_category_evaluation(split_dict_into_four_groups(user_mainstream_dict),
↪'Precision', k=15)
```




```
visualize_category_evaluation(split_dict_into_four_groups(user_mainstream_dict),
↪'Recall', k=15)
```




```
visualize_category_evaluation(split_dict_into_four_groups(user_mainstream_dict), 'Hit␣
↪Rate', k=15)
```



An ideal artist recommender system should not alienate users with niche music habitats. We notice two main threads:

- the performance of the recommender systems that incorporates the **cosine similarity decreases** across much of the examined metrics as user's listening habitats become more mainstream.
- the performance of the recommender systems that incorporates the **dot product increases** across much of the examined metrics as user's listening habitats become more mainstream.

These graphs make intitule sense and confirms our observation from our overall evaluation i.e. as dot product takes into account the norm of the vectors, it is more likely to benefits users that listen to mainstream artists. It must be noted that there is no discernable pattern in the spread of the evaluation metrics across mainstream groups. Although we do notice the same observations as our overall evualtion, ie the tendency of the spread to decrease as k increases.

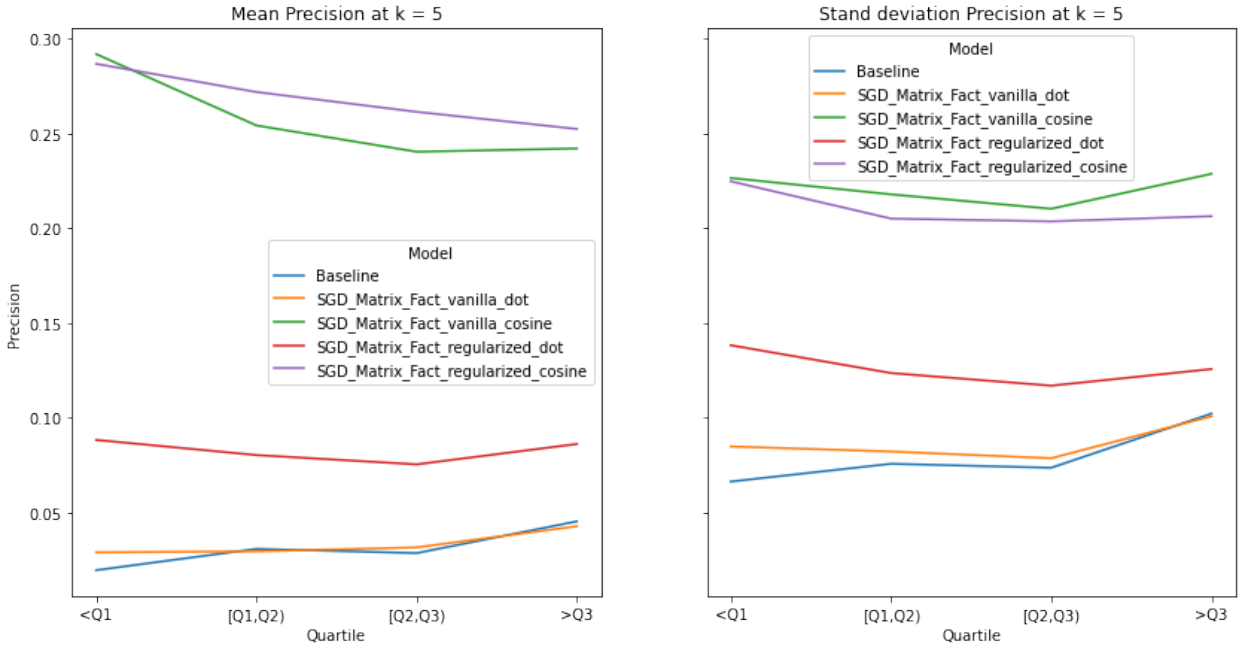
4.4.3 Activity

Finally, we plot the mean and standard deviation of the various evaluation metric values (described above), for each group of varying activity levels. **Note**, the group's (defined on the x axis) collective activity score increases as we move along the axis. We determine activity in terms of the number of plays a user has made on Last.fm, rather than the amount of artists they have listened to. For instance, the group labelled as ">Q3" represents a cluster of user IDs, where all of the user's activity scores are greater than the 75th global percentile.

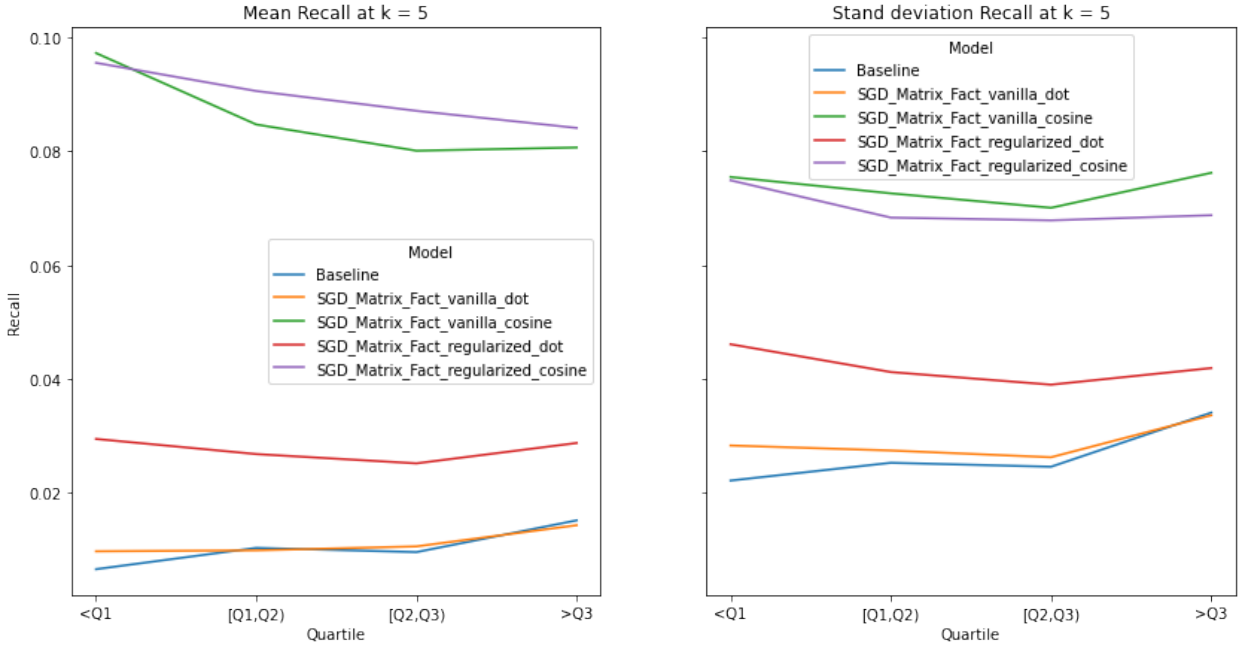
Activity k = 5

```
user_activity_dict = calculate_activity(user_artists)
```

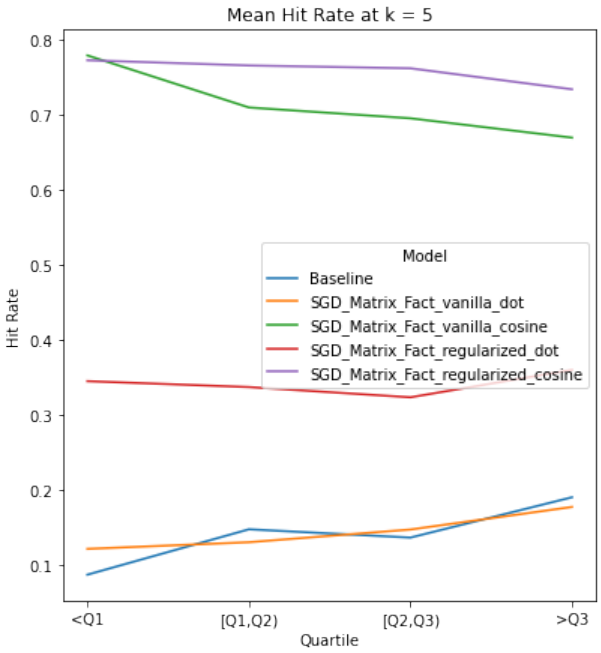
```
visualize_category_evaluation(split_dict_into_four_groups(user_activity_dict),
↪'Precision', k=5)
```

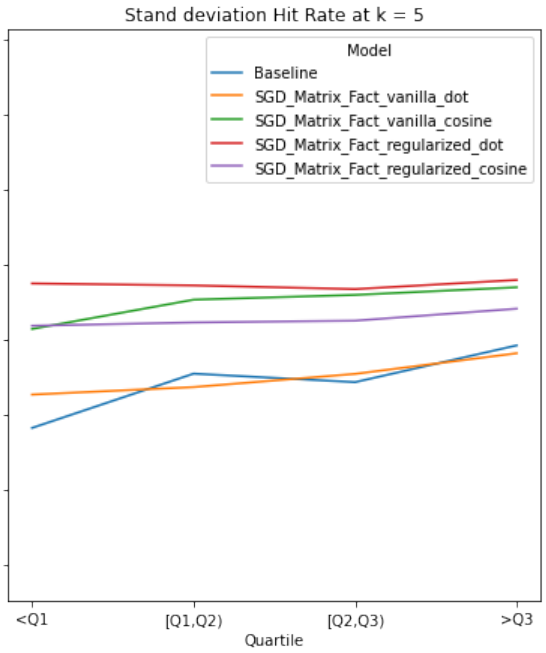



```
visualize_category_evaluation(split_dict_into_four_groups(user_activity_dict), 'Recall
↪', k=5)
```



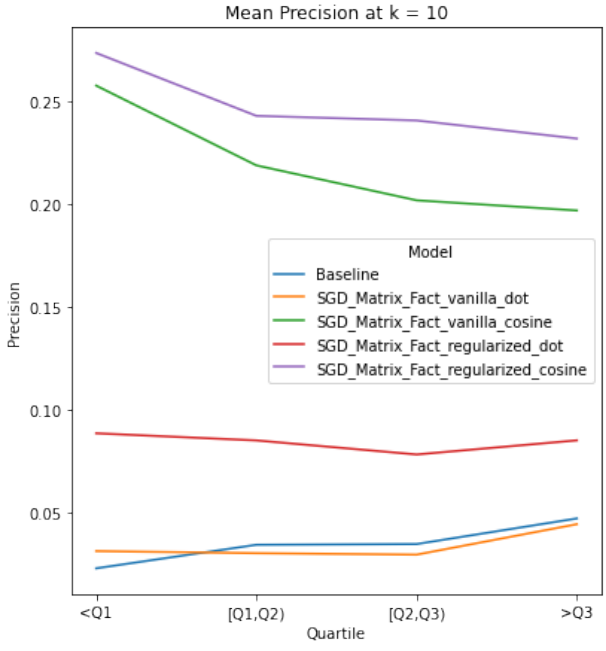

```
visualize_category_evaluation(split_dict_into_four_groups(user_activity_dict), 'Hit␣
↪Rate', k=5)
```

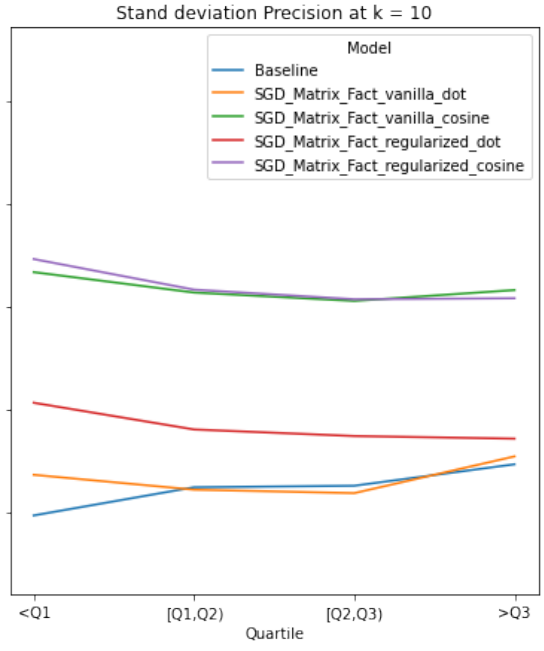




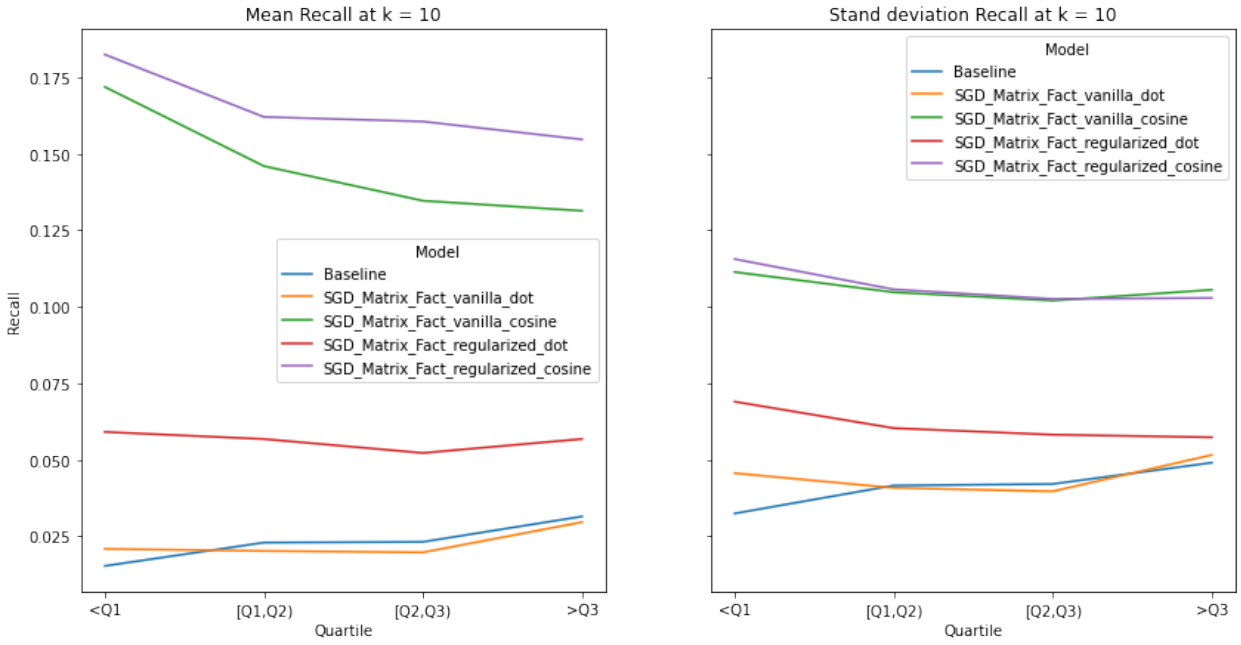
k = 10

```
visualize_category_evaluation(split_dict_into_four_groups(user_activity_dict),
 ↪'Precision', k=10)
```

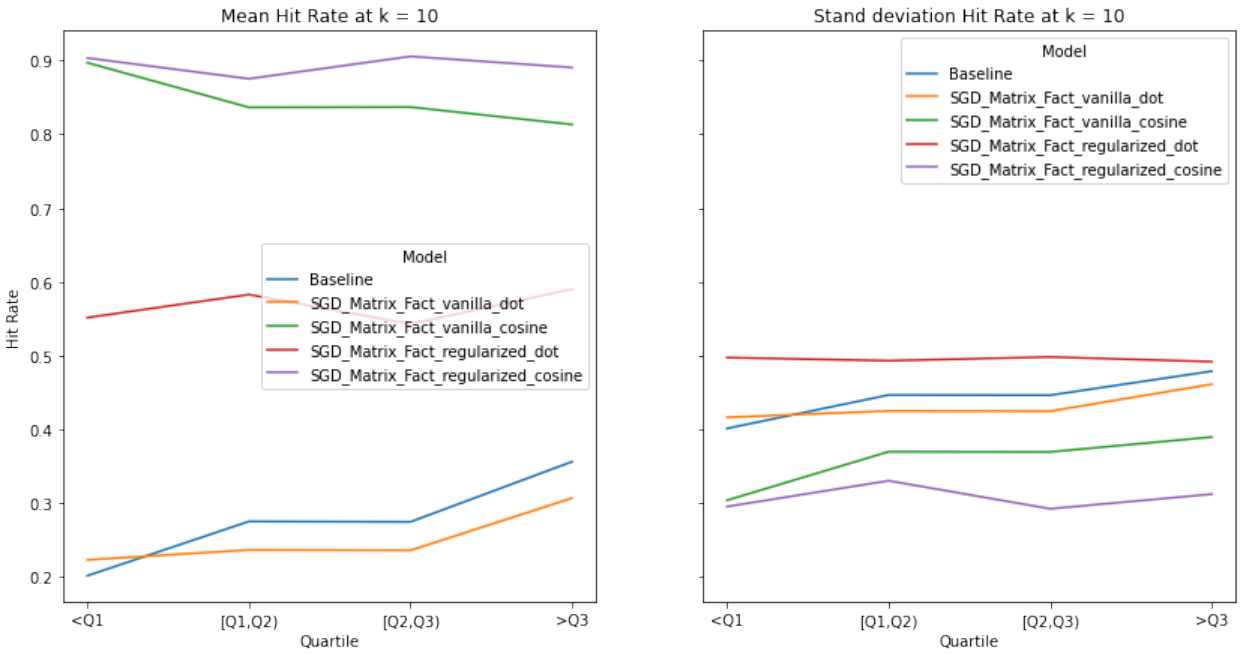





```
visualize_category_evaluation(split_dict_into_four_groups(user_activity_dict), 'Recall
 ↪', k=10)
```

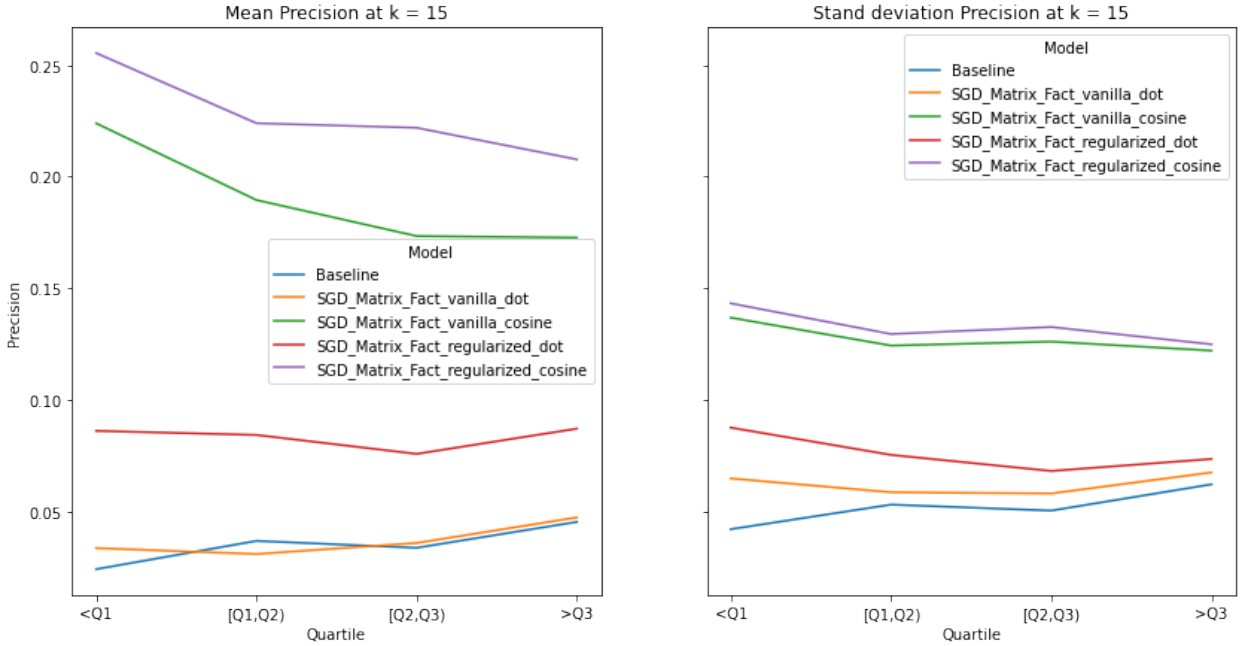



```
visualize_category_evaluation(split_dict_into_four_groups(user_activity_dict), 'Hit␣
↪Rate', k=10)
```

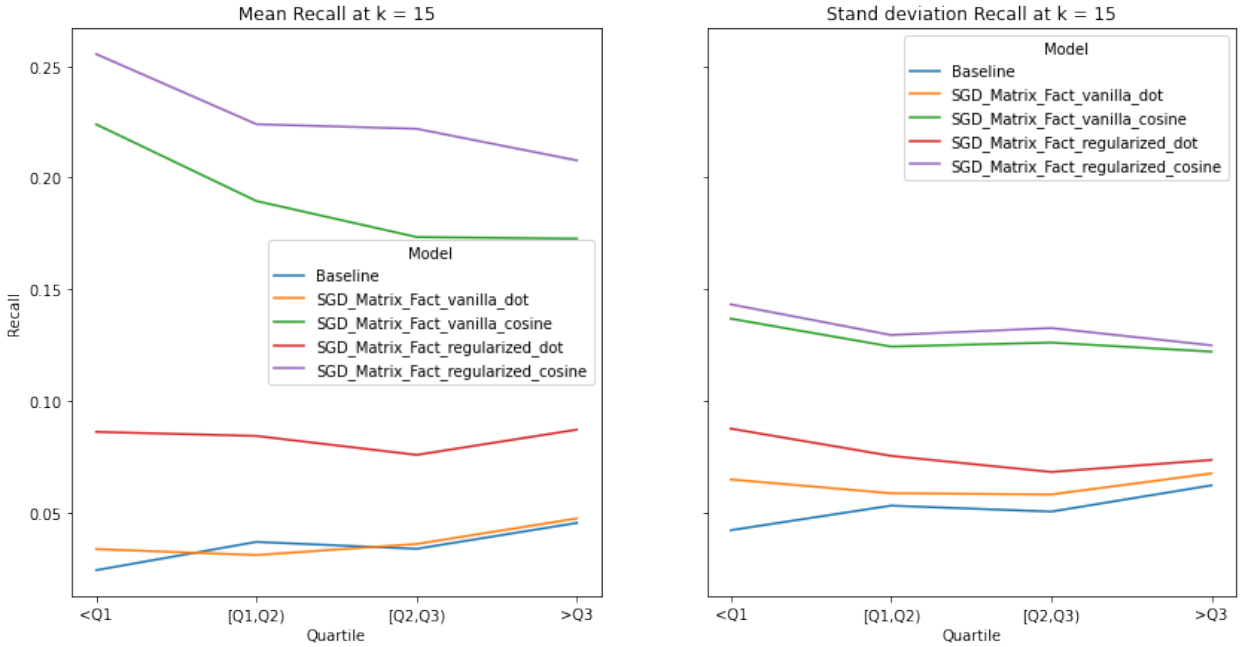


K= 15

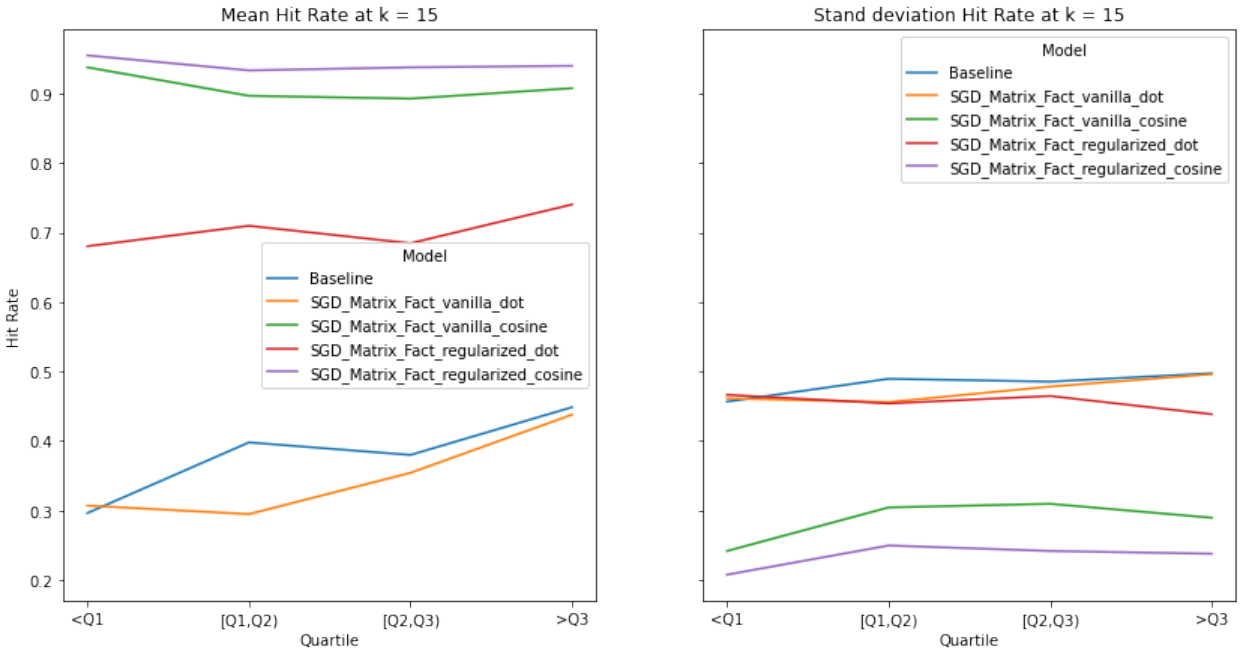
```
visualize_category_evaluation(split_dict_into_four_groups(user_activity_dict),
↪'Precision', k=15)
```




```
visualize_category_evaluation(split_dict_into_four_groups(user_activity_dict), 'Recall
↪', k=15)
```




```
visualize_category_evaluation(split_dict_into_four_groups(user_activity_dict), 'Hit␣
↪Rate', k=15)
```



There is no discernable trend or bias observed in terms our recommendation's systems ability to recommend items to users of different activity level. This makes intuitive sense as we use a binary matrix to encode user's interactions rather than min-max normalizing the listening counts. Perhaps if we did the latter, we would observe increased performance for user's who interact with Last.fm more. Although there is a slight trend that the performance of models with cosine similarity decrease with increased user activity where as those who incorporate the dot product increase, We also notice that regularization increases a model's performance significantly, indicating the model generalize well to user's with different listening habitats.

CHAPTER

FIVE

GRAPH ANALYSIS OF LAST.FM DATA.

Graph analysis is performed in an attempt to better understand the listening habitats of the Last.FM users. Three undirected graphs $G(N, E)$ are constructed that consists of a finite set of nodes N and a finite set of edges E. With each edge E of G, there will be an associated real number $w(E)$, called its weight.

Graph_1 takes advantage of the present $friend \longleftrightarrow friend$ relationships detailed in the data. As this is a friendship relationship rather than a follow relationships, the graph is undirected. In this case, $W(E) = 1$

Currently, there is $user \longleftrightarrow artist$ relationships present in the data. In order to analyse this occurrence, we project this bipartite area of interest into a monopartite sub-graph. *Graph_2* contains $user \longleftrightarrow user$ edges where $w(E) = CA$

where CA is the number of common artists between users.

As is evident so far in this investigation, the are a large number of tags (11946) and these user-submitted semantic categories often contain overlap or are junk. We construct *Graph_3* in attempt to find "super tags" i.e. novel tags that effectively covers numerous sub-tags in an attempt to remove redundancy. We believe this may also improve Last.fm's recommendation system if these novel tags were part of the system. *Graph_3* contains $tag \longleftrightarrow tag$ relationships where $w(E)$= number of common artists. If two tags have no overlapping artists, there exists no relationship.

5.1 Quick Links

- *Importing libraries and data files*
- *Construct Graphs*
- *Degree centrality*
- *Local clustering coefficient*
- *Betweeness Centrality*
- *Louvian Modularity*
- *Label propagation (Finding Super Tags)*

In this investigation, all the graph algorithms described above besides Label propagation are performed on *Graph_1* and *Graph_2*. We will only use Label propagation on *Graph_3*

5.2 Importing libraries and Data files

```
import networkx as nx
from collections import defaultdict
import os
import itertools
import seaborn as sns
import collections
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import ast
! pip install python-louvain
import community as cp
import community as community_louvain
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_66672/1850670818.py in <module>
----> 1 import networkx as nx
      2 from collections import defaultdict
      3 import os
      4 import itertools
      5 import seaborn as sns

ModuleNotFoundError: No module named 'networkx'
```

```
#from google.colab import drive
#drive.mount('/content/drive/')
#os.chdir("/content/drive/My Drive/DCU/fouth_year/advanced_machine_learning/
 ↪assignment-3/music-recommodation-system")
```

```
user_artists = pd.read_csv('data/user_artists.dat', sep='\t')
user_friendships =  pd.read_csv('data/unqiue_friendships.csv')
tags = pd.read_csv(open('data/tags.dat', errors='replace'), sep='\t')
user_taggedartists = pd.read_csv(r'data/user_taggedartists-timestamps.dat', sep='\t')
user_taggedartists = pd.merge(user_taggedartists, tags, on=['tagID'])
tags = pd.read_csv(open('data/tags.dat', errors='replace'), sep='\t')
```

5.3 Constructing the graphs

In order to construct the graphs, we must further construct a list of vertices and a list of tuples detailing edges in a particular graph. We use the *Graph* object as we want an undirected graph. Note, calculating the edges and vertices for *Graph_2* and *Graph_3* wil take several hours. Jupyter book raises a run time exception in this scenario. Therefore, we decide to pre-compute the required edges and vertices and save them into csv files. The code required to construct these graphs is provided for verification purposes.

```
graph_1 = nx.Graph()
graph_2 = nx.Graph()
graph_3 = nx.Graph()
```

```
# GRAPH 1
```

(continues on next page)

(continued from previous page)

```
# graph_1_vertcies = list(set(user_friendships['friend1_id'].values).union(set(user_
↪friendships['friend2_id'].values)))
# graph_1_edges = list()
# for index, row in user_friendships.iterrows():
#   graph_1_edges.append((row['friend1_id'], row['friend2_id']))

# GRAPH 2
# pairs_users = list(itertools.combinations(user_artists['userID'].unique(), 2))
# graph_2_vertices = set()
# graph_2_edges = []
# for user_1,user_2 in pairs_users:
#   user_1_artists = set(user_artists[user_artists['userID'] == user_1]['artistID'].
↪values)
#   user_2_artists = set(user_artists[user_artists['userID'] == user_2]['artistID'].
↪values)
#   common_artist_count = len(user_1_artists.intersection(user_2_artists))
#   if common_artist_count > 0:
#     graph_2_vertices.add(user_1)
#     graph_2_vertices.add(user_2)
#     graph_2_edges.append((user_1, user_2, {'weight': common_artist_count}))

# GRAPH 3
# pair_tags = list(itertools.combinations(user_taggedartists['tagValue'].unique(), 2))
# graph_3_vertices = set()
# graph_3_edges = []
# for tag_1,tag_2 in pair_tags:
#   tag_1_artists = set(user_taggedartists[user_taggedartists['tagValue'] == tag_1][
↪'artistID'].values)
#   tag_2_artists = set(user_taggedartists[user_taggedartists['ustagValueerID'] ==␣
↪tag_1]['artistID'].values)
#   common_artist_count = len(tag_1_artists.intersection(tag_2_artists))
#   if common_artist_count > 0:
#     graph_3_vertices.add(tag_1)
#     graph_3_vertices.add(tag_2)
#     graph_3_edges.append((tag_1, tag_2, {'weight': common_artist_count}))
```

```
def convert_df_list_tuples(df, weight=True):
    """Converts edge df columns to a list of tuples"""
    edges = list()
    if not weight:
        for index, row in df.iterrows():
            edges.append((row[0], row[1], {'weight':1}))
    else:
        for index, row in df.iterrows():
            edges.append((row[0], row[1], ast.literal_eval(row[2])))
    return edges
```

```
graph_1_vertcies =  pd.read_csv('data/graph/graph_1_vertices.csv')['friends'].values.
↪tolist()
graph_1_edges =  pd.read_csv('data/graph/graph_1_edges.csv')
graph_1_edges = convert_df_list_tuples(graph_1_edges, False)

graph_2_vertices =  pd.read_csv('data/graph/graph_2_vertices.csv')['users'].values.
↪tolist()
graph_2_edges =  pd.read_csv('data/graph/graph_2_edges.csv')
graph_2_edges = convert_df_list_tuples(graph_2_edges, True)
```

(continues on next page)

(continued from previous page)

```
graph_3_vertices =  pd.read_csv('data/graph/graph_3_vertices.csv')['tags'].values.
↪tolist()
graph_3_edges =  pd.read_csv('data/graph/graph_3_edges.csv')
graph_3_edges = convert_df_list_tuples(graph_3_edges, True)
```

```
graph_1.add_nodes_from(graph_1_vertcies)
graph_1.add_edges_from(graph_1_edges)

graph_2.add_nodes_from(graph_2_vertices)
graph_2.add_edges_from(graph_2_edges)

graph_3.add_nodes_from(graph_3_vertices)
graph_3.add_edges_from(graph_3_edges)
```

```
## Validation checks of the graphs.
assert len(graph_1_vertcies) == len(graph_1.nodes())
assert len(graph_1_edges) == len(graph_1.edges())
assert graph_1.get_edge_data(2, 4) == None

assert len(graph_2_vertices) == len(graph_2.nodes())
assert len(graph_2_edges) == len(graph_2.edges())
assert graph_2.get_edge_data(2, 4) == {'weight': 7}


assert len(graph_3_edges) == len(graph_3.edges())
assert len(graph_3_vertices) == len(graph_3.nodes())
assert graph_3.get_edge_data('chillout', 'downtempo') == {'weight': 263}
```

5.4 Degree centrality

Our initial graphs analysis is very simple. We compute the amount of degrees each nodes has in *Graph_1* and *Graph_2*. The degree is the number of links incident upon a node. The nodes are then sorted by degree descending to find the users that have the most number of friends in *Graph_1* and the users who listen to the most artists in *Graph_2*.

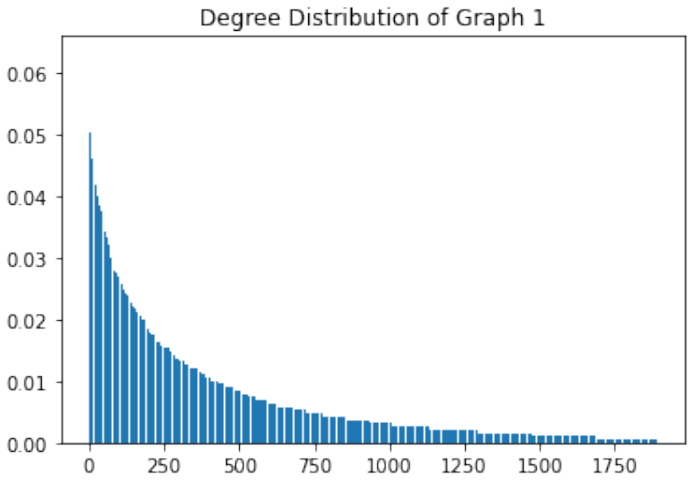
```
def get_five(dictionary, algo_string):
    count = 0
    user_id  = []
    values = []
    for user, value in dictionary.items():
        user_id.append(user)
        values.append(value)
        if count == 4:
            break
        count+=1
    return pd.DataFrame(data={"userID":user_id,algo_string:values})
```

Graph_1: The top five users with the most number of friends

```
get_five(dict(sorted(nx.algorithms.centrality.degree_centrality(graph_1).items(),␣
↪key=lambda item: item[1], reverse=True)), "Degree Centrality")
```

```
   userID  Degree Centrality
0    1543           0.062930
1    1281           0.058170
2     831           0.056055
3     179           0.051296
4    1503           0.050238
```

```
dc = nx.degree_centrality(graph_1)
degrees = dc.values()
degrees = sorted(degrees, reverse=True)
plt.bar(range(len(degrees)), degrees, align='center')
plt.title('Degree Distribution of Graph 1')
plt.show()
```



The above figure plots the degree centrality distribution of *graph_1*. As expected, the distribution histogram of degree centrality has a long tail, which means their are few people have a lot of friends while most of people have a relatively small number of friends.

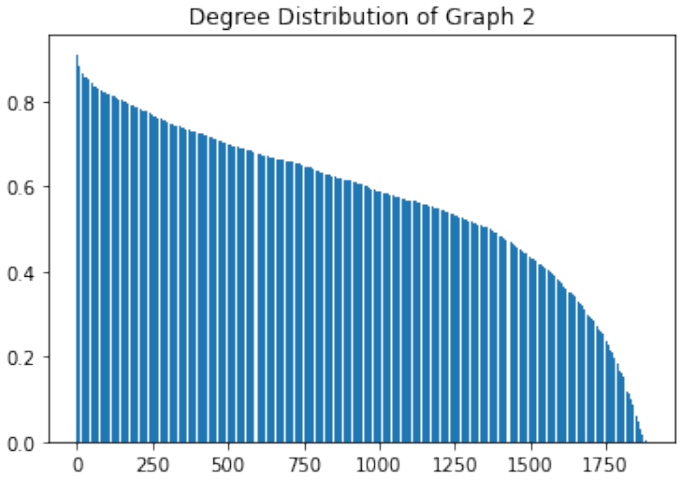
Graph_2: The top five users who have listened to the most amount of artists.

```
get_five(dict(sorted(nx.algorithms.centrality.degree_centrality(graph_2).items(),␣
↪key=lambda item: item[1], reverse=True)), "Degree Centrality")
```

```
   userID  Degree Centrality
0    1368           0.909766
1     458           0.904989
2    1469           0.898620
3     864           0.890127
4    1114           0.885350
```

```
dc = nx.degree_centrality(graph_2)
degrees = dc.values()
degrees = sorted(degrees, reverse=True)
plt.bar(range(len(degrees)), degrees, align='center')
plt.title('Degree Distribution of Graph 2')
plt.show()
```

Again, the above plot describes the degree centrality distribution of *graph_2*. The distribution is only slightly right skewed. This indicates that the userbase of Last.fm generally listens to a limited number of artists with a significant number also



listening to a wide range of artists.

5.5 Clustering coefficient and triangular count

Triangular count refers to the number of triangles or 3-clique for each node in a graph. A triangle is a set of three nodes where each node has a relationship to the other two. Triangular count is particularly useful in social media analysis where it is used to detect communities and graph cohesiveness. It also used as part is used as part of the local clustering coefficient algorithm (C_n) of a node n. . This algorithm describes the likelihood ([0,1]) that n's neighbour's are also connected. **Note**, this algorithm is computed for each node. For unweighted graphs like *graph_1*, $C_n = \frac{2T_n}{d_n(d_n)-1}$ where T_n is the number of triangles that node n is part of, and d_n is the degree of node n.

For weighted graphs like *Graph_2*, there are several ways to define clustering (Saramäki et al, 2007). Networkx's uses the geometric average of the subgraph edge weights, that is $C_n = \frac{1}{d_n(d_n)-1} \sum_{vw} (\hat{w}_{uv}\hat{w}_{uw}\hat{w}_{vw})^{\frac{1}{3}}$ The edge weights $\hat{w}_{uv}$ are normalized by dividing by the maximum weight in the network. $\hat{w}_{uv} = \frac{w_{uv}}{max(w)}$ Note, if $d_n < 2$, it is assigned a value of 0.

To measure the global cohesiveness of a graph, we can calculate the average local clustering coefficient from all nodes. Due to the quite large graphs, we will just display the output and comment out the code. Otherwise, we run into timeout errors when trying to build the jupyter book.

```
#networkx.algorithms.cluster.average_clustering(graph_1)
```

0.18654479547922215

```
#networkx.algorithms.cluster.average_clustering(graph_2, weight='weight')
```

0.06315564340785443

5.6 Betweeness centrality

Betweenness centrality detects the amount of influence a node has over the flow of information in a graph i.e. detecting 'bridge points' between one part of a graph to another. For *graph_1*, we interrupt these bridging nodes as famous Djs or prominent influencers within the graph. For *graph_2*, we interpret these bridging nodes as people with varied music tastes, joining users with almost mono music interest.

Graph_1: The most influential users of Last.FM

```
get_five(dict(sorted(nx.algorithms.centrality.betweenness_centrality(graph_1).items(),
 ↪ key=lambda item: item[1], reverse=True)), "Degree Centrality")
```

	userID	Degree Centrality
0	1543	0.066527
1	1281	0.054549
2	831	0.048691
3	1258	0.043025
4	78	0.029337

Graph_2: Users who listen to the most diverse set of artists

```
get_five(dict(sorted(nx.algorithms.centrality.betweenness_centrality(graph_2).items(),
 ↪ key=lambda item: item[1], reverse=True)), "Degree Centrality")
```

	userID	Degree Centrality
0	1989	0.001067
1	240	0.000911
2	1914	0.000769
3	458	0.000761
4	1368	0.000743

5.7 Louvian modularity:

Louvian modularity is not part of the official networkx library. Therefore, we decide to use the `python-louvian` package. Louvain modularity is used to find communities, hierarchies and/or evaluate different grouping thresholds. Louvain modularity compares relationship densities in given clusters to densities between clusters i.e. we want a lot of connections in the set and few connections outside the set. We apply this algorithm to *Graph_1* and *Graph_2* to find:

- (a) distinct social communities
- (b) distinct communities of similar music tastes

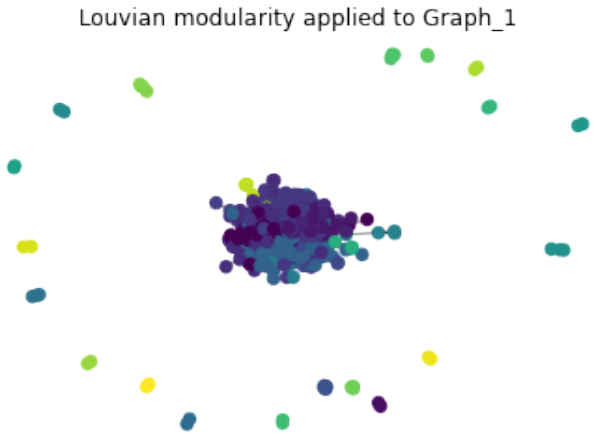
We relate (b) to the tags, in an effort to see if there is a correlation between the implicit data of artist listenership vs the explicit data of user-submitted tags.

```
partition = community_louvain.best_partition(graph_1)
# draw the graph_1
pos = nx.spring_layout(graph_1)

cmap = cm.get_cmap('viridis', max(partition.values()) + 1)
nx.draw_networkx_nodes(graph_1, pos, partition.keys(), node_size=40,
                       cmap=cmap, node_color=list(partition.values()))
nx.draw_networkx_edges(graph_1, pos, alpha=0.5)
plt.title('Louvian modularity applied to Graph_1')
```

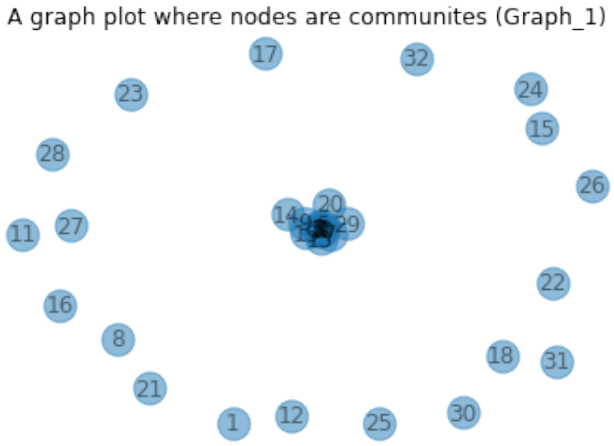
(continues on next page)

(continued from previous page)

```
plt.axis('off')
plt.show()
```




```
IG = community_louvain.induced_graph(partition, graph_1)
pos = nx.spring_layout(IG)
plt.title('A graph plot where nodes are communites (Graph_1)')
nx.draw_networkx(IG, pos, alpha=.5)
plt.axis('off')
plt.show()
```




```
print(f'Louvian modularity identifies { max(partition.values()) + 1} communities in␣
↪the social network graph')
```

```
Louvian modularity identifies 33 communities in the social network graph
```

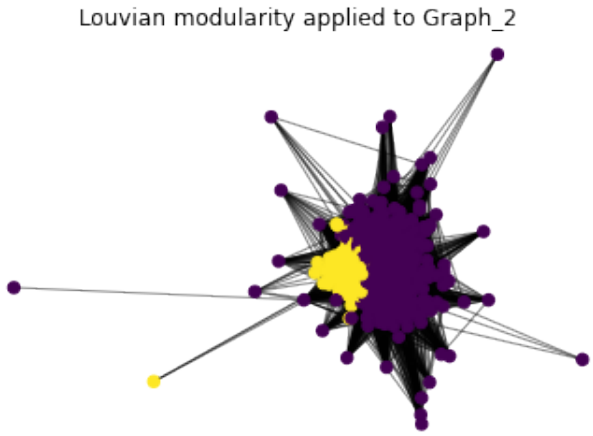
```
print('Modularity score for the partition:', community_louvain.modularity(partition,␣
↪graph_1))
```

```
Modularity score for the partition: 0.45093205542255865
```

The modularity score is a measure of the structure of the graph. Graphs with high modularity scores have many connec-

tions within a community but only few pointing outwards to other communities where as the opposite is true for the inverse result. Modularity score lies between [-0.5, 1), so the final modularity score for the partition is pretty high, indicating that these number of communities is reasonably accurate.

```
partition = community_louvain.best_partition(graph_2, weight='weight')
# draw the graph_2
pos = nx.spring_layout(graph_2)
cmap = cm.get_cmap('viridis', max(partition.values()) + 1)
nx.draw_networkx_nodes(graph_2, pos, partition.keys(), node_size=40,
                       cmap=cmap, node_color=list(partition.values()))
plt.title('Louvian modularity applied to Graph_2')
nx.draw_networkx_edges(graph_2, pos, alpha=0.5)
plt.axis('off')
plt.show()
```




```
print(f'Louvian modularity identifies { max(partition.values()) + 1} communities in␣
↪graph 2')
```

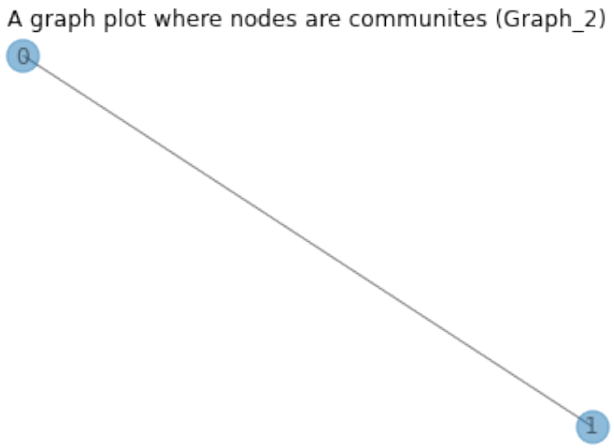
```
Louvian modularity identifies 2 communities in  graph 2
```

```
print('Modularity score for the partition :', community_louvain.modularity(parts,␣
↪graph_2))
```

```
Modularity score for the partition : 0.28234715583451236
```

```
IG = community_louvain.induced_graph(partition, graph_2, weight='weight')
plt.title('A graph plot where nodes are communites (Graph_2)')
pos = nx.spring_layout(IG)
nx.draw_networkx(IG, pos, alpha=.5)
plt.axis('off')
plt.show()
```

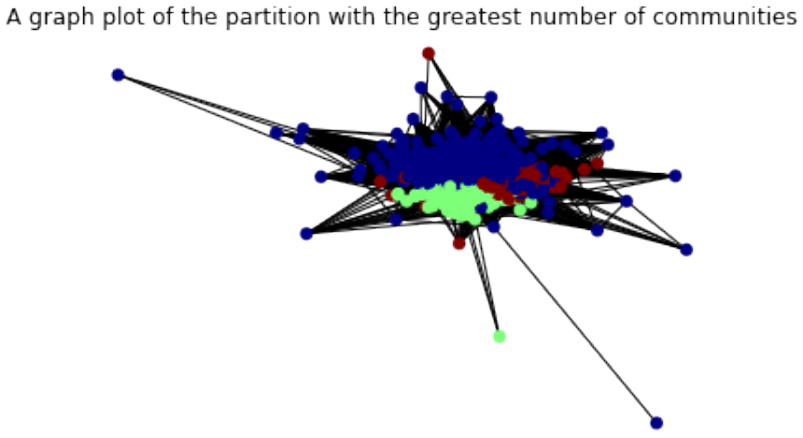
These form during the algorithms second pass where several of the possible intuitive communities are merged together. Louvian modulairty has the power to reveal hierarchy of communities at different scales. The function `generate_dendrogram` gives us a dictionary mapping from level of hierarchy to the partition. Finally, `partition_at_level` takes the result of `generate_dendrogram` and returns a dictionary of the partition, where keys are nodes and values are the index of partition the node belongs to. We use to find the level with the most number of communities.




```
dendrogram = community_louvain.generate_dendrogram(graph_2, weight='weight')
parts = community_louvain.partition_at_level(dendrogram, len(dendrogram) - 1) # get␣
 ↪the largest possible communities
values = [parts.get(node) for node in graph_2.nodes()]
spring_pos = nx.spring_layout(graph_2)
print('Community Node Distribution(Hierarchical Clustering):', collections.
 ↪Counter(values))
```

```
Community Node Distribution(Hierarchical Clustering): Counter({0: 1097, 1: 589, 2:␣
 ↪199})
```

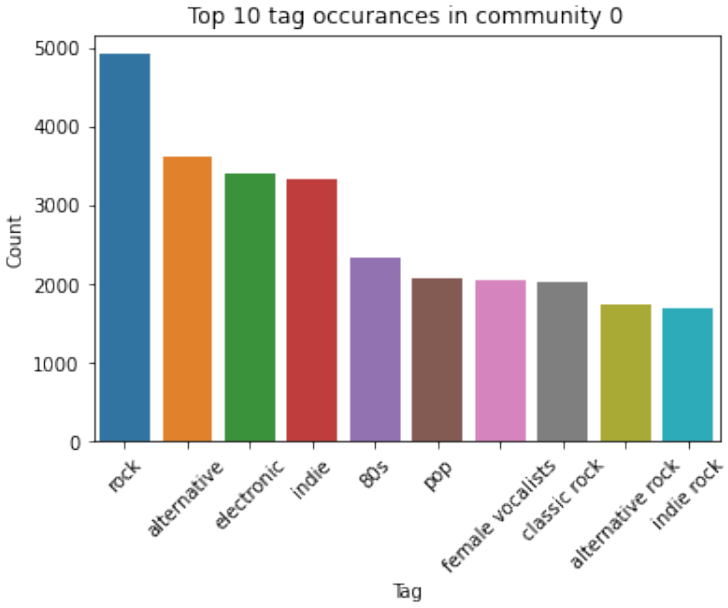
```
plt.title('A graph plot of the partition with the greatest number of communities')
plt.axis('off')
nx.draw_networkx(graph_2, pos = spring_pos, cmap = plt.get_cmap("jet"), node_color =␣
 ↪values, node_size = 35, with_labels = False)
```

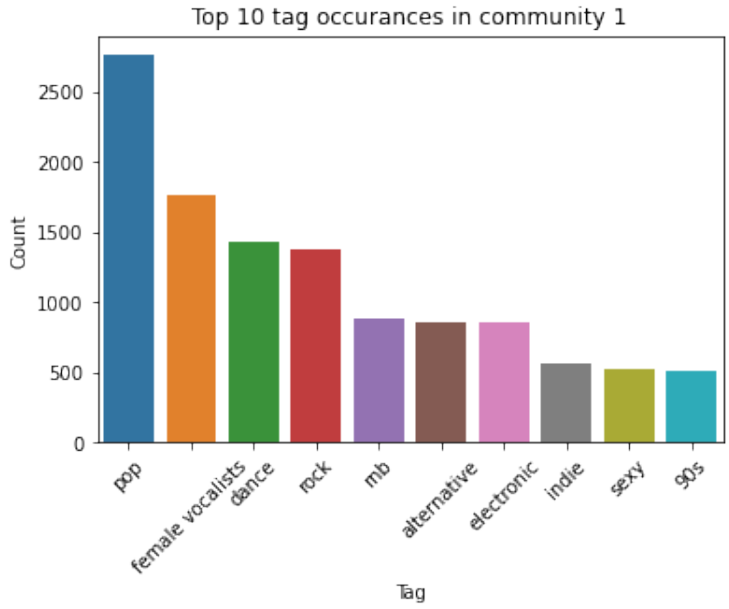


Unfortunately, only a marginal number of additional communities were revealed when we reviewed the different levels of partitions. Finally, the top ten tags by occurrence are plotted in these communities.

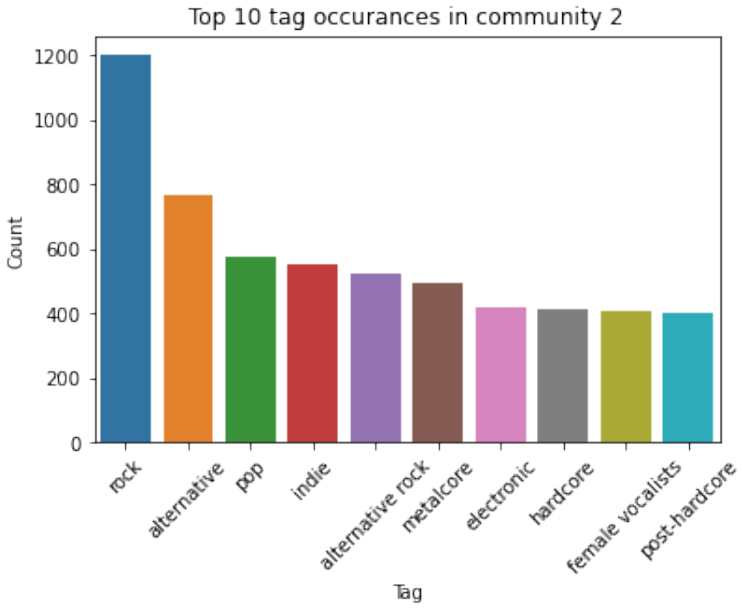
```
community_user_id_dict =  defaultdict(list)
for node_id, community in parts.items():
    community_user_id_dict[community].append(node_id)
```

```
for community_number in community_user_id_dict:
    tag_list = user_taggedartists[user_taggedartists.userID.isin(community_user_id_
↪dict[community_number])]['tagValue'].values.tolist()
    tag_occurances =  collections.Counter(tag_list).most_common(10)
    tag_df = pd.DataFrame.from_dict(tag_occurances)
    tag_df.columns =['Tag','Count']
    sns.barplot(x="Tag", y="Count", data=tag_df)
    plt.title(f'Top 10 tag occurances in community {community_number}')
    plt.xticks(rotation=45)
    plt.show()
```





We observe some commonality in these two communities. For instance, rock, electronic, pop, alternative were present in the top 10 assigned tags in all three communities. Although these are also the top four global tags, so we would expect



this given the small number of communities identified.

Reminder: The weight attribute in Graph 2 represents the number of common artists in user-user relationship.

Besides this, we observe a clear distinction in music tastes among the three clusters. Cluster 0 has a more of tendency to listen to rock and sub genres of rock where as community 1 listens to 90s and dance music. Community 2 seems to follow the mainstream, with little defining music taste.

5.8 Label Propagation - finding super tags

Similar to Louvian Modulaity, Label Propagation is used to find communities within a graph. After initializing each node with a unique label, the algorithm repeatedly sets the label of a node to be the label that appears most frequently among that nodes neighbors. The algorithm halts when each node has the label that appears most frequently among its neighbors. From this notation of 'projected labeling', spun out the idea of creating "super tags" in an effort to reduce redundancy. We envision that these novel tags could be incorporated into Last.fMs retrieval or recommendation systems. This algorithm is applied to *graph_3*.

```
c = list(nx.algorithms.community.label_propagation.label_propagation_
 ↪communities(graph_3))
clusters = ([list(x) for x in c])
for cluster in clusters:
    if len(cluster) < 8:
        print(cluster)
```

```
['jaebeom', 'park jaebum', 'park jaebeom', 'jaebum', 'jaywalkers']
['kwasa', 'angola', 'semba']
['underground hip hp', 'luso hip hop', 'hip hop under', 'raptuga', 'hip hop angolano']
```

When these clusters were examined, we found some intuitive examples. For example, the tags 'underground hip hop','luso hip hop', 'hip hop under', 'raptuga', and 'hip hop angolano' being clustered together. They are all sub-groups of hip-hop. The tags 'kwasa', 'angola', 'semba' are all types of traditional music, originating from central and south-western Africa.

Finally, the first clustering group's tags all relate to the Korean-American rapper Jay Park. This example demonstrates our overall goal of removing redundancy.

CHAPTER

SIX

CONCLUSION

In this report, we investigate numerous recommendation algorithms in an effort to supply worthwhile recommendations to the user base of Last.fm. Through our initial data exploration of the Last.fm data, we noticed inconstancy's within. We flatten the original relational data model present, to enable the quick development of our recommendation systems. We examine three main recommendation algorithms , a stochastic gradient descent matrix factorization algorithm, a regularized stochastic gradient descent matrix factorization algorithm, and a softmax model. In addition, we employ two different similarity measures, cosine similarity and the dot product to provide more nuance for the discussion of results. We evaluate the aforementioned algorithms at the point of development and through model comparison at the end, utilizing metrics typically used to evaluate information retrieval systems. Finally, we conclude this report by employing popular graph algorithms to understand the listening patterns of the Last.fm users.

Our most significant results are the following:

1. Exploratory data analysis revealed that Alternative, ELectronic, Rock and Pop were the most popular genres on the Last.fm website, both in terms of plays and amount of users. Although, the number of user's a genre has garnered does not necessarily correlate with listening count. For instance, the "Electronic" genre had highest average play count but the lowest amount of individual listeners in 2009. In addition, user activity on the site showed a general increase from 2005 to 2011, the year it was collected.

2. Our best performing model, SGD_Matrix_Fact_regularized_cosine show-cased the attributes of a good recommendation system i.e. relatively high precision for lower values of k. Although the spread of the evaluation metrics was higher for smaller values of K.

3. The addition of the regularization on the SGD Weighted Matrix factorization model proved to be significant. We observed increased performance in recall, precision and hit-rate, which was particularly evident when we drilled down into users according to some defined behaviour (e.g. users with diverse tastes). We believe the vanilla model not only "learned" relevant patterns but also random noise within the test data. Therefore, the regularization of the model prevented this overfilling, which led to a better generalization ability.

4. The choice of similarity measure proved to be the most significant indicator of performance. As mentioned previously, cosine similarity just takes into account the angle where as dot product takes into account the angle **and** magnitude. The choice of similarity measure is highly domain-specific for in-silico applications. For instance, in information retrieval, the dot product will take the document length into account, whereas the cosine similarity will not. For our use case, the dot product is biased towards mainstream artists i.e. it is more likely to recommend artist with a larger following. We suggest this hypothesis as mainstream artists embeddings tend to have larger norms. This observation is particularity pronounced when we examine users with varying predispositions to mainstream listenership. We notice that performance of models that incorporated the cosine similarity degraded as user's listening habitats became more mainstream where as the opposite is true for models that incorporated the dot product. Overall, models that used the cosine similarity performed better than those who did not.

5. There was no distinguishable trends when examining the performance of our models with users of varying activity levels. This is most certainly due to the way we encode user interactions through a binary matrix.

6. The social network of Last.fm does the observe the qualities of a small-world network. We did not observe a small-world phenomenon of users being linked together by a short chain of acquaintances. This is likely due to the fact that Last.fm is not a social network or messaging service.
7. Label propagation was effective in removing redundancy of user-submitted tags. The newly identified communities of tags were intuitive. For instance, 'underground hip hp', 'luso hip hop', 'hip hop under', 'raptuga', and 'hip hop angolano' being clustered together.

6.1 Future work

There are several possibilities to extend this work in future. Currently, we encode user interactions via a binary matrix. As we notice no major trends when cross-referencing our models' performance with user's activity level, we plan to normalize the listening count feature and evaluate accordingly. The WALS algorithm can take advantage of side information when recommending items to users. We would be interested in supplying side features such as our newly created super tags, and the local/betweenness clustering coefficients of users of *Graph_1* and *Graph_2*. After, we plan to examine this model with our pre-existing models. Presently, we just use precision, recall and hit-rate to evaluate our models. However, these metrics do not completely capture the requirements of a recommender system as outlined in our introductory paraph. Other metrics we plan to experiment with include coverage, personalization (1 - similarity between user's lists of recommendations) and intra-list similarity. From the evaluation section, we noticed some models struggled while others succeeded for different types of users. By segmenting the last.dm data into separate but homogeneous user datasets and training different recommender systems on those separate data chunks, we hope to potentially improving our results. Finally, we would be interested in building a hybrid recommendation system. The major weakness of our current collaborative filtering models is that by forgoing the actual characteristics of artists and their songs, the interpretability of recommendations diminishes. Our idea is to integrate both the features of an artist's songs (content-based approach) and user similarity (collaborative filtering)for better music recommendation. The music genome project would serve as the basis for song feature extraction.

CHAPTER

SEVEN

REFERENCES

Beel, J., Langer, S. and Genzmehr, M., 2013. Sponsored vs. Organic (Research Paper) Recommendations and the Impact of Labeling.

Blodgett, S.L., Barocas, S., Daumé III, H. and Wallach, H., 2020. Language (technology) is power: A critical survey of" bias" in nlp. arXiv preprint arXiv:2005.14050.

Buolamwini, J. and Gebru, T., 2018, January. Gender shades: Intersectional accuracy disparities in commercial gender classification. In Conference on fairness, accountability and transparency (pp. 77-91). PMLR.

Garrido-Muñoz, I., Montejo-Ráez, A., Martínez-Santiago, F. and Ureña-López, L.A., 2021. A Survey on Bias in Deep NLP. Applied Sciences, 11(7), p.3184. Hofmann, T., 2004. Latent semantic models for collaborative filtering. ACM Transactions on Information Systems (TOIS), 22(1), pp.89-115.

He, X., Liao, L., Zhang, H., Nie, L., Hu, X. and Chua, T.S., 2017, April. Neural collaborative filtering. In Proceedings of the 26th international conference on world wide web (pp. 173-182).

Hurley, N. and Zhang, M., 2011. Novelty and diversity in top-n recommendation–analysis and evaluation. ACM Transactions on Internet Technology (TOIT), 10(4), pp.1-30.

Marple Jr, S.L. and Carey, W.M., 1989. Digital spectral analysis with applications.

Perez, S., 2021. TechCrunch is part of the Yahoo family of brands. [online] Techcrunch.com. Available at: https://techcrunch.com/2020/06/18/tiktok-explains-how-the-recommendation-system-behind-its-for-you-feed-works/ [Accessed 22 November 2021].

Rendle, S., Freudenthaler, C. and Schmidt-Thieme, L., 2010, April. Factorizing personalized markov chains for next-basket recommendation. In Proceedings of the 19th international conference on World wide web (pp. 811-820).

Saramäki, J., Kivelä, M., Onnela, J.P., Kaski, K. and Kertesz, J., 2007. Generalizations of the clustering coefficient to weighted complex networks. Physical Review E, 75(2), p.027105.

Sembium, V., Rastogi, R., Tekumalla, L. and Saroop, A., 2018, April. Bayesian models for product size recommendations. In Proceedings of the 2018 World Wide Web Conference (pp. 679-687).

Su, X. and Khoshgoftaar, T.M., 2009. A survey of collaborative filtering techniques. Advances in artificial intelligence, 2009.

Tsintzou, V., Pitoura, E. and Tsaparas, P., 2018. Bias disparity in recommendation systems. arXiv preprint arXiv:1811.01461.