Contents

Introduction

Computer-based systems provide us with the opportunity of mining user history to reveal trends and personal preferences that they, and those around them, may not have previously realised [Ekstrand *et al.*, 2011]. A vast amount of research has been conducted on how to best recommend content to users and many methods have been proposed [Balabanović and Shoham, 1997, Guttman *et al.*, 1998, Pazzani, 1999].

In this research, we will be focusing on Collaborative Filtering [Su and Khoshgoftaar, 2009, Thorat *et al.*, 2015], an approach that decides what to recommend to a user based on the preferences of similar users [Elahi *et al.*, 2016, Su and Khoshgoftaar, 2009]. Similarity between users is estimated by comparing user-interactions and ratings with products or services [Elahi *et al.*, 2016, Thorat *et al.*, 2015]. The table below highlights some popular services currently using recommender systems.

Service	Recommendations
Amazon	Retail Products
Facebook (Meta)	Friends
Netflix	Movies / Shows
Spotify	Music
Youtube	Videos
Google News	News

Recommending something to someone else is a natural process that aims to help somebody sift through a large corpus of information efficiently to find only the items of most interest to them. Recommender systems augment that process, and as such have become a vital tool for information-availability in our increasingly digital lives [Su and Khoshgoftaar, 2009].

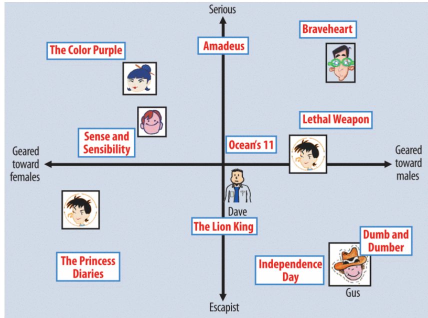
In the following Jupyter Book, we will be using data acquired from the HETREC 2011[Cantador *et al.*, 2011] 2nd International Workshop on Information Heterogeneity and Fusion in Recommender Systems. In particular, we will be focusing on the *artist listening records* dataset published by Last.FM. This dataset consists of 92,800 artists listening records from 1,892 users. We will explore this data in more detail in Preliminary Data Analysis. In the following section, we will discuss in-depth two approaches to Collaborative Filtering, namely; *Matrix Factorisation*, and *Recommendation with Deep Neural Networks*.

Collaborative Filtering

Collaborative Filtering (CF) was a term first coined by the makers of *Tapestry*, the first recommender system. CF aims to solve the problem of recommending m items, by n users where the data is often represented by a $n \times m$ matrix [Laishram *et al.*, 2016]. There are two main approaches to CF, *neighborhood methods*, and *latent factor models*. The neighborhood methods compute the relationships between users, or alternatively items. Latent Factor models aim to characetrize users and items on many abstract factors infered from the rating patterns. The most successful realisations of latent factor models are based on *matrix facotrisation* (MF). [Koren *et al.*, 2009]. In this section, we will look at MF more closely, focusing on the aspects we will incorporate during our research.

Matrix Factorisation (MF)

In its basic form, MF characterises querys (users) and items by vectors which are inferred from rating patterns. High correspondance between query vector and item vector results in a recommendation. The most useful data from an MF model is *explicit* feedback. This constitutes explicit input from users regarding interest in items. We refer to this as *rating* [Koren *et al.*, 2009]. The construction of query and item vectors intends to model user preferences as shown in the figure below.



As stated previously, in MF, we are given the feedback matrix denoted $A \in \mathbb{R}^{m \times n}$, where m is the number of queries and n is the number of items. We will denote our user vector $U \in \mathbb{R}^{m \times d}$, and our item vector $V \in \mathbb{R}^{n \times d}$. We will use the product UV^T as our approximation of A.

Our objective function is then defined as follows: $\min_{U \in \mathbb{R}^{m \times d}, V \in \mathbb{R}^{n \times d}} \sum_{(i,j) \in obs} (A_{ij} - \langle U_i, V_j \rangle)^2$

The two standard approaches to minimizing error in CF are *Alternating Least Squares* (ALS) [Cichocki and Zdunek, 2007], and *Stochastic Gradient Descent* (SGD) [Gemulla *et al.*, 2011]. *Weighted Alternating Least Squares* (WALS) is a modified version of ALS that is suited to MF. In the literature, it is often cited that WALS is the prefered algorithm for recommender systems. This is because WALS is optimised for parallelization, meaning it is scalable to large scale data. As well as this, ALS performs better than SGD on systems centered on implicit data [Koren *et al.*, 2009]. SGD cannot directly scale to very large data [Gemulla *et al.*, 2011]. Taking our own problem into consideration, either approach is suitable. The recognised benefits of WALS are not of interest to us as parallelization is out of the scope for this research. We have no information regarding intrinsic user data such as age, country, etc. Our dataset is also small enough that the performance of SGD is not severly hindered. Therefore, we will adopt SGD in this approach.

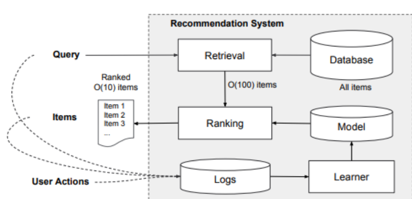
Deep Neural Networks

Advances in deep learning-based recommender systems have gained notable traction over the past years. This goes hand-in-hand with the increased adoption of deep learning frameworks in most ML applications. Deep learning can effectively capture nonlinear and nontrivial user-item relationships through complex data abstraction [Zhang *et al.*, 2019].

To develop our own deep neural network recommender, we will employ Tensorflow V2, with an emphasis on the *Tensorflow Recommedation Systems* (TFRS) package. Tensorflow is developed by Google, the leading experts in recommendation systems. The TFRS package can be used through Keras which is an API built atop Tensorflow. TFRS provides a large volume functions designed spcifically for working with recommender systems.

Google have also authored many publications regarding their philosophy regarding recommender systems for their various applications [Cheng *et al.*, 2016, Covington *et al.*, 2016]. They regularly follow a repeating pattern of developing two seperate 'Towers', a user (query) tower, and an item (candidate) tower. Each of these towers can have varying levels of complexity, and can host deep learning embedings. A combined model of the outputs of these seperate towers is then designed as a feed-forward deep network.

Google also regularly advice a two-tier approach to recommender systems. That is: a retrieval model, and a ranking model. Their typical architeture is displayed in the image below, which has been abstracted from [Cheng *et al.*, 2016]



The TFRS package allows for the easy creation of recommender models by following simple steps. Firstly, one must inheret from the base `tfrs.Model` class, which is packaged with many utility functions. We build our objective function (*denoted as the task*) on top of this class. Using different objective functions is simply a case of swapping out a single line of code. In our examples, we will make use of the *Adaptive Gradient Algorithm* (Adagrad), which is an SGD optimiser. It works by maintaining low learning rates (usually denoted α) for frequently occuring features, and high learning rates for less frequent features. Where we can, we will use *Root Mean Squared Error* (RMSE) as our metric for model comparison.

Preliminary Data Analysis

Objective

The following notebook aims to perform some preliminary data analysis on the last.fm dataset used at HetRec2011 which can be found Here. We aim to perform the following analysis:

- **Missing Values**
  - Test for the occurence of missing values across our data
  - Understand the impact of missing values on our analysis
- **Data Distribution**
  - Is a particular genre over or under represented?
  - With respect to users-artists, what is the distribution of ‘number of listens’?

As well as these defined objectives, this notebook intends to explore our data, and develop a sense for the nature of the information and how disparate data sources relate to one another.

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

```
#Let's view the user-artist data first
user_artist = pd.read_csv('../data/user_artists.dat', sep='\t', encoding='latin-1')
user_artist.head()
```

	userID	artistID	weight
0	2	51	13883
1	2	52	11690
2	2	53	11351
3	2	54	10300
4	2	55	8983

Missing Values

```
#Test for the occurence of missing values
import os
for file in os.listdir('../data/'):
    if file != 'readme.txt':
        df = pd.read_csv(f'../data/{file}', sep='\t', encoding='latin-1')
        print(f'Reading file: {file} | Contains missing data: {df.isnull().values.any()}')
```

```
Reading file: artists.dat | Contains missing data: True
```

```
---------------------------------------------------------------------------
PermissionError                           Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_6512/1821401783.py in <module>
      3 for file in os.listdir('../data/'):
      4     if file != 'readme.txt':
----> 5         df = pd.read_csv(f'../data/{file}', sep='\t', encoding='latin-1')
      6         print(f'Reading file: {file} | Contains missing data:
{df.isnull().values.any()}')

~\AppData\Roaming\Python\Python38\site-packages\pandas\io\parsers.py in
parser_f(filepath_or_buffer, sep, delimiter, header, names, index_col, usecols, squeeze,
prefix, mangle_dupe_cols, dtype, engine, converters, true_values, false_values,
skipinitialspace, skiprows, skipfooter, nrows, na_values, keep_default_na, na_filter, verbose,
skip_blank_lines, parse_dates, infer_datetime_format, keep_date_col, date_parser, dayfirst,
cache_dates, iterator, chunksize, compression, thousands, decimal, lineterminator, quotechar,
quoting, doublequote, escapechar, comment, encoding, dialect, error_bad_lines, warn_bad_lines,
delim_whitespace, low_memory, memory_map, float_precision)
    674         )
    675 
--> 676         return _read(filepath_or_buffer, kwds)
    677 
    678     parser_f.__name__ = name

~\AppData\Roaming\Python\Python38\site-packages\pandas\io\parsers.py in
_read(filepath_or_buffer, kwds)
    446 
    447     # Create the parser.
--> 448     parser = TextFileReader(fp_or_buf, **kwds)
    449 
    450     if chunksize or iterator:

~\AppData\Roaming\Python\Python38\site-packages\pandas\io\parsers.py in __init__(self, f,
engine, **kwds)
    878             self.options["has_index_names"] = kwds["has_index_names"]
    879 
--> 880         self._make_engine(self.engine)
    881 
    882     def close(self):

~\AppData\Roaming\Python\Python38\site-packages\pandas\io\parsers.py in _make_engine(self,
engine)
   1112     def _make_engine(self, engine="c"):
   1113         if engine == "c":
-> 1114             self._engine = CParserWrapper(self.f, **self.options)
   1115         else:
   1116             if engine == "python":

~\AppData\Roaming\Python\Python38\site-packages\pandas\io\parsers.py in __init__(self, src,
**kwds)
   1872         if kwds.get("compression") is None and encoding:
   1873             if isinstance(src, str):
-> 1874                 src = open(src, "rb")
   1875                 self.handles.append(src)
   1876 

PermissionError: [Errno 13] Permission denied: '../data/songs'
```

```python
#Investigating missing data in artists.dat
artists = pd.read_csv('../data/artists.dat', sep='\t', encoding='latin-1')
artists.shape
```

```
(17632, 4)
```

```python
#Total number of rows in artists with missing values
artists[artists.isna().any(axis=1)].shape
```

```
(444, 4)
```

```python
#Columns in artists with missing values
for col in artists.columns:
    print(f'Column: {col} | Contains missing data: {artists[col].isnull().values.any()}')
```

```
Column: id | Contains missing data: False
Column: name | Contains missing data: False
Column: url | Contains missing data: False
Column: pictureURL | Contains missing data: True
```

```
#Let's check that artist names and ID's are unique
print(artists.id.is_unique)
print(artists.name.is_unique)
```

```
True
True
```

Findings

The data is very clean with $5/6$ datasets having no missing values. The artists data does contain missing values, however only in the 'pictureURL' column. This column will not be a fundamental part of any analysis. Therefore, there is no impact of missing values on our data.

Data Distribution

```
#Let's investigate how many users we have
print(len(user_artist.userID.unique()))

#Let's investigate the number of artists
print(len(artists.id.unique()))
```

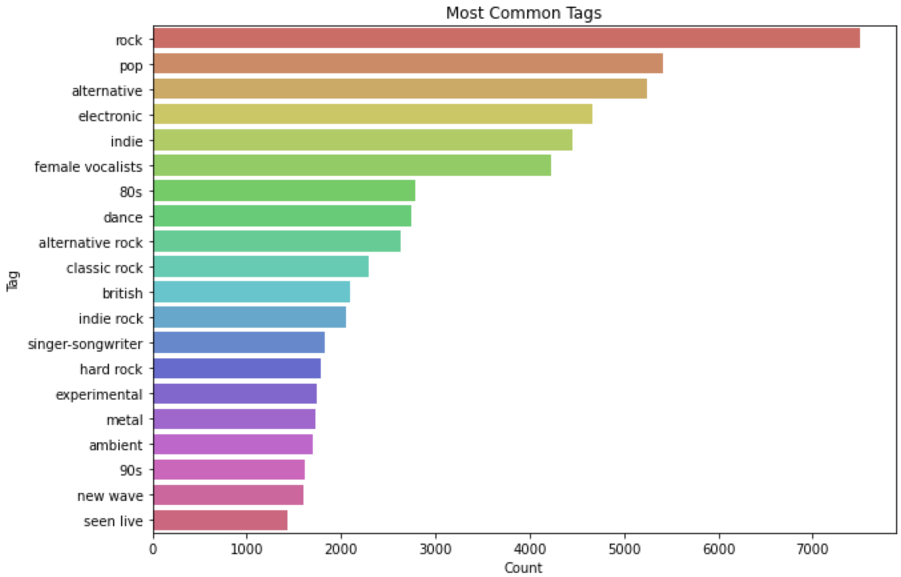
```
1892
17632
```

```
# Let's investigate how different genres are represented in our data
tagged_artists = pd.read_csv('../data/user_taggedartists.dat', sep='\t', encoding='latin-1')
tags = pd.read_csv('../data/tags.dat', sep='\t', encoding='latin-1')
tagged_artists.head()
```

	userID	artistID	tagID	day	month	year
0	2	52	13	1	4	2009
1	2	52	15	1	4	2009
2	2	52	18	1	4	2009
3	2	52	21	1	4	2009
4	2	52	41	1	4	2009

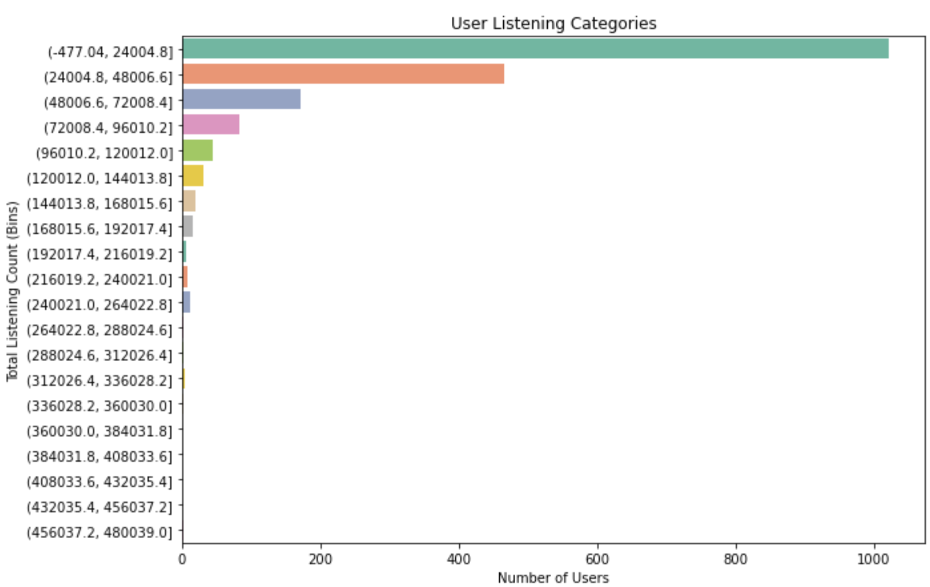
```
#Let's count the tags that appear
tag_count = tagged_artists[['tagID', 'artistID']].groupby('tagID').count().reset_index()
tag_count.rename({'artistID':'count'}, axis=1, inplace=True)

#Find string text values
tag_text = tag_count.merge(tags, on='tagID')

#Extract top 20 tags
top_20_tags = tag_text.sort_values(by='count', ascending=False).iloc[:20]
```

```
# Initialize the matplotlib figure
f, ax = plt.subplots(figsize=(10, 7))

col = sns.color_palette("hls", 20)

sns.barplot(x='count', y='tagValue', data=top_20_tags,
            label="tagValue", palette=col)

plt.title('Most Common Tags')
plt.ylabel('Tag')
plt.xlabel('Count')
plt.show()
```

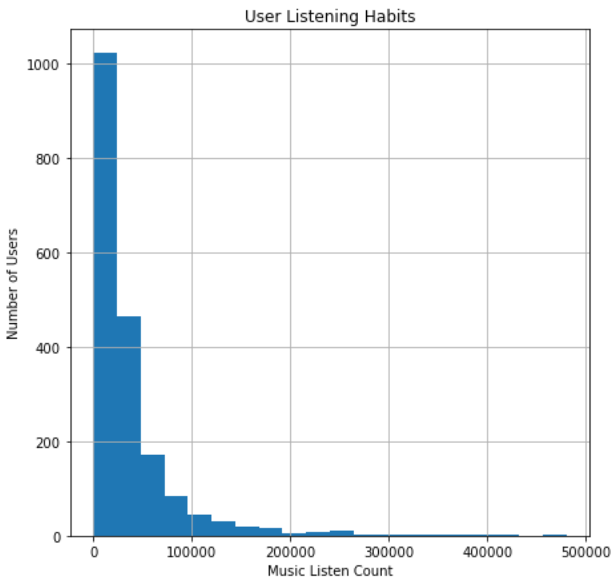



```python
#Let's investigate the distribution of number of total listens by users
listen_per_user = user_artist[['userID', 'weight']].groupby('userID').sum()

#Give each user a bin to fit in
listen_per_user['bin'] = pd.cut(listen_per_user.weight, 20, precision=2)

#Count how many users per bin
count_per_bin = listen_per_user.groupby('bin').count()
```

```python
#Plot the bins
f, ax = plt.subplots(figsize=(10, 7))

col = sns.color_palette("Set2", 8)

sns.barplot(x='weight', y='bin', data=count_per_bin.reset_index(),
            label="bin", palette=col)

plt.title('User Listening Categories')
plt.ylabel('Total Listening Count (Bins)')
plt.xlabel('Number of Users')
plt.show()
```



```
#Same graph but as a conventional histogram
plt.figure(figsize=(7,7))
listening_habits = user_artist[['userID', 'weight']].groupby('userID').sum()

plt.hist(listening_habits.weight, bins=20)
plt.title('User Listening Habits')
plt.xlabel('Music Listen Count')
plt.ylabel('Number of Users')
plt.grid(True)
plt.show()
```




```
# What portion of users are contained within the first bins
bin_1 = count_per_bin.iloc[0][0]
bin_2 = count_per_bin.iloc[1][0]
bin_3 = count_per_bin.iloc[2][0]
total_pop = len(user_artist.userID.unique())
print(f'Users in largest bin: {bin_1} | Portion of total: {(bin_1/total_pop)*100:.2f}%')
print(f'Users in second largest bin: {bin_2} | Portion of total: {(bin_2/total_pop)*100:.2f}%')
print(f'Users in third largest bin: {bin_3} | Portion of total: {(bin_3/total_pop)*100:.2f}%')

print(f'The mean of user listening habits: {listening_habits.weight.mean():.2f}')
```

```
Users in largest bin: 1022 | Portion of total: 54.02%
Users in second largest bin: 466 | Portion of total: 24.63%
Users in third largest bin: 171 | Portion of total: 9.04%
The mean of user listening habits: 36566.58
```

Findings

As this dataset was used in 2011, the general music trends are those observed for that period. These musical trends differ substantially from the music we listen to today. Therefore, recommender systems build ontop of this data will produce recommendations biased towards that time period. This is not an error, rather just a property of the data.

We see that **rock** features heavily in the top 20 tags found throughout the data, arising in many forms from metal and hard, to classic and indie. In terms of users listening habits, we observe that the vaste majority of users (**54%**) have listened to songs 24,000 or less times. By the third bin, a cummulative **87%** of users have listened to music 72,000 times or less. The majority of the final bins contain extraim outliers such as users who have listened to music over 480,000 times.

I did expect user listening habits to follow more of a normal distribution, and was surprised to find the distribution follows a curve more closely related to exponential decay. The distribution is skewed to the right and decreases sharply. The mean is 36,566 listens.

Conclusions

The usability of this data is quite high, therefore requiring only a short preliminary analysis. The data is clean and structured, featuring missing values only for unimportant features like ‘pictureURL’. There are no duplicate artists featured in the data.

An analysis of the user listening habits allows us to better understand the distribution of the data we are using. Investigation of the most popular tags used to describe music in this dataset helped us understand trends which are representative of the time period in which the data was released. With this knowledge, we will better understand the output of the recommender system we are developing later on.

Collaborative Filtering - Matrix Factorization

In this Notebook, we will be performing collaborative filtering using matrix factorization. The steps we will follow are detailed in the introduction. We will be using TensorFlow as our ML framework for the development of our music recommender system. Let's begin by importing our packages, and building a sparse representation of our ratings matrix.

```
import pandas as pd
import numpy as np
import tensorflow.compat.v1 as tf
import collections
from mpl_toolkits.mplot3d import Axes3D
from IPython import display
import sklearn
import sklearn.manifold
from matplotlib import pyplot as plt
tf.compat.v1.disable_eager_execution()
```

We are importing a range of helper functions in the below cell. These functions are defined in `CFUtils.py`, and `CFModel.py`, both of which can be found in the Github Repository for this assignment.

```
#Let's import our helper functions defined externally
from CFUtils import build_rating_sparse_tensor, split_dataframe, sparse_mean_square_error,
build_model, item_neighbors, user_recommendations
from CFModel import CFModel
```

```
# Add some convenience functions to Pandas DataFrame.
pd.options.display.max_rows = 10
pd.options.display.float_format = '{:.3f}'.format
def mask(df, key, function):
    """Returns a filtered dataframe, by applying function to key"""
    return df[function(df[key])]

def flatten_cols(df):
    df.columns = [' '.join(col).strip() for col in df.columns.values]
    return df

pd.DataFrame.mask = mask
pd.DataFrame.flatten_cols = flatten_cols
```

Feature Engineering

In the following cells, we will be re-mapping our user ID's to fit into a scale that is defined as $[0, m]$, where m is our number of unique users. Currently, despite there being only 1,892 unique users, profiles exist with ID's such as 1,893, 2,000, and so on. This is an issue because when we generate our ratings matrix, we will be creating rows that do not represent any users. Mapping our user ID's to a suitable scale is good practice and will avoid issues in the future.

The case is the same for artist ID's with them not being presented in a contiguous order. We will apply the same principal above to remap our artist ID's to the correct scale.

```
#Let's define our amount of users
rating_matrix = pd.read_csv('../data/user_artists.dat', sep='\t', encoding='latin-1')
num_users = len(rating_matrix.userID.unique())

#Extract userID column
userids = np.asarray(rating_matrix.userID)

#Remap the column
u_mapper, u_ind = np.unique(userids, return_inverse=True)
```

```
#Let's define our amount of artists
artists = pd.read_csv('../data/artists.dat', sep='\t', encoding='latin-1')
artists.rename(columns={'id':'artistID'}, inplace=True)
num_artists = len(artists.artistID.unique())

#Extract artistID column
artistids = np.asarray(rating_matrix.artistID)

#Remap the column
a_mapper, a_ind = np.unique(artistids, return_inverse=True)
```

```
#Assert that u_ind and userID column are of same size
assert(len(u_ind) == len(rating_matrix.userID))

#Assert that a_ind and artistID column are of same size
assert(len(a_ind) == len(rating_matrix.artistID))
```

```
# Let's replace old columns with new ind ones
rating_matrix.userID = u_ind
rating_matrix.artistID = a_ind

#Let's ensure the max value is approriate
assert(rating_matrix.userID.unique().max() == 1891)
assert(rating_matrix.artistID.unique().max() == 17631)
```

The Problem With Rating Values

For this collaborative filtering model, I would have liked to make use of the explicit rating data available in the 'weight' column of the ratings matrix. Through trial and error, I've come to the conclusion that using this data explicitly will not create an effective recommendation system.

The reason for this is to do with the the distribution of user to artist listens. Many user to artist combinations have only a value of 1 for the 'weight' column. When aggregating the total listens for artists by summing the weights, we see a huge portion of the artists still only have 1 or so listens, whilst popular artists such as Katy Perry, Lady Gaga, and Britney Spears, can accumulate millions of listens, Britney Spears coming out on top with 2.4 million listens from users in this dataset. This problem is known as the *Long Tail Problem* [Anderson, 2006].

Intuitively, this is not suitable for a recommendation system because in this case if we ignore the outliers, we are ignoring the most important features of the dataset. Normalising our 'weight' column has little-to-no effect on performance as squashing such disparate values into a smaller range does not fix the disparity.

Another approach is to use a binary representation of the ratings matrix. If a user has ever listened to an artist, the 'listened' column receives a value of 1. Otherwise, the value is 0. This approach does result in some information loss, as we are no longer aware of how regular of a listener a user is to a particular artist.

```
#We're going to make a binary representation of the matrix. If the user ever listened to the
artist, they get a value of 1. 0 otherwise
rating_matrix['listened'] = 1.0
```

```
rating_matrix.describe()
```

	userID	artistID	weight	listened
count	92834.000	92834.000	92834.000	92834.000
mean	944.222	3235.737	745.244	1.000
std	546.751	4197.217	3751.322	0.000
min	0.000	0.000	1.000	1.000
25%	470.000	430.000	107.000	1.000
50%	944.000	1237.000	260.000	1.000
75%	1416.000	4266.000	614.000	1.000
max	1891.000	17631.000	352698.000	1.000

Adding Genres to Artists

For visualising our recommendations, we will view the dispersion of genres in latent space. This will allow us to understand if music of similar genres generally cluster around each other. For this, we would like to be able to join our artists and genres tables.

We must firstly merge the `tags.dat` data with the `user_taggedartists.dat` data. We then `groupby` artist name and use a lambda function to list the genres for that artist. It should be noted that `user_taggedartists.dat` is user-generated information, and therefore may contain some innacurate or unimportant artist tags. To counteract this, we will only be including a tag as part of the artists genre if it has been associated with that artist on 3 or more seperate occasions.

```
#Let's read in genres and tags
genres = pd.read_csv('../data/tags.dat', sep='\t', encoding='latin-1')
artists_tagged = pd.read_csv('../data/user_taggedartists.dat', sep='\t', encoding='latin-1')
```

```
#Let's match artists to genres
artists_tagged = artists_tagged.merge(genres[['tagID', 'tagValue']], on='tagID')
artists_tagged = (artists_tagged.groupby('artistID')['tagValue'].apply(lambda grp:
list(grp))).reset_index()
```

We know that as tags are applied by users, they can be innaccurate and messy. In the following cell we perform these actions:

1. Loop through each artist in our `artists_tagged` table
2. Create a dictionary for that artist's tags
3. Count the tags using the dictionary
4. Order the artist tags by number of appearances and save them in a `new_tags` variable.
5. The artist receives the `new_tags` variable ("No Tags" string if list is empty.)

```
for index, row in artists_tagged.iterrows():
    d = {}
    new_tags = []
    for val in row.tagValue:
        if val not in d:
            d[val] = 1
        else:
            d[val] += 1
    for key, value in d.items():
        if d[key] >=3:
            new_tags.append([key, value])
    new_tags.sort(key=lambda x:x[1], reverse=True)
    if new_tags:
        artists_tagged.at[index, "tagValue"] = [tag[0] for tag in new_tags]
        artists_tagged.at[index, 'genre'] = artists_tagged.at[index, 'tagValue'][0]
```

```
#Let's add these tags to our artists
artists = artists.join(artists_tagged, on='artistID', how='left', rsuffix='right')
artists.tagValue = artists.tagValue.fillna('No Tags')
artists.genre = artists.genre.fillna('No Tags')
artists.drop(columns=['artistIDright', 'url', 'pictureURL'], inplace=True)
artists.rename(columns={'tagValue': 'genres'}, inplace=True)
```

```
artists
```

	artistID	name	genres	genre
0	1	MALICE MIZER	[darkwave, german, gothic]	darkwave
1	2	Diary of Dreams	[black metal]	black metal
2	3	Carpathian Forest	[j-rock, japanese, visual kei, gothic]	j-rock
3	4	Moi dix Mois	[darkwave]	darkwave
4	5	Bella Morte	[gothic metal, doom metal, black metal]	gothic metal
...	...	...	...	...
17627	18741	Diamanda GalÃ¡s	No Tags	No Tags
17628	18742	Aya RL	No Tags	No Tags
17629	18743	Coptic Rain	No Tags	No Tags
17630	18744	Oz Alchemist	No Tags	No Tags
17631	18745	Grzegorz Tomczak	No Tags	No Tags

17632 rows × 4 columns

Artist Total Listens Value

Later on we will use an artists total number of listens as part of our visualisations.

```
#Let's groupby artist ID and sum weight for their total listens
artist_tot_listens = rating_matrix[['artistID', 'weight']].groupby('artistID').count()

#Let's also add the artist name for clarity
artist_tot_listens = artist_tot_listens.join(other=artists, lsuffix='artistID', rsuffix='id')
[['name', 'weight']].reset_index()
```

Collaborative Filtering Model

At this point, we are ready to build a very basic first version of our collaborative filtering model. In the below code, we will be using a modified version of the `build_model` function defined in `CFUtils.py`. We modified this function to include two extra parameters:

- `num_queries` - Defined as the number of users in our case. Used to define the length of the U user embedding.
- `num_items` - Defined as the number of artists in our case. Used to define the length of the V item embedding.

The helper functions in `CFUtils.py`, and the `CFModel` class provided in `CFModel.py` were designed specifically for Recommendation System Colab, meaning that the length of the embedding vectors were hard-coded in. The inclusion of these two extra parameters allows us to apply this model for new use-cases. In the cell below, we will do a trial run of our model and discuss the results after.

```
# Build the CF model and train it.
model = build_model(rating_matrix, num_queries=num_users, num_items=num_artists,
embedding_dim=30, init_stddev=0.05)
model.train(num_iterations=1000, learning_rate=10.)
```

```
WARNING:tensorflow:From D:\DCU\4th_year\CA4015\CA4015-Music-RecSystem\book\CFModel.py:55: start_queue_runners (from tensorflow.python.training.queue_runner_impl) is deprecated and will be removed in a future version.
Instructions for updating:
To construct input pipelines, use the `tf.data` module.
```

```
WARNING:tensorflow:`tf.train.start_queue_runners()` was called when no queue runners were defined. You can safely remove the call to this deprecated function.
```

```
 iteration 0: train_error=1.000140, test_error=0.999945
```

```
 iteration 10: train_error=0.999423, test_error=0.999942
```

```
 iteration 20: train_error=0.998703, test_error=0.999937
```

```
 iteration 30: train_error=0.997972, test_error=0.999921
```

```
 iteration 40: train_error=0.997219, test_error=0.999888
```

```
 iteration 50: train_error=0.996433, test_error=0.999827
```

```
 iteration 60: train_error=0.995600, test_error=0.999725
```

```
 iteration 70: train_error=0.994699, test_error=0.999566
```

```
 iteration 80: train_error=0.993707, test_error=0.999329
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_13476/1265995664.py in <module>
      1 # Build the CF model and train it.
      2 model = build_model(rating_matrix, num_queries=num_users, num_items=num_artists,
embedding_dim=30, init_stddev=0.05)
----> 3 model.train(num_iterations=1000, learning_rate=10.)

D:\DCU\4th_year\CA4015\CA4015-Music-RecSystem\book\CFModel.py in train(self, num_iterations,
learning_rate, plot_results, optimizer)
     63             # Train and append results.
     64             for i in range(num_iterations + 1):
---> 65                 _, results = self._session.run((train_op, metrics))
     66                 if (i % 10 == 0) or i == num_iterations:
     67                     print("\r iteration %d: " % i + ", ".join(

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\client\session.py in
run(self, fetches, feed_dict, options, run_metadata)
    968 
    969     try:
--> 970       result = self._run(None, fetches, feed_dict, options_ptr,
    971                          run_metadata_ptr)
    972       if run_metadata:

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\client\session.py in
_run(self, handle, fetches, feed_dict, options, run_metadata)
   1191     # or if the call is a partial run that specifies feeds.
   1192     if final_fetches or final_targets or (handle and feed_dict_tensor):
-> 1193       results = self._do_run(handle, final_targets, final_fetches,
   1194                              feed_dict_tensor, options, run_metadata)
   1195     else:

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\client\session.py in
_do_run(self, handle, target_list, fetch_list, feed_dict, options, run_metadata)
   1371 
   1372     if handle is None:
-> 1373       return self._do_call(_run_fn, feeds, fetches, targets, options,
   1374                            run_metadata)
   1375     else:

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\client\session.py in
_do_call(self, fn, *args)
   1378   def _do_call(self, fn, *args):
   1379     try:
-> 1380       return fn(*args)
   1381     except errors.OpError as e:
   1382       message = compat.as_text(e.message)

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\client\session.py in
_run_fn(feed_dict, fetch_list, target_list, options, run_metadata)
   1361       # Ensure any changes to the graph are reflected in the runtime.
   1362       self._extend_graph()
-> 1363       return self._call_tf_sessionrun(options, feed_dict, fetch_list,
   1364                                       target_list, run_metadata)
   1365 

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\client\session.py in
_call_tf_sessionrun(self, options, feed_dict, fetch_list, target_list, run_metadata)
   1454   def _call_tf_sessionrun(self, options, feed_dict, fetch_list, target_list,
   1455                           run_metadata):
-> 1456     return tf_session.TF_SessionRun_wrapper(self._session, options, feed_dict,
   1457                                             fetch_list, target_list,
   1458                                             run_metadata)

KeyboardInterrupt: 
```

In the above graph, it can be observed that both `train_eror` and `test_error` are small values, indicating good performance of the model overall. However, we already know that the recommendations for this basic model will likely be poor.

Let's take a closer look at our embeddings. We'll view the embeddings near 'Adele'.

```
item_neighbors(model,title_substring="Adele", measure='dot', items=artists)
item_neighbors(model, title_substring="Adele", measure='cosine', items=artists)
```

```
Nearest neighbors of : Adele.
```

	dot score	name
1919	1.196	Adele
1444	1.168	Selena Gomez
334	1.164	Cher
518	1.161	Jeffree Star
303	1.153	Danity Kane
1449	1.152	Miranda Cosgrove

```
Nearest neighbors of : Adele.
```

	cosine score	name
1919	1.000	Adele
1444	0.954	Selena Gomez
1449	0.951	Miranda Cosgrove
518	0.950	Jeffree Star
532	0.950	Maroon 5
680	0.949	Selena Gomez & the Scene

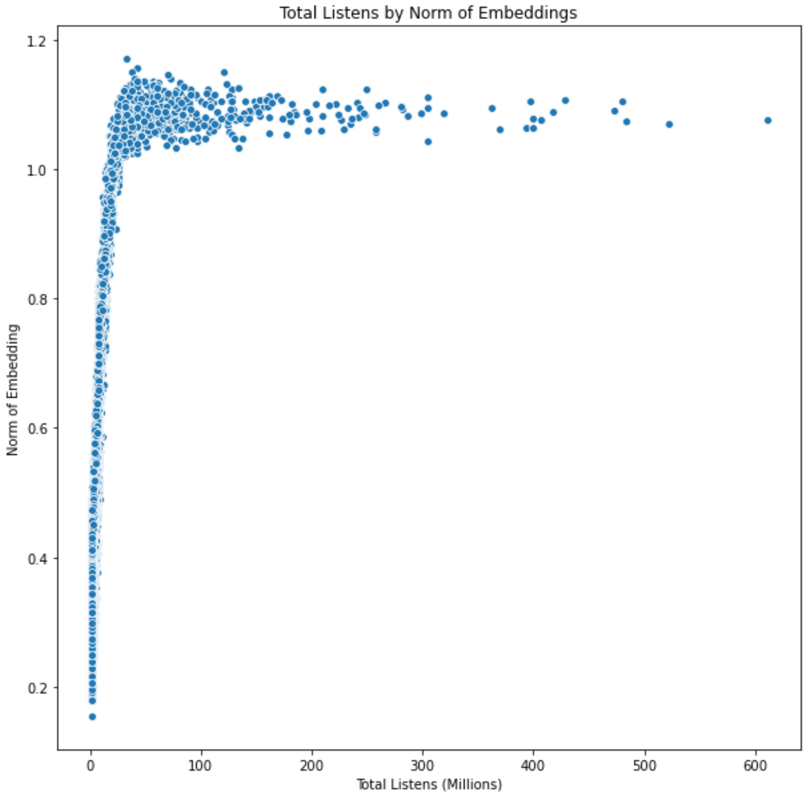
Current Model Predictions

At a glance, the embeddings closest to Adele seem reasonable, in the sense that they are all popular artist names that are somewhat familiar. This makes sense, Adele being one of the most popular artists in 2011.

With basic recommendation systems, we often observe the appearance of 'niche' items (artists in our case) near our embeddings. This is due to the fact that these little-known artists can have a randomly assigned norm with a high value that is never corrected. We somewhat avoid this issue in our model generation by incorporating a relatively small `init_stddev` value of 0.05.

```
from CFUtils import tsne_item_embeddings, visualize_item_embeddings, item_embedding_norm,
build_regularized_model
```

```
item_embedding_norm(model, artist_tot_listens, rating_matrix)
```



Norm Analysis

Viewing the relation between the number of users who listen to an artist and that artist's norm, we see that our norm values are somewhat reasonable. The values seem to plateaux near the top. This results in lesser known artists unreasonably receiving higher norm values. Again, instantiating the norms using a normal distribution with a small standard deviation ensures that few niche artists receive high norms. This does not completely solve the issue.

To correct this behaviour, we can use regularization terms. Regularization terms introduce some bias in to our data which aims to reduce the impact of the behaviour previously discussed. We should expect to see less-performant error scores for our regularized model. This is okay however, as we should see improved recommendations being made.

Regularisation

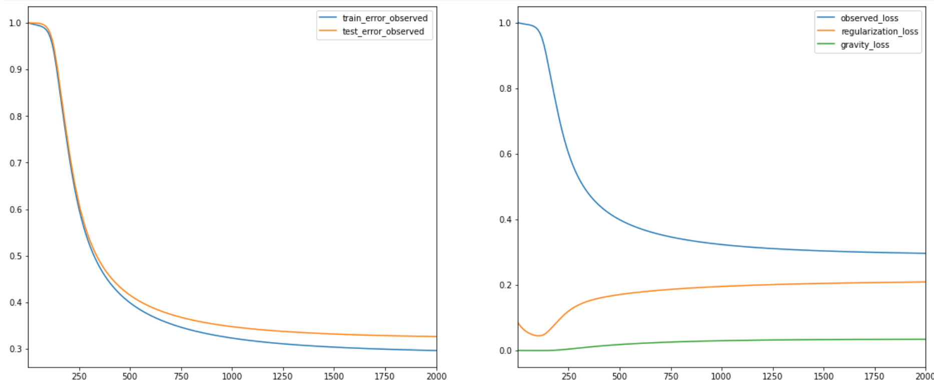
We'll now use two regularization techniques defined and discussed in our introduction section. Namely, *L2 Regularization*, and *Gravity Term* regularization. Essentially, these two terms aim to correct embeddings after they are initialised.

Our `build_regularized_model()` helper function defines both of our regularization terms and combines them into our `total_loss` function which is then provided to our `CFModel` class in place of the previous simplistic loss function.

```
reg_model = build_regularized_model(
    rating_matrix, num_queries=num_users , num_items=num_artists, regularization_coeff=0.5,
    gravity_coeff=1.0, embedding_dim=35, init_stddev=.05)
reg_model.train(num_iterations=2000, learning_rate=20.)
```

```
WARNING:tensorflow:From D:\DCU\4th_year\CA4015\CA4015-Music-RecSystem\book\CFModel.py:55:
start_queue_runners (from tensorflow.python.training.queue_runner_impl) is deprecated and will
be removed in a future version.
Instructions for updating:
To construct input pipelines, use the `tf.data` module.
WARNING:tensorflow:`tf.train.start_queue_runners()` was called when no queue runners were
defined. You can safely remove the call to this deprecated function.
 iteration 2000: train_error_observed=0.296637, test_error_observed=0.326611,
observed_loss=0.296637, regularization_loss=0.209359, gravity_loss=0.034404
```

```
[{'train_error_observed': 0.29663688, 'test_error_observed': 0.32661134},
 {'observed_loss': 0.29663688,
  'regularization_loss': 0.20935923,
  'gravity_loss': 0.034403637}]
```



Regularised Model Analysis

If we compare the graph on the left of our regularised model to the graph of our unregularised model, we can see that our regularised model has higher loss values. Higher loss values for an MF recommender system can be favourable to a certain degree. This is because our scoring functions ("DOT" and "COSINE") regularly make recommendations for items based on popularity. It's important for recommendation systems to include some type of personalisation in their recommendations, otherwise they will just recommend the most popular items to all users, which is useful to no one.

As before, we can now take a closer look at our embeddings. Let's once again view the embeddings near Adele, and try to interpret and compare them to our previous embeddings for our unregularised model. Let's first inspect the dot product as our measure, followed by cosine.

```
item_neighbors(reg_model,title_substring="Adele", measure='dot', items=artists)
item_neighbors(model,title_substring="Adele", measure='dot', items=artists)
```

```
Nearest neighbors of : Adele.
```

	dot score	name
83	4.443	Lady Gaga
61	4.409	Madonna
283	4.404	Britney Spears
282	4.375	Rihanna
294	4.375	Katy Perry
327	4.375	Avril Lavigne

```
Nearest neighbors of : Adele.
```

	dot score	name
1919	1.196	Adele
1444	1.168	Selena Gomez
334	1.164	Cher
518	1.161	Jeffree Star
303	1.153	Danity Kane
1449	1.152	Miranda Cosgrove

Dot Product Comparison of Embeddings in Regularised and Unregularised

I think it's clear that although both models do make somewhat reasonable recommendations, the more appropriate embedding space is definitely found in the regularised model. The unregularised model seems to make less useful recommendations such as 'Miranda Cosgrove', and 'Jefree Star'. The regularised model returns popular, solo-female artists that may not make music exactly similar to Adele, but were equally as popular as Adele in 2011.

```
item_neighbors(model,title_substring="Adele", measure='cosine', items=artists)
item_neighbors(reg_model, title_substring="Adele", measure='cosine', items=artists)
```

```
Nearest neighbors of : Adele.
```

	cosine score	name
1919	1.000	Adele
1444	0.954	Selena Gomez
1449	0.951	Miranda Cosgrove
518	0.950	Jeffree Star
532	0.950	Maroon 5
680	0.949	Selena Gomez & the Scene

```
Nearest neighbors of : Adele.
```

	cosine score	name
1919	1.000	Adele
673	0.999	Glee Cast
542	0.999	Ellie Goulding
91	0.999	Duffy
1043	0.999	Pink
3061	0.999	Sara Bareilles

Cosine Comparison of Embeddings in Regularised and Unregularised

We see a similar behaviour here, in that the regularised model embeddings are clearly more suitable. The appearance of *Glee Cast* as the most similar artist to Adele did seem like a strange recommendation. However, I found an [article]() that gives meaning to this recommendation. In 2011, Glee aired a mash-up of Adele's "Rumor Has It / Someone Like You" in one of their episodes which saw their iTunes song release reach No.1 on the charts.

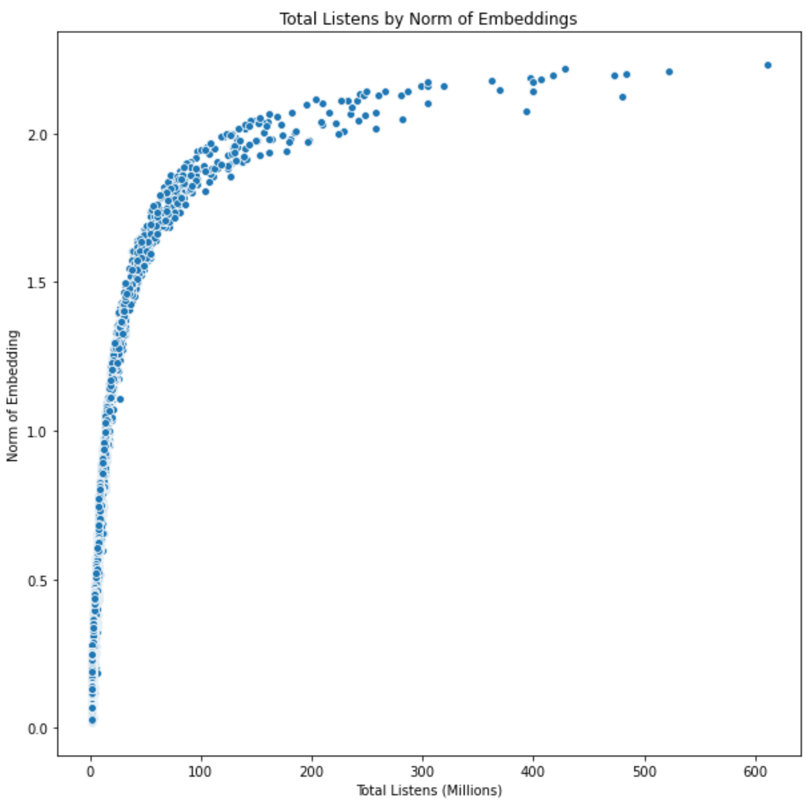
Once again, we see the regularised model recommending popular solo-female artists from that time. Both cosine and dot product recommendations seem suitable for Adele.

Norm Embedding Analysis

As previously, we will now view our model embeddings. We can see in the below image that the regularised model was able to break the plateaux previously observed in the unregularised model. As a result, this model is able to give more appropriate recommendations.

The below graph visually shows an improvement over our unregularised model. We see that embeddings no longer plateaux beyond a certain point, which leads to better recommendations.

```
item_embedding_norm(reg_model, artist_tot_listens, rating_matrix)
```



Deep Neural Network Recommender System

In this Notebook, we will build another music recommendation system. This time, we will incorporate a deep learning approach as explained in the introduction. We will be using Keras on Tensorflow V2 for this model. Keras is a high-level API built on-top of Tensorflow. It offers a range of utilities for getting started with recommender systems which we will use throughout this Notebook.

```python
import os
import pprint
import tempfile

from typing import Dict, Text

import numpy as np
import tensorflow as tf
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
```

```python
import tensorflow_recommenders as tfrs
import tensorflow_datasets as tfds
```

Data Preprocessing

Here we just repeat some data preprocessing steps as previous. However, we will incorporate a new way to normalise our weight values. In the previous notebook, the normalised weight values lead to poor performance of the model. Because of this, we opted to use a binary model. However, in this notebook we will normalise the weight column on a per user basis. This will be discussed shortly.

```python
#Let's define our amount of users
rating_matrix = pd.read_csv('../data/user_artists.dat', sep='\t', encoding='latin-1')
num_users = len(rating_matrix.userID.unique())

#Extract userID column
userids = np.asarray(rating_matrix.userID)

#Remap the column
u_mapper, u_ind = np.unique(userids, return_inverse=True)
```

```python
#Let's define our amount of artists
artists = pd.read_csv('../data/artists.dat', sep='\t', encoding='latin-1')
artists.rename(columns={'id':'artistID'}, inplace=True)
num_artists = len(artists.artistID.unique())

#Extract artistID column
artistids = np.asarray(rating_matrix.artistID)

#Remap the column
a_mapper, a_ind = np.unique(artistids, return_inverse=True)
```

```python
# Let's replace old columns with new ind ones
rating_matrix.userID = u_ind
rating_matrix.artistID = a_ind

#Let's ensure the max value is approriate
assert(rating_matrix.userID.unique().max() == 1891)
assert(rating_matrix.artistID.unique().max() == 17631)
```

```python
#We convert the ID's to string so we can use the StringLookup function later
rating_matrix.userID = rating_matrix.userID.apply(str)
rating_matrix.artistID = rating_matrix.artistID.apply(str)
```

Normalising the ‘weight’ Column

Rather than normalising the entire weight column as one array, we can normalise the values for each user individually. Simply put, the artist that a particular user listens to the most will have a high value such as 1. The artist they listen to the least will be closer to 0.

This is a better representation of a user’s preferences than normalising the entire weight column, which can lead to very insignificant values. Normalising the entire weight column will lead to user-artist pairs having small values for weight solely because other users have listened to more music in general.

In the following function, we create a new rating matrix which is populated by each users ratings one at a time. Once this is complete, we replace our old `rating_matrix` variable with `new_rating_matrix` to keep continuity of naming conventions.

```
#Let's normalise our weight column
new_rating_matrix = pd.DataFrame(columns=['userID', 'artistID', 'weight'])
for user_id in rating_matrix.userID.unique():
    user_ratings = rating_matrix[rating_matrix.userID == user_id]
    ratings = np.array(user_ratings['weight'])
    user_ratings['weight'] = tf.keras.utils.normalize(ratings, axis=-1, order=2)[0]
    new_rating_matrix = new_rating_matrix.append(user_ratings)
rating_matrix = new_rating_matrix
rating_matrix.describe()
```

```
C:\Users\seanc\AppData\Local\Temp/ipykernel_1744/4251931852.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  user_ratings['weight'] = tf.keras.utils.normalize(ratings, axis=-1, order=2)[0]
```

	weight
count	92834.000000
mean	0.091235
std	0.109804
min	0.000008
25%	0.030397
50%	0.062543
75%	0.109109
max	1.000000

Create DataSet

In the following cells, we will be creating a tensor DataSet using the `from_tensor_slices()` function. We must firstly create our `interactions_dict`, which is just our rating_matrix in dictionary form.

```
## we transform the table inta a dictionary , which then we feed into tensor slices
# this step is crucial as this will be the type of data fed into the embedding layers
interactions_dict = {name: np.array(value) for name, value in rating_matrix.items()}
interactions = tf.data.Dataset.from_tensor_slices(interactions_dict)

## we do similar step for item, where this is the reference table for items to be recommended
items_dict = rating_matrix[['artistID']].drop_duplicates()
items_dict = {name: np.array(value) for name, value in items_dict.items()}
items = tf.data.Dataset.from_tensor_slices(items_dict)

## map the features in interactions and items to an identifier that we will use throught the
embedding layers
## do it for all the items in interaction and item table
## you may often get itemtype error, so that is why here i am casting the quantity type as float
to ensure consistency
interactions = interactions.map(lambda x: {'userID' : x['userID'],
                                            'artistID' : x['artistID'],
                                            'weight' : x['weight']})

items = items.map(lambda x: x['artistID'])
```

We must also create variables representing our unique user ID’s and artist ID’s.

This will allow us to map the raw values of our categorical features to embeddings vectors in our models. For this, we need a vocabulary mapping raw feature values to an integfer in a contiguous range. This allows us to look up the corresponding embeddings in our tables.

```
### get unique item and user id's as a lookup table
unique_artist_ids = (np.unique(a_ind)).astype(str)
unique_user_ids = (np.unique(u_ind)).astype(str)

# Randomly shuffle data and split between train and test.
tf.random.set_seed(42)
shuffled = interactions.shuffle(100_000, seed=42, reshuffle_each_iteration=False)
train = shuffled.take(62_000)
test = shuffled.skip(62_000).take(30_000)
```

```
print(f'our test set is: {len(train)}')
print(f'our train set is: {len(test)}')
```

```
our test set is: 62000
our train set is: 30000
```

Simple Retrieval Model

In the following cells, we define a simple retrieval model. Although defined differently, this model works on the same principals as our previous matrix factorisation model. We just include this step as a progression towards a more advanced deep learning model.

The TFRS package with Keras makes development of recommender systems simple and intuitive. We simply inherit from the base `tfrs.Model` class, define our query (user) and item (artist) towers or embeddings, then define our loss function and metrics to be used and our model is ready to deploy.

```
#Basic retrieval model
    def __init__(self, user_model, item_model):
        super().__init__()

        ### we pass the embedding layer into item model
        self.item_model: tf.keras.Model = item_model

        ### We pass the embedding layer into user model
        self.user_model: tf.keras.Model = user_model

        ### for retrieval model. we take top-k accuracy as metrics
        metrics = tfrs.metrics.FactorizedTopK(candidates=items.batch(128).map(item_model))

        # define the task, which is retrieval
        task = tfrs.tasks.Retrieval(metrics=metrics)
        self.task: tf.keras.layers.Layer = task

    def compute_loss(self, features: Dict[Text, tf.Tensor], training=False) -> tf.Tensor:
        # We pick out the user features and pass them into the user model.
        user_embeddings = self.user_model(features["userID"])
        # And pick out the item features and pass them into the item model,
        # getting embeddings back.
        positive_item_embeddings = self.item_model(features["artistID"])

        # The task computes the loss and the metrics.
        return self.task(user_embeddings, positive_item_embeddings)
```

```
  File "C:\Users\seanc\AppData\Local\Temp/ipykernel_1744/2244004975.py", line 2
    def __init__(self, user_model, item_model):
    ^
IndentationError: unexpected indent
```

Train and Evaluate Model

The following cell is used to train and evaluate our model. We firstly define the size of our embedding dimensions. Then, we build our user and item embedding vectors using `tf.keras.layers.Embedding`. In this function, we add an additional embedding layer to account for unknown tokens (if any).

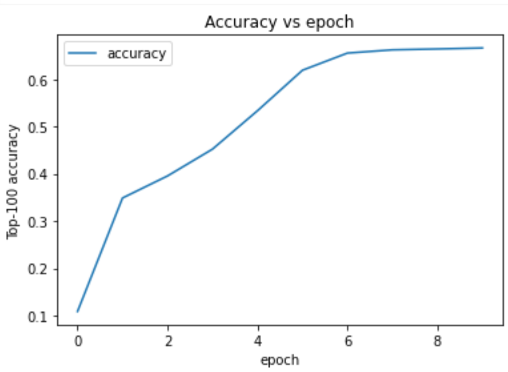
We simply instantiate our model and compile it with an optimisation function of our choice. We will use Adagrad in this instance. We then fit the model on our training data and provide our number of epochs. Following from this, we fit the data on our test set.

```
### Fitting and evaluating
### we choose the dimensionality of the query and candicate representation.
embedding_dimension = 32
## we pass the model, which is the same model we created in the query and candidate tower, into
the model
user_model = tf.keras.Sequential(
    [tf.keras.layers.experimental.preprocessing.StringLookup(
                                vocabulary=unique_user_ids, mask_token=None),
                                # We add an additional embedding to account for unknown tokens.
                                tf.keras.layers.Embedding(len(unique_user_ids) + 1,
embedding_dimension)])

item_model = tf.keras.Sequential(
    [tf.keras.layers.experimental.preprocessing.StringLookup(
                                vocabulary=unique_artist_ids, mask_token=None),
                                tf.keras.layers.Embedding(len(unique_artist_ids) + 1,
embedding_dimension)])

model = MusicModel(user_model, item_model)

# a smaller learning rate may make the model move slower and prone to overfitting, so we stick to
0.1
# other optimizers, such as SGD and Adam, are listed here
https://www.tensorflow.org/api_docs/python/tf/keras/optimizers
model.compile(optimizer=tf.keras.optimizers.Adagrad(learning_rate=0.1))
cached_train = train.shuffle(100_000).batch(10_000).cache()
cached_test = test.batch(10_000).cache()

## fit the model with ten epochs
model_hist = model.fit(cached_train, epochs=10)

#evaluate the model
model.evaluate(cached_test, return_dict=True)

# num_validation_runs =
len(one_layer_history.history["val_factorized_top_k/top_100_categorical_accuracy"])
epochs = [i for i in range(10)]

plt.plot(epochs, model_hist.history["factorized_top_k/top_100_categorical_accuracy"],
label="accuracy")
plt.title("Accuracy vs epoch")
plt.xlabel("epoch")
plt.ylabel("Top-100 accuracy");
plt.legend()
```

```
Epoch 1/10
7/7 [==============================] - 54s 7s/step -
factorized_top_k/top_1_categorical_accuracy: 0.0016 -
factorized_top_k/top_5_categorical_accuracy: 0.0114 -
factorized_top_k/top_10_categorical_accuracy: 0.0237 -
factorized_top_k/top_50_categorical_accuracy: 0.0783 -
factorized_top_k/top_100_categorical_accuracy: 0.1086 - loss: 72704.5215 - regularization_loss:
0.0000e+00 - total_loss: 72704.5215
Epoch 2/10
7/7 [==============================] - 57s 8s/step -
factorized_top_k/top_1_categorical_accuracy: 0.0080 -
factorized_top_k/top_5_categorical_accuracy: 0.0430 -
factorized_top_k/top_10_categorical_accuracy: 0.0805 -
factorized_top_k/top_50_categorical_accuracy: 0.2462 -
factorized_top_k/top_100_categorical_accuracy: 0.3486 - loss: 69661.6450 - regularization_loss:
0.0000e+00 - total_loss: 69661.6450
Epoch 3/10
7/7 [==============================] - 58s 8s/step -
factorized_top_k/top_1_categorical_accuracy: 0.0091 -
factorized_top_k/top_5_categorical_accuracy: 0.0512 -
factorized_top_k/top_10_categorical_accuracy: 0.0927 -
factorized_top_k/top_50_categorical_accuracy: 0.2789 -
factorized_top_k/top_100_categorical_accuracy: 0.3953 - loss: 65582.3188 - regularization_loss:
0.0000e+00 - total_loss: 65582.3188
Epoch 4/10
7/7 [==============================] - 58s 8s/step -
factorized_top_k/top_1_categorical_accuracy: 0.0099 -
factorized_top_k/top_5_categorical_accuracy: 0.0615 -
factorized_top_k/top_10_categorical_accuracy: 0.1117 -
factorized_top_k/top_50_categorical_accuracy: 0.3244 -
factorized_top_k/top_100_categorical_accuracy: 0.4520 - loss: 61863.0859 - regularization_loss:
0.0000e+00 - total_loss: 61863.0859
Epoch 5/10
7/7 [==============================] - 58s 8s/step -
factorized_top_k/top_1_categorical_accuracy: 0.0100 -
factorized_top_k/top_5_categorical_accuracy: 0.0696 -
factorized_top_k/top_10_categorical_accuracy: 0.1288 -
factorized_top_k/top_50_categorical_accuracy: 0.3822 -
factorized_top_k/top_100_categorical_accuracy: 0.5333 - loss: 58430.2305 - regularization_loss:
0.0000e+00 - total_loss: 58430.2305
```

```
Epoch 6/10
7/7 [==============================] - 59s 8s/step -
factorized_top_k/top_1_categorical_accuracy: 0.0091 -
factorized_top_k/top_5_categorical_accuracy: 0.0715 -
factorized_top_k/top_10_categorical_accuracy: 0.1387 -
factorized_top_k/top_50_categorical_accuracy: 0.4615 -
factorized_top_k/top_100_categorical_accuracy: 0.6190 - loss: 55323.5352 - regularization_loss:
0.0000e+00 - total_loss: 55323.5352
Epoch 7/10
7/7 [==============================] - 58s 8s/step -
factorized_top_k/top_1_categorical_accuracy: 0.0079 -
factorized_top_k/top_5_categorical_accuracy: 0.0679 -
factorized_top_k/top_10_categorical_accuracy: 0.1340 -
factorized_top_k/top_50_categorical_accuracy: 0.5257 -
factorized_top_k/top_100_categorical_accuracy: 0.6553 - loss: 52639.0669 - regularization_loss:
0.0000e+00 - total_loss: 52639.0669
Epoch 8/10
7/7 [==============================] - 58s 8s/step -
factorized_top_k/top_1_categorical_accuracy: 0.0060 -
factorized_top_k/top_5_categorical_accuracy: 0.0604 -
factorized_top_k/top_10_categorical_accuracy: 0.1397 -
factorized_top_k/top_50_categorical_accuracy: 0.5488 -
factorized_top_k/top_100_categorical_accuracy: 0.6621 - loss: 50468.8599 - regularization_loss:
0.0000e+00 - total_loss: 50468.8599
Epoch 9/10
7/7 [==============================] - 62s 9s/step -
factorized_top_k/top_1_categorical_accuracy: 0.0044 -
factorized_top_k/top_5_categorical_accuracy: 0.0600 -
factorized_top_k/top_10_categorical_accuracy: 0.1734 -
factorized_top_k/top_50_categorical_accuracy: 0.5566 -
factorized_top_k/top_100_categorical_accuracy: 0.6640 - loss: 48793.3975 - regularization_loss:
0.0000e+00 - total_loss: 48793.3975
Epoch 10/10
7/7 [==============================] - 59s 8s/step -
factorized_top_k/top_1_categorical_accuracy: 0.0037 -
factorized_top_k/top_5_categorical_accuracy: 0.0724 -
factorized_top_k/top_10_categorical_accuracy: 0.2022 -
factorized_top_k/top_50_categorical_accuracy: 0.5613 -
factorized_top_k/top_100_categorical_accuracy: 0.6661 - loss: 47525.4175 - regularization_loss:
0.0000e+00 - total_loss: 47525.4175
3/3 [==============================] - 29s 9s/step -
factorized_top_k/top_1_categorical_accuracy: 1.3333e-04 -
factorized_top_k/top_5_categorical_accuracy: 0.0015 -
factorized_top_k/top_10_categorical_accuracy: 0.0056 -
factorized_top_k/top_50_categorical_accuracy: 0.0702 -
factorized_top_k/top_100_categorical_accuracy: 0.1372 - loss: 89393.0645 - regularization_loss:
0.0000e+00 - total_loss: 89393.0645
```

```
<matplotlib.legend.Legend at 0x15d8642d3d0>
```



Simple Retrieval Model Analysis

The above model trains for 10 epoch and is then evaluated on testing data. In the training epoch information, we see information such as `top_5_categorical_accuracy`. This is indicative of the models ability to return a true-positive in the top-5 retrieved items (artists) from the entire set. We see that for each epoch, the model generally gets better at returning true-positives.

In the evaluation information, we see these accuracies drop significantly. This signifies that are model is overfitting to our training data. We can introduce regularisation terms as previous to mitigate this effect. Also, in deep learning models, we may introduce more user and artist features that help the model generalise better.

```python
# Create a model that takes in raw query features, and
index = tfrs.layers.factorized_top_k.BruteForce(model.user_model)
# recommends item out of the entire items dataset.
index.index_from_dataset(
    tf.data.Dataset.zip(items.batch(100).map(model.item_model)))

# Get recommendations.
j = str(10)
_, titles = index(tf.constant([j]))

arr = np.array(titles)[0]
names = []

for artist_id in arr:
    names.append(artists[artists['artistID'] == int(artist_id)]['name'])
names = np.asarray(names)
print(f"Recommendations for user %s: {names}" %(j))
```

```
Recommendations for user 10: [['Savage Garden']
 ['Ke$ha']
 ['Library Tapes']
 ['nevershoutnever!']
 ['Poney Express']
 ['Ra Ra Riot']
 ['The Rakes']
 ['Astrud Gilberto']
 ['Bon Iver']
 ['Metro Station']]
```

Ranking Model

In the following cells, we will upgrade from our previous model by using a ranking model. In this model, we will be making use of `dense` keras layers with 'RELU' activation functions. This is a deep learning approach. We will also be attempting to make use of the explicit rating values in the `weight` column of our rating matrix.

Moderm recommender systems usually operate in two stages:

- Retrieval
- Ranking

We will firstly define our `RankingModel` class which will act as our ranking model. Our `MusicModel` will act as the retrieval model as previous.

```python
#Define our ranking model
class RankingModel(tf.keras.Model):
    def __init__(self):
        super().__init__()
        embedding_dimension = 32

        ### we pass the embedding layer into item model
        self.item_model: tf.keras.Model = item_model

        ### We pass the embedding layer into user model
        self.user_model: tf.keras.Model = user_model

        # Compute predictions.
        self.ratings = tf.keras.Sequential([
                      # Learn multiple dense layers.
                      tf.keras.layers.Dense(256, activation="relu"),
                      tf.keras.layers.Dense(64, activation="relu"),
                      # Make rating predictions in the final layer.
                      tf.keras.layers.Dense(1)])

    def call(self, inputs):
        user_id, item_title = inputs
        user_embedding = self.user_model(user_id)
        item_embedding = self.item_model(item_title)
        return self.ratings(tf.concat([user_embedding, item_embedding], axis=1))
```

```python
#Now our base retrieval model incorporates our ranking model
class MusicModel(tfrs.models.Model):

    def __init__(self):
        super().__init__()
        self.ranking_model: tf.keras.Model = RankingModel()

        #Let's define our Loss function and optimisation function
        self.task: tf.keras.layers.Layer = tfrs.tasks.Ranking(
                          loss = tf.keras.losses.MeanSquaredError(),
                          metrics=[tf.keras.metrics.RootMeanSquaredError()])

    def call(self, features: Dict[str, tf.Tensor]) -> tf.Tensor:
        return self.ranking_model(
            (features["userID"], features["artistID"]))

    def compute_loss(self, features: Dict[Text, tf.Tensor], training=False) -> tf.Tensor:
        rating_predictions = self.ranking_model(
            (features["userID"], features["artistID"]))

        # The task computes the loss and the metrics.
        return self.task(labels=features["weight"], predictions=rating_predictions)
```

Note: This model differs to our previous retrieval model as we are now incorporating the weight column, whereas our previous model worked solely on binary values. This Deep Learning Framework is more robust and can handle the disparity between values in our weight column. We normalised our weight column values earlier. This is an important preprocessing step to achieve good model performance.

```python
### Fitting and evaluating
### we choose the dimensionality of the query and candicate representation.
embedding_dimension = 32
## we pass the model, which is the same model we created in the query and candidate tower, into
the model
user_model = tf.keras.Sequential(
    [tf.keras.layers.experimental.preprocessing.StringLookup(
                                vocabulary=unique_user_ids, mask_token=None),
                                # We add an additional embedding to account for unknown tokens.
                                tf.keras.layers.Embedding(len(unique_user_ids) + 1,
embedding_dimension)])

item_model = tf.keras.Sequential(
    [tf.keras.layers.experimental.preprocessing.StringLookup(
                                vocabulary=unique_artist_ids, mask_token=None),
                                tf.keras.layers.Embedding(len(unique_artist_ids) + 1,
embedding_dimension)])

model = MusicModel()

# a smaller learning rate may make the model move slower and prone to overfitting, so we stick to
0.1
# other optimizers, such as SGD and Adam, are listed here
https://www.tensorflow.org/api_docs/python/tf/keras/optimizers
model.compile(optimizer=tf.keras.optimizers.Adagrad(learning_rate=0.1))
cached_train = train.shuffle(100_000).batch(10_000).cache()
cached_test = test.batch(10_000).cache()

## fit the model with ten epochs
model_hist = model.fit(cached_train, epochs=10)

#evaluate the model
model.evaluate(cached_test, return_dict=True)
```

```
Epoch 1/10
7/7 [==============================] - 2s 78ms/step - root_mean_squared_error: 0.1133 - loss:
0.0129 - regularization_loss: 0.0000e+00 - total_loss: 0.0129
Epoch 2/10
7/7 [==============================] - 0s 48ms/step - root_mean_squared_error: 0.1097 - loss:
0.0123 - regularization_loss: 0.0000e+00 - total_loss: 0.0123
Epoch 3/10
7/7 [==============================] - 0s 48ms/step - root_mean_squared_error: 0.1096 - loss:
0.0123 - regularization_loss: 0.0000e+00 - total_loss: 0.0123
Epoch 4/10
7/7 [==============================] - 0s 44ms/step - root_mean_squared_error: 0.1095 - loss:
0.0123 - regularization_loss: 0.0000e+00 - total_loss: 0.0123
Epoch 5/10
7/7 [==============================] - 0s 48ms/step - root_mean_squared_error: 0.1095 - loss:
0.0123 - regularization_loss: 0.0000e+00 - total_loss: 0.0123
Epoch 6/10
7/7 [==============================] - 0s 50ms/step - root_mean_squared_error: 0.1094 - loss:
0.0122 - regularization_loss: 0.0000e+00 - total_loss: 0.0122
Epoch 7/10
7/7 [==============================] - 0s 49ms/step - root_mean_squared_error: 0.1094 - loss:
0.0122 - regularization_loss: 0.0000e+00 - total_loss: 0.0122
Epoch 8/10
7/7 [==============================] - 0s 45ms/step - root_mean_squared_error: 0.1093 - loss:
0.0122 - regularization_loss: 0.0000e+00 - total_loss: 0.0122
Epoch 9/10
7/7 [==============================] - 0s 45ms/step - root_mean_squared_error: 0.1093 - loss:
0.0122 - regularization_loss: 0.0000e+00 - total_loss: 0.0122
Epoch 10/10
7/7 [==============================] - 0s 48ms/step - root_mean_squared_error: 0.1093 - loss:
0.0122 - regularization_loss: 0.0000e+00 - total_loss: 0.0122
3/3 [==============================] - 1s 70ms/step - root_mean_squared_error: 0.1113 - loss:
0.0122 - regularization_loss: 0.0000e+00 - total_loss: 0.0122
```

```
{'root_mean_squared_error': 0.11132322996854782,
 'loss': 0.011565779335796833,
 'regularization_loss': 0,
 'total_loss': 0.011565779335796833}
```

Ranking Model Performance Analysis

We allowed our model to train for a total of 10 epochs before being evaluated on unseen testing data. Using RMSE as our objective function, we see this model is able to achieve great performance. The performance of our model does not drop significantly for unseen data.

Using RMSE as our metric allows us to quantitately compare the performance of this model with our previous matrix factorisation approach. In matrix factorisation, we achieved:

- `'train_error': 0.15067895, 'test_error': 0.23825963` for our unregularised model.
- `'train_error': 0.29663688, 'test_error': 0.32661134` for our regularised model.

We noted that the increase in error for our regularised model leads to better generalisations. In this deep learning model, we achieved:

- `'train_error': 0.1093, 'test_error': 0.1113`.

Qualitative Performance Analysis

Although the quantitative analysis of this model seems good, we do not know whether this leads to good recommendations or not. Let's perform a qualitative analysis of the model. We'll do the following:

- Take a typical user with a reasonable amount of artists they listen to
- Sample 5 random artists they listen to
- Along with the user id, provide the artist id's selected to our model
- We expect our model to return a ranking of these artist ID's in the order that the user is most interested in them.

```
#User 100
user_0 = rating_matrix[rating_matrix['userID'] == "100"]

#Sample 5 artists they listen to
user_0_sample = user_0.sample(5, random_state=74)
user_0_sample
```

	userID	artistID	weight
4986	100	2649	0.205387
4980	100	2146	0.164745
4990	100	2653	0.150230
4972	100	769	0.137167
4996	100	2659	0.083461

The following cell takes in artist ID's and a userID and computes a score for their combination. The model returns the artists in order of the scores they achieve. We expect this order to be similar to the order shown in the above sample.

```
test_ratings = {}
test_artist_ids = np.array(["2649", "2146", "2653", "769", "2659"])
for artistid in test_artist_ids:
    test_ratings[artistid] = model({
        "userID": np.array(["100"]),
        "artistID": np.array([artistid])
    })

print("Ratings:")
for artist, score in sorted(test_ratings.items(), key=lambda x: x[1], reverse=True):
    print(f"{artist}: {score}")
```

```
Ratings:
2649: [[0.09791555]]
2653: [[0.09428623]]
2146: [[0.0825875]]
769: [[0.08212695]]
2659: [[0.07627478]]
```

Output Analysis

We can see in the above cell that from the 5 sampled artists, our recommender system correctly identified the most liked artist as "2649". The system did rank "2653" in second place, when in reality that artist is the user's third favourite artist. However, the true ratings of the second and third most liked artist for this particular user differ only marginally.

From this small example, we can see that our model performs quite well.

Conclusion

In this notebook, we were able to create a two-part recommender system using Tensorflow V2. Modern recommender systems commonly adopt a two-step approach to making recommendations. The first step involves retrieving items that a particular query (user) will enjoy. The second step involves ranking this items and returning the ranked list to the user.

There exists plenty room to expand our model from here. TensorFlow offers a vast amount of API's for further expansion of recommender systems. In the grand scheme of things, this system is still considered quite trivial despite making strong recommendations. In our next notebook, we will attempt to incorporate some extra features into our recommendation system, such as genre tags as well as timestamps for when these genres were applied.

Advanced Deep Learning Recommender System

In the following notebook, we will be creating another Deep-Learning Recommender using Tensorflow V2. For this iteration, we will try to incorporate text and timestamp data available to us. As already stated multiple times, the tags in this data are user-generated. Therefore, they are messy, inconsistent, and may not be entirely accurate and or useful.

The TFRS package is incredibly robust, and offers plenty of direction for expansion of recommender systems. The library can tokenize text and timestamps into features. It processes text into a 'bag-of-words' representation, which it can then use to find similarities. It will be interesting to see if this approach alters recommendations to be affected more by the genre or tags associated with artists.

Similarly, it will be interesting to see how the inclusion of temporal data changes recommendations. In our data, a timestamp is associated with a *user*, *artist*, and *tag*. It indicates the exact time that particular user gave that artist that tag. It is entirely possible that amongst the tag information users who do not like particular artists have left negative tags. I wonder if an association will be made between few listens (*low weight*) and particular tag tokens.

```
import os
import pprint
import tempfile

from typing import Dict, Text

import numpy as np
import tensorflow as tf
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import tensorflow_recommenders as tfrs
```

Accounting for Tag Information

In the following cell, we will prove that in our data, users can tag the same artist multiple times. Preferably, we would like only 1 tag for any user-artist association. We will order a user-artist tags dataset by their creation time and use the most recent tag and timestamp for each user-artist combination.

```
#Let's read in genres and tags
tags = pd.read_csv('../data/tags.dat', sep='\t', encoding='latin-1')
user_tagging = pd.read_csv('../data/user_taggedartists.dat', sep='\t', encoding='latin-1')
user_tagging_time = pd.read_csv('../data/user_taggedartists-timestamps.dat', sep='\t',
encoding='latin-1')

#Check if duplicates are present
if True in user_tagging_time[['userID', 'artistID']].duplicated():
    print("Contains Duplicate user-artist combinations.")
```

```
Contains Duplicate user-artist combinations.
```

```
u_tag_a = user_tagging.merge(tags[['tagID', 'tagValue']], on='tagID')
u_tag_a = u_tag_a.merge(user_tagging_time, on=['userID', 'artistID', 'tagID'])
print("Displaying 3 random samples for tag data:")
u_tag_a.sample(3)
```

```
Displaying 3 random samples for tag data:
```

	userID	artistID	tagID	day	month	year	tagValue	timestamp
45368	791	13415	72	1	1	2009	hard rock	1230764400000
118264	364	7414	537	1	4	2008	neofolk	1207000800000
10291	47	4915	39	1	8	2010	dance	1280613600000

```
#Group by user-artist combo, sort by timestamp and extract that tagValue
u_tag_a = u_tag_a.sort_values(by='timestamp', ascending=False).groupby(['userID',
'artistID']).first().reset_index()
```

```
#Let's check if this dataset contains duplicate user-artist combinations
if True in np.unique(u_tag_a[['userID', 'artistID']].duplicated().values):
    print("Contains Duplicate user-artist combinations.")
else:
    print("Does not contain Duplicate user-artist combinations.")
```

```
Does not contain Duplicate user-artist combinations.
```

Data Preprocessing

Our preprocessing steps are as before for the most part. For a final step, we will merge our tag information dataset with our ratings matrix. To start, we normalise our weight column as previous.

There will be many cases where a user listens to a particular artist, but never provides that artist with a tag. In those cases, we will let the tag value be `no tag`, and for the corresponding timestamp value, we will use a value corresponding to today. We obviously want our model to find associations between users and common tags. However, our model can also build associations in situations where a user has decided to not provide a tag.

The timestamps provided in the dataset do not correspond to the correct year and must have the final 3 digits removed. For this, we can just divide them all by 1,000. Using [this website](), I entered some of the corrected timestamps to ensure they do indeed correspond to the appropriate year. All entries I checked returned values around 2008 to 2011, which makes sense for this dataset.

```
import time
import math

#Correct timestamp data in u_tag_a
u_tag_a['timestamp'] = u_tag_a['timestamp'].apply(lambda x: math.floor(x))
```

```
#Let's define our amount of users
rating_matrix = pd.read_csv('../data/user_artists.dat', sep='\t', encoding='latin-1')
num_users = len(rating_matrix.userID.unique())
```

```
#Let's normalise our weight column per user
new_rating_matrix = pd.DataFrame(columns=['userID', 'artistID', 'weight'])
for user_id in rating_matrix.userID.unique():
    user_ratings = rating_matrix[rating_matrix.userID == user_id]
    ratings = np.array(user_ratings['weight'])
    user_ratings['weight'] = tf.keras.utils.normalize(ratings, axis=-1, order=2)[0]
    new_rating_matrix = new_rating_matrix.append(user_ratings)
rating_matrix = new_rating_matrix
rating_matrix.describe()
```

```
C:\Users\seanc\AppData\Local\Temp/ipykernel_9228/1239368141.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  user_ratings['weight'] = tf.keras.utils.normalize(ratings, axis=-1, order=2)[0]
```

	weight
count	92834.000000
mean	0.091235
std	0.109804
min	0.000008
25%	0.030397
50%	0.062543
75%	0.109109
max	1.000000

```
#Let's use left merge to merge our tag data and rating matrix
rating_matrix = rating_matrix.merge(u_tag_a[['userID', 'artistID', 'tagValue', 'timestamp']],
                                    on=['userID', 'artistID'], how='left')

#Get today's timestamp
now = math.floor(time.time())

#Fill missing values as stated
rating_matrix.tagValue = rating_matrix['tagValue'].fillna('no tag')
rating_matrix.timestamp = rating_matrix['timestamp'].fillna(now)
```

```
print("Displaying Sample of new Rating Matrix")
rating_matrix[rating_matrix.tagValue != 'no tag'].sample(5)
```

```
Displaying Sample of new Rating Matrix
```

	userID	artistID	weight	tagValue	timestamp
58288	1304	646	0.375415	good	1.304615e+12
42008	927	233	0.347750	rock	1.196464e+12
20243	439	6619	0.164645	favorite	1.293836e+12
9434	203	4170	0.078750	metalcore	1.262300e+12
33323	725	9565	0.016109	conservative	1.243807e+12

The small sample above gives an indication for some of the values we can expect to find for tags. There is a large amount of distinct values in our tag data. It will be interesting to see how the recommender system interprets these.

The below pre-processing steps are as before in our other notebooks. We are correcting the scale of user and artist ID's, then ensuring their maximum values are appropriate before replacing the columns in our rating matrix.

Artist Preprocessing

To make use of the tags generally associated with artists, we will calculate their most popular tag in our data. We will use this information later on when developing our candidate model. The function in the cell below performs as previous. Essentially, it finds the most popular tag for each artist and attaches it to their profile.

We will add this extra information to our ratings matrix, as well as the artists name. Using the artist name as an identifier will make more sense to us than an ID number.

```python
#Let's match artists to genres
artists = pd.read_csv('../data/artists.dat', sep='\t', encoding='latin-1')
artists_tagged = user_tagging.merge(tags[['tagID', 'tagValue']], on='tagID')
artists_tagged = (artists_tagged.groupby('artistID')['tagValue'].apply(lambda grp:
list(grp))).reset_index()

#This function performs as previous.
for index, row in artists_tagged.iterrows():
    d = {}
    new_tags = []
    for val in row.tagValue:
        if val not in d:
            d[val] = 1
        else:
            d[val] += 1
    for key, value in d.items():
        if d[key] >=3:
            new_tags.append([key, value])
    new_tags.sort(key=lambda x:x[1], reverse=True)
    if new_tags:
        artists_tagged.at[index, "tagValue"] = [tag[0] for tag in new_tags]
        artists_tagged.at[index, 'genre'] = artists_tagged.at[index, 'tagValue'][0]

#Let's add these tags to our artists
artists.rename(columns={'id':'artistID'}, inplace=True)
artists = artists.join(artists_tagged, on='artistID', how='left', rsuffix='right')
artists.tagValue = artists.tagValue.fillna('No Tags')
artists.genre = artists.genre.fillna('No Tags')
artists.rename(columns={'tagValue': 'genres'}, inplace=True)
```

```python
#We add the extra info to our ratings matrix
rating_matrix = rating_matrix.merge(artists[['artistID', 'name', 'genre']], on='artistID')
```

```
#Extract userID column
userids = np.asarray(rating_matrix.userID)

#Remap the column
u_mapper, u_ind = np.unique(userids, return_inverse=True)

#Let's define our amount of artists
artists = pd.read_csv('../data/artists.dat', sep='\t', encoding='latin-1')
artists.rename(columns={'id':'artistID'}, inplace=True)
num_artists = len(artists.artistID.unique())

#Extract artistID column
artistids = np.asarray(rating_matrix.artistID)

#Remap the column
a_mapper, a_ind = np.unique(artistids, return_inverse=True)
```

```
# Let's replace old columns with new ind ones
rating_matrix.userID = u_ind
rating_matrix.artistID = a_ind

#Let's ensure the max value is approriate
assert(rating_matrix.userID.unique().max() == 1891)
assert(rating_matrix.artistID.unique().max() == 17631)
```

```
#We convert the ID's to string so we can use the StringLookup function later
rating_matrix.userID = rating_matrix.userID.apply(str)
rating_matrix.artistID = rating_matrix.artistID.apply(str)

rating_matrix.timestamp = rating_matrix.timestamp.apply(int)
```

```
rating_matrix.sample(5)
```

	userID	artistID	weight	tagValue	timestamp	name	genre
66177	913	3601	0.085014	dance	1167606000000	Girl Talk	alternative
10551	1328	225	0.160945	no tag	1638620584	Devendra Banhart	No Tags
60881	799	2751	0.072099	post-rock	1233442800000	Russian Circles	No Tags
41799	225	1035	0.053948	no tag	1638620584	Slipknot	j-rock
86831	1081	12514	0.039889	indie	1296514800000	Anna Calvi	No Tags

Model Development

We will perform the steps to developing a model as previous. However, we will now utilise our tags and timestamps. We do this by instantiating our interactions dictionary and including the extra features.

Instantiate Interaction Dictionary

Our interactions dictionary is just our rating matrix. It contains the following features `userID`, `artistID`, `name`, `weight`, `tag`, `genre`, and `timestamp`. We create a mapping for our dictionary below. We also create seperate individual mappings for artist ID's, tags, and genres.

```
#Let's build our interactions dictionary as previous
interactions_dict = {name: np.array(value) for name, value in rating_matrix.items()}
interactions = tf.data.Dataset.from_tensor_slices(interactions_dict)

items_dict = rating_matrix[['artistID']].drop_duplicates()
items_dict = {name: np.array(value) for name, value in items_dict.items()}
items = tf.data.Dataset.from_tensor_slices(items_dict)

names_dict = rating_matrix[['name']].drop_duplicates()
names_dict = {name: np.array(value) for name, value in names_dict.items()}
names = tf.data.Dataset.from_tensor_slices(names_dict)

tags_dict = rating_matrix[['tagValue']].drop_duplicates()
tags_dict = {name: np.array(value) for name, value in tags_dict.items()}
tags = tf.data.Dataset.from_tensor_slices(tags_dict)

genre_dict = rating_matrix[['genre']].drop_duplicates()
genre_dict = {name:np.array(value) for name, value in genre_dict.items()}
genres = tf.data.Dataset.from_tensor_slices(genre_dict)

interactions = interactions.map(lambda x: {
                                            'userID' : x['userID'],
                                            'artistID' : x['artistID'],
                                            'name' : x['name'],
                                            'weight' : float(x['weight']),
                                            'tag' : x['tagValue'],
                                            'genre': x['genre'],
                                            'timestamp': x["timestamp"],})

#artists = names.map(lambda x: x['name'])
items = items.map(lambda x: x['artistID'])
tags = tags.map(lambda x: x['tagValue'])
genres = genres.map(lambda x: x['genre'])
```

Timestamp Normalisation

As timestamps are represented as large integers, they are not healthy to use as direct input into our model. We firstly normalise our timestamps by calculaitng our minimum and maximum timestamp, then creating buckets at equal intervals between these two times. We instantiate 1000 buckets which are used to host our timestamps.

```
#Let's create bins for our timestamps
max_timestamp = interactions.map(lambda x: x["timestamp"]).reduce(
    tf.cast(0, tf.int64), tf.maximum).numpy().max()

min_timestamp = interactions.map(lambda x: x["timestamp"]).reduce(
    np.int64(1e9), tf.minimum).numpy().min()

timestamp_buckets = np.linspace( min_timestamp, max_timestamp, num=1000,)

timestamps = interactions.map(lambda x: x["timestamp"]).batch(100)
```

Lookup Tables & Training, Test Data Split

In the following cell, we define various lookup tables which we may use later on. We also shuffle our data and create testing and training batches which will be fed into our model.

```
### get unique item and user id's as a lookup table
unique_artist_ids = (np.unique(a_ind)).astype(str)
unique_user_ids = (np.unique(u_ind)).astype(str)
unique_genre_ids = np.unique(rating_matrix.genre)
unique_user_tags = np.unique(rating_matrix.tagValue)
```

```
# Randomly shuffle data and split between train and test.
tf.random.set_seed(42)
shuffled = interactions.shuffle(100_000, seed=42, reshuffle_each_iteration=False)

train = shuffled.take(62_000)
test = shuffled.skip(62_000).take(30_000)

cached_train = train.shuffle(62_000).batch(5_000)
cached_test = test.batch(2_500).cache()
```

```
print(f'our test set is: {len(train)}')
print(f'our train set is: {len(test)}')
```

```
our test set is: 62000
our train set is: 30000
```

Model Creation

The below cells host a more complex version of the model we saw previously. In our previous model, we simply instantiated our user and item embeddings as we would in a regular collaborative filtering model. In this instance, we further develop our user and item models.

User Model

In the below cell, we develop our user model. We incorporate the user ID, as well as the timestamp data. As the timestamp data signifies when a user provided a tag to an artist, it is more suitably found in the user model.

In our user model, we have included a parameter, `_use_timestamps`. When set to true, the model incorporates time stamp information. This will allow us to compare the results of the model with, or without the use of timestamps.

In our dataset, it's hard to interpret the utility of timestamps. This is because timestamp information is related to when the user-provided tags were actually applied to the artist, rather than when the user posted the tag. Also, there is an argument that including timestamps of today's date may have negative effects on the model. This model is trained on information from 2009 to 2011, essentially making it a model of that period. Timestamps from today allow the model to 'see into the future', which is obviously not a realistic trait of ML models.

```
### user model

class UserModel(tf.keras.Model):

    def __init__(self, use_timestamps):
        super().__init__()
        max_tokes=25_000

        self._use_timestamps = use_timestamps

        ## embed user id from unique_user_ids
        self.user_embedding = tf.keras.Sequential([
            tf.keras.layers.experimental.preprocessing.StringLookup(
                vocabulary=unique_user_ids),
            tf.keras.layers.Embedding(len(unique_user_ids) + 1, 32),
        ])

        ## embed timestamp
        if use_timestamps:
            self.timestamp_embedding = tf.keras.Sequential([

tf.keras.layers.experimental.preprocessing.Discretization(timestamp_buckets.tolist()),
              tf.keras.layers.Embedding(len(timestamp_buckets) + 1, 32),
            ])
            self.normalized_timestamp =
tf.keras.layers.experimental.preprocessing.Normalization(axis=None)
            self.normalized_timestamp.adapt(timestamps)

    def call(self, inputs):
        if not self._use_timestamps:
              return self.user_embedding(inputs["userID"])

        ## all features here
        return tf.concat([
            self.user_embedding(inputs["userID"]),
            self.timestamp_embedding(inputs["timestamp"]),
            tf.reshape(self.normalized_timestamp(inputs["timestamp"]), (-1, 1)),
        ], axis=1)
```

Item Model

Our item model incorporates our artist ID as before. However, it also makes use of the genre associated with the artist.

To make use of genre strings, we must first instantiate our `genre_vectorizer`. This will allow us to convert our genre string into a numerical representation. The `genre_vectorizer` is then used by our `genre_text_embedding` processing step to create an embedding of this word vector. Word embeddings allow us to measure similarity between text.

```python
### candidate model

class ItemModel(tf.keras.Model):

    def __init__(self):
        super().__init__()
        max_tokens = 10_000

        ## embed artist id from unique_artist_ids
        self.artist_embedding = tf.keras.Sequential([
            tf.keras.layers.experimental.preprocessing.StringLookup(
                vocabulary=unique_artist_ids),
            tf.keras.layers.Embedding(len(unique_artist_ids) + 1, 32),])

        ## processing text features: item genre vectorizer
        self.artist_vectorizer = tf.keras.layers.experimental.preprocessing.TextVectorization(
            max_tokens=max_tokens)

        ## we apply genre vectorizer to genres
        self.artist_text_embedding = tf.keras.Sequential([
                              self.artist_vectorizer,
                              tf.keras.layers.Embedding(max_tokens, 32, mask_zero=True),
                              tf.keras.layers.GlobalAveragePooling1D(),])

        self.artist_vectorizer.adapt(genres)

    def call(self, inputs):
        return tf.concat([
            self.artist_embedding(inputs["artistID"]),
            self.artist_text_embedding(inputs["genre"]),], axis=1)
```

Combining Models

The following cell is our parent model which we use to combine the output of both our User and Item models. We feed the outputs of each model into two dense embedding layers both of the same shape (*32*).

We then define our task (in this case FactorizedTopK), then compute the loss as we did previously.

```python
class MusicModel(tfrs.models.Model):
    def __init__(self, use_timestamps):
        super().__init__()

        ## query model is user model
        self.query_model = tf.keras.Sequential([
                         UserModel(use_timestamps),
                         tf.keras.layers.Dense(32)])

        ## candidate model is the item model
        self.candidate_model = tf.keras.Sequential([
                              ItemModel(),
                              tf.keras.layers.Dense(32)])

        ## retrieval task, choose metrics
        self.task = tfrs.tasks.Retrieval(
                    metrics=tfrs.metrics.FactorizedTopK(
                        candidates=items.batch(128).map(self.candidate_model),),)

    def compute_loss(self, features, training=False):
        # We only pass the user id and timestamp features into the query model. This
        # is to ensure that the training inputs would have the same keys as the
        # query inputs. Otherwise the discrepancy in input structure would cause an
        # error when loading the query model after saving it.

        query_embeddings = self.query_model({ "userID": features["userID"],
                                              "timestamp": features["timestamp"],
                                               "tag": features["tag"],
                                           })

        item_embeddings = self.candidate_model(features["genre"])

        return self.task(query_embeddings, item_embeddings)
```

Model Fitting and Evaluation

In the following cells, we will perform two different fitting and evaluation scenarios: (a) `_use_timestamps = True`, and (b) `_use_timestamps = False`.

```
model = MusicModel(use_timestamps=False)
model.compile(optimizer=tf.keras.optimizers.Adagrad(0.1))
model.fit(cached_train, epochs=3)
model.evaluate(cached_test, return_dict=True)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_9228/2711739428.py in <module>
----> 1 model = MusicModel(use_timestamps=False)
      2 model.compile(optimizer=tf.keras.optimizers.Adagrad(0.1))
      3 model.fit(cached_train, epochs=3)
      4 model.evaluate(cached_test, return_dict=True)

~\AppData\Local\Temp/ipykernel_9228/2758195486.py in __init__(self, use_timestamps)
     16         self.task = tfrs.tasks.Retrieval(
     17                     metrics=tfrs.metrics.FactorizedTopK(
---> 18                         candidates=items.batch(128).map(self.candidate_model),),)
     19 
     20     def compute_loss(self, features, training=False):

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\data\ops\dataset_ops.py in
map(self, map_func, num_parallel_calls, deterministic, name)
   2002         warnings.warn("The `deterministic` argument has no effect unless the "
   2003                       "`num_parallel_calls` argument is specified.")
-> 2004       return MapDataset(self, map_func, preserve_cardinality=True, name=name)
   2005     else:
   2006       return ParallelMapDataset(

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\data\ops\dataset_ops.py in
__init__(self, input_dataset, map_func, use_inter_op_parallelism, preserve_cardinality,
use_legacy_function, name)
   5453     self._use_inter_op_parallelism = use_inter_op_parallelism
   5454     self._preserve_cardinality = preserve_cardinality
-> 5455     self._map_func = StructuredFunctionWrapper(
   5456         map_func,
   5457         self._transformation_name(),

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\data\ops\dataset_ops.py in
__init__(self, func, transformation_name, dataset, input_classes, input_shapes, input_types,
input_structure, add_to_graph, use_legacy_function, defun_kwargs)
   4531         fn_factory = trace_tf_function(defun_kwargs)
   4532 
-> 4533     self._function = fn_factory()
   4534     # There is no graph to add in eager mode.
   4535     add_to_graph &= not context.executing_eagerly()

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\eager\function.py in
get_concrete_function(self, *args, **kwargs)
   3242          or `tf.Tensor` or `tf.TensorSpec`.
   3243     """
-> 3244     graph_function = self._get_concrete_function_garbage_collected(
   3245         *args, **kwargs)
   3246     graph_function._garbage_collector.release()  # pylint: disable=protected-access

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\eager\function.py in
_get_concrete_function_garbage_collected(self, *args, **kwargs)
   3208       args, kwargs = None, None
   3209     with self._lock:
-> 3210       graph_function, _ = self._maybe_define_function(args, kwargs)
   3211       seen_names = set()
   3212       captured = object_identity.ObjectIdentitySet(

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\eager\function.py in
_maybe_define_function(self, args, kwargs)
   3555 
   3556           self._function_cache.missed.add(call_context_key)
-> 3557           graph_function = self._create_graph_function(args, kwargs)
   3558           self._function_cache.primary[cache_key] = graph_function
   3559 

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\eager\function.py in
_create_graph_function(self, args, kwargs, override_flat_arg_shapes)
   3390     arg_names = base_arg_names + missing_arg_names
   3391     graph_function = ConcreteFunction(
-> 3392         func_graph_module.func_graph_from_py_func(
   3393             self._name,
   3394             self._python_function,

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\framework\func_graph.py in
func_graph_from_py_func(name, python_func, args, kwargs, signature, func_graph, autograph,
autograph_options, add_control_dependencies, arg_names, op_return_value, collections,
capture_by_value, override_flat_arg_shapes, acd_record_initial_resource_uses)
   1141         _, original_func = tf_decorator.unwrap(python_func)
   1142 
-> 1143       func_outputs = python_func(*func_args, **func_kwargs)
```

```
   1144
   1145       # invariant: `func_outputs` contains only Tensors, CompositeTensors,

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\data\ops\dataset_ops.py in
wrapped_fn(*args)
   4508           attributes=defun_kwargs)
   4509       def wrapped_fn(*args):  # pylint: disable=missing-docstring
-> 4510         ret = wrapper_helper(*args)
   4511         ret = structure.to_tensor_list(self._output_structure, ret)
   4512         return [ops.convert_to_tensor(t) for t in ret]

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\data\ops\dataset_ops.py in
wrapper_helper(*args)
   4438       if not _should_unpack(nested_args):
   4439         nested_args = (nested_args,)
-> 4440       ret = autograph.tf_convert(self._func, ag_ctx)(*nested_args)
   4441       if _should_pack(ret):
   4442         ret = tuple(ret)

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\autograph\impl\api.py in
wrapper(*args, **kwargs)
    694       try:
    695         with conversion_ctx:
--> 696           return converted_call(f, args, kwargs, options=options)
    697       except Exception as e:  # pylint:disable=broad-except
    698         if hasattr(e, 'ag_error_metadata'):

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\autograph\impl\api.py in
converted_call(f, args, kwargs, caller_fn_scope, options)
    381
    382   if not options.user_requested and conversion.is_allowlisted(f):
--> 383     return _call_unconverted(f, args, kwargs, options)
    384
    385   # internal_convert_user_code is for example turned off when issuing a dynamic

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\autograph\impl\api.py in
_call_unconverted(f, args, kwargs, options, update_cache)
    462
    463   if kwargs is not None:
--> 464     return f(*args, **kwargs)
    465   return f(*args)
    466

~\AppData\Roaming\Python\Python38\site-packages\keras\utils\traceback_utils.py in
error_handler(*args, **kwargs)
     65     except Exception as e:  # pylint: disable=broad-except
     66       filtered_tb = _process_traceback_frames(e.__traceback__)
---> 67       raise e.with_traceback(filtered_tb) from None
     68     finally:
     69       del filtered_tb

~\AppData\Roaming\Python\Python38\site-packages\tensorflow\python\autograph\impl\api.py in
wrapper(*args, **kwargs)
    697       except Exception as e:  # pylint:disable=broad-except
    698         if hasattr(e, 'ag_error_metadata'):
--> 699           raise e.ag_error_metadata.to_exception(e)
    700         else:
    701           raise

TypeError: Exception encountered when calling layer "item_model" (type ItemModel).

in user code:

    File "C:\Users\seanc\AppData\Local\Temp/ipykernel_9228/3669663950.py", line 30, in call  *
        self.artist_text_embedding(inputs["genre"]),], axis=1)

    TypeError: Only integers, slices (`:`), ellipsis (`...`), tf.newaxis (`None`) and scalar
tf.int32/tf.int64 tensors are valid indices, got 'artistID'


Call arguments received:
  • inputs=tf.Tensor(shape=(None,), dtype=string)
```

```python
model = MusicModel(use_timestamps=True)
model.compile(optimizer=tf.keras.optimizers.Adagrad(0.1))
model.fit(cached_train, epochs=3)
model.evaluate(cached_test, return_dict=True)
```

```
Epoch 1/3
WARNING:tensorflow:Layers in a Sequential model should only have a single input tensor, but we
receive a <class 'dict'> input: {'userID': <tf.Tensor 'IteratorGetNext:5' shape=(None,)
dtype=string>, 'timestamp': <tf.Tensor 'IteratorGetNext:4' shape=(None,) dtype=int64>, 'tag':
<tf.Tensor 'IteratorGetNext:3' shape=(None,) dtype=string>}
Consider rewriting this model with the Functional API.
WARNING:tensorflow:Layers in a Sequential model should only have a single input tensor, but we
receive a <class 'dict'> input: {'userID': <tf.Tensor 'IteratorGetNext:5' shape=(None,)
dtype=string>, 'timestamp': <tf.Tensor 'IteratorGetNext:4' shape=(None,) dtype=int64>, 'tag':
<tf.Tensor 'IteratorGetNext:3' shape=(None,) dtype=string>}
Consider rewriting this model with the Functional API.
13/13 [==============================] - 62s 4s/step -
factorized_top_k/top_1_categorical_accuracy: 0.2838 -
factorized_top_k/top_5_categorical_accuracy: 0.3035 -
factorized_top_k/top_10_categorical_accuracy: 0.3124 -
factorized_top_k/top_50_categorical_accuracy: 0.3349 -
factorized_top_k/top_100_categorical_accuracy: 0.3480 - loss: 39029.6232 - regularization_loss:
0.0000e+00 - total_loss: 39029.6232
Epoch 2/3
13/13 [==============================] - 60s 4s/step -
factorized_top_k/top_1_categorical_accuracy: 0.3954 -
factorized_top_k/top_5_categorical_accuracy: 0.4175 -
factorized_top_k/top_10_categorical_accuracy: 0.4273 -
factorized_top_k/top_50_categorical_accuracy: 0.4529 -
factorized_top_k/top_100_categorical_accuracy: 0.4650 - loss: 38513.8764 - regularization_loss:
0.0000e+00 - total_loss: 38513.8764
Epoch 3/3
13/13 [==============================] - 63s 5s/step -
factorized_top_k/top_1_categorical_accuracy: 0.5219 -
factorized_top_k/top_5_categorical_accuracy: 0.5371 -
factorized_top_k/top_10_categorical_accuracy: 0.5441 -
factorized_top_k/top_50_categorical_accuracy: 0.5613 -
factorized_top_k/top_100_categorical_accuracy: 0.5698 - loss: 38298.1392 - regularization_loss:
0.0000e+00 - total_loss: 38298.1392
WARNING:tensorflow:Layers in a Sequential model should only have a single input tensor, but we
receive a <class 'dict'> input: {'userID': <tf.Tensor 'IteratorGetNext:5' shape=(None,)
dtype=string>, 'timestamp': <tf.Tensor 'IteratorGetNext:4' shape=(None,) dtype=int64>, 'tag':
<tf.Tensor 'IteratorGetNext:3' shape=(None,) dtype=string>}
Consider rewriting this model with the Functional API.
12/12 [==============================] - 30s 2s/step -
factorized_top_k/top_1_categorical_accuracy: 0.4899 -
factorized_top_k/top_5_categorical_accuracy: 0.5035 -
factorized_top_k/top_10_categorical_accuracy: 0.5103 -
factorized_top_k/top_50_categorical_accuracy: 0.5257 -
factorized_top_k/top_100_categorical_accuracy: 0.5324 - loss: 19461.5293 - regularization_loss:
0.0000e+00 - total_loss: 19461.5293
```

```
{'factorized_top_k/top_1_categorical_accuracy': 0.4898666739463806,
 'factorized_top_k/top_5_categorical_accuracy': 0.5034999847412109,
 'factorized_top_k/top_10_categorical_accuracy': 0.5102666616439819,
 'factorized_top_k/top_50_categorical_accuracy': 0.5256999731063843,
 'factorized_top_k/top_100_categorical_accuracy': 0.5324333310127258,
 'loss': 19464.4296875,
 'regularization_loss': 0,
 'total_loss': 19464.4296875}
```

Output Analysis

From the above output, it's clear that timestamps do not improve recommendations for this model. This is likely due to the fact that a substantial amount of the timestamps are of today's date which is throwing off the model. Perhaps a better data imputation would have been the median date observed in the data.

Otherwise both models seem to have reasonable performance with a positive item being returned as the top candidate 50% of the time. These models will supply a basis for our next advanced deep retrieval model.

Conclusions

In this notebook, we experimented with building a more complex deep learning framework by enhancing the input used in our query and item towers. Leveraging context features such as timestamps and text data can lead to better model performance and higher quality recommendations being produced.

However, we learned that proper imputation of data is an important aspect of data quality. Poorly imputated data can lead to contextual features having a negative effect on retrieval. In our case, it does not make sense to use today's timestamp in our data, as this allows the model to essentially 'see into the future'. This is an unrealisticquality of our model.

Although being more complex than our previos models, it still remains in its imfancy, and TFRS offers many directions to expand our model further. We will leave this for future work.

Conclusions

The following research aimed to roughly follow the outline of previous innovation and advances in recommender systems while simultaneously producing varying recommender models built on top of the HETREC 2011 [Cantador *et al.*, 2011] 2nd International Workshop on Information Heterogeneity and Fusion in Recommender Systems Last.FM dataset.

We began by incorporating a Matrix Factorisation model, which calculates the product between learned user and item embeddings to produce a score for that particular user-item combination. This type of model proved very intuitive, but lacks in some predictive power and tends to be biased towards recommending popular items.

We then moved on to developing deep learning recommender systems using Keras on top of Tensorflow, specifically using their TFRS package. This package makes development of recommender systems an easy and fluid process. We only scratched the surface of what is capable with this package. In the future, it would be interesting to explore the more complex offerings of TFRS with this relatively simple dataset.

Bibliography

[And06]
Chris Anderson. *The long tail: Why the future of business is selling less of more*. Hachette Books, 2006.

[BalabanovicS97]
Marko Balabanović and Yoav Shoham. Fab: content-based, collaborative recommendation. *Commun. ACM*, 40(3):66–72, mar 1997. URL: https://doi.org/10.1145/245108.245124, doi:10.1145/245108.245124.

[CBK11]
Iván Cantador, Peter Brusilovsky, and Tsvi Kuflik. Second workshop on information heterogeneity and fusion in recommender systems (hetrec2011). In *Proceedings of the fifth ACM conference on Recommender systems*, 387–388. 2011.

[CKH+16]
Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, 7–10. Association for Computing Machinery, Sep 2016. URL: https://doi.org/10.1145/2988450.2988454, doi:10.1145/2988450.2988454.

[CZ07]
Andrzej Cichocki and Rafal Zdunek. Regularized alternating least squares algorithms for non-negative matrix/tensor factorization. In Derong Liu, Shumin Fei, Zengguang Hou, Huaguang Zhang, and Changyin Sun, editors, *Advances in Neural Networks – ISNN 2007*, 793–802. Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[CAS16]
Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, 191–198. Association for Computing Machinery, Sep 2016. URL: https://doi.org/10.1145/2959100.2959190, doi:10.1145/2959100.2959190.

[ERK11]
Michael D Ekstrand, John T Riedl, and Joseph A Konstan. *Collaborative filtering recommender systems*. Now Publishers Inc, 2011.

[ERR16]
Mehdi Elahi, Francesco Ricci, and Neil Rubens. A survey of active learning in collaborative filtering recommender systems. *Computer Science Review*, 20:29–50, 2016. doi:https://doi.org/10.1016/j.cosrev.2016.05.002.

[GNHS11]
Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, 69–77. New York, NY, USA, 2011. Association for Computing Machinery. URL: https://doi.org/10.1145/2020408.2020426, doi:10.1145/2020408.2020426.

[GMM98]
Robert H Guttman, Alexandros G Moukas, and Pattie Maes. Agent-mediated electronic commerce: a survey. *The Knowledge Engineering Review*, 13(2):147–159, 1998.

[KBV09]
Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, Aug 2009. doi:10.1109/MC.2009.263.

[LSPU16]
Ayangleima Laishram, Satya Prakash Sahu, Vineet Padmanabhan, and Siba Kumar Udgata. Collaborative filtering, matrix factorization and population based search: the nexus unveiled. In *International Conference on Neural Information Processing*, 352–361. Springer, 2016.

[Paz99]
Michael J Pazzani. A framework for collaborative, content-based and demographic filtering. *Artificial intelligence review*, 13(5):393–408, 1999.

[SK09]
Xiaoyuan Su and Taghi M Khoshgoftaar. A survey of collaborative filtering techniques. *Advances in artificial intelligence*, 2009.

[TGB15]
Poonam B Thorat, RM Goudar, and Sunita Barve. Survey on collaborative filtering, content-based filtering and hybrid recommendation system. *International Journal of Computer Applications*, 110(4):31–36, 2015.

[ZYST19]
Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: a survey and new perspectives. *ACM Computing Surveys*, 52(1):5:1–5:38, Feb 2019. doi:10.1145/3285029.
