Contents

Recommender Systems Assignment - Michael Mc Cahill

In the world we live in recommender systems are everywhere. They come in many shapes and sizes. All tech companies are known to use recommender systems in one way or another. Spotify use recommender systems to provide song recomendations for users, Amazon use recommender systems to give shopping recommendations for shopper and Youtube use recommender systems to give users suggestions on what to watch next. The list is endless. A recommendation system helps users find compelling content in a large corpora “1”. These systems use Machine Learning operations in order to provide meaningful recommendations to users.Recommender Systems have numerous benefits including providing personalization for customers of e-commerce and promoting one-to-one marketing just to name a few. Having a recommendation engine makes browsing content easier. Plus, a great recommendation system helps users find things they wouldn’t have thought to look for on their own.

Note

40% of app installs on Google Play come from recommendations. 60% of watch time on YouTube comes from recommendations.

Dataset Description

A full description of the data can be found here

Data Set name	Description
artists.dat	This file contains information about music artists listened and tagged by the users.
tags.dat	This file contains the set of tags available in the dataset.
user.artists.dat	This file contains the artists listened by each user.
user.taggedartists.dat	These files contain the tag assignments of artists provided by each particular user.
user.friends.dat	These files contain the friend relations between users in the database.

This paper will cover many of the sections based on the work done in Recommendation Systems Colab. This system is used to best recommend movies for users based on users historical ratings. Our system will be similar but will be adapted for the LastFM music data.

This report will contain 4 sections of work. These sections are:

Section 1: Data Preparation and Analysis **Section 2: Recommender System Building and Tuning ** **Section 3: Cluster Analysis Section 4:Personal Spotify Adapatiation**

The Data Preparation and Analysis will have information on preparing and analysing our data. The various data cleaning aspects that were undertaken as part of the report and also some interesting analysis of our data.

Section 2 we will be building our recommender system. There are many section as part of this section including, building and training our model. We will also analyise our recommendations in this system.

As part of Section 3 we look to see if our clusters can enhance our learning of the embeddings of our model.

Finally in section 4, we use data from my own personal spotify in order to test our recommender system.

Data Preparation and Exploration

In the data preparation and exploration section we aim to do a few things

- Clean the data, this includes checking for null values
- Explore distribution of genres

- Look at the most popular artists

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

The first step in the data preparation phase is to read in our data. The data being used is the Last.FM data set. The data for this project can be found here

```
# Reading in data from Last.FM
user_artists = pd.read_csv('data/user_artists.dat', sep='\t', encoding='latin-1')
artists = pd.read_csv('data/artists.dat', sep='\t', encoding='latin-1')
tags = pd.read_csv('data/tags.dat', sep='\t', encoding='latin-1')
user_friends = pd.read_csv('data/user_friends.dat', sep='\t', encoding='latin-1')
user_taggedartists = pd.read_csv('data/user_taggedartists.dat', sep='\t',
encoding='latin-1')
```

Now we check the data frames for null values

```
# A for loop to iterate through our data sets to detect null values
df_list = [user_artists, artists, tags, user_friends, user_taggedartists]
df_list_names = ["user_artists", "artists", "tags", "user_friends",
"user_taggedartists"]

for i, n in zip(df_list, df_list_names):
    null_status = i.isnull().values.any()
    if null_status:
        print(n + " has null values")
    else:
        print(n + " doesn't have null values")
```

```
user_artists doesn't have null values
artists has null values
tags doesn't have null values
user_friends doesn't have null values
user_taggedartists doesn't have null values
```

From running this quick query we can see that only the artists data frame contains null values. Lets explore this further

```
# Investigating null values in artists data frame
artists.isna().sum()
```

```
id              0
name            0
url             0
pictureURL    444
dtype: int64
```

There are 444 null values in pictureURL column in this data frame. We can drop this column as it will not be an important feature of our recommender system.

```
# This is to show number of columns before deletion
artists.shape
```

```
(17632, 4)
```

```
# Dropping pictureURL column
artists = artists.drop(columns=['pictureURL'])
artists.shape
```

```
(17632, 3)
```

Data Analysis

In this section we will be trying to get a greater understanding of the data. We will investigate, the most popular genres and the most popular artists

First we will look at distribution of artists

The user_artists data frame contains the artists listened by each user, by using this data frame we will be able to identify the most listened to artists

```python
# Most popular artists analysis
artist_count = user_artists["artistID"].value_counts().reset_index()
artist_count
# value_counts() provides us with the amount of times an artist has been listened to by
a unique user
```

	index	artistID
0	89	611
1	289	522
2	288	484
3	227	480
4	300	473
...	...	...
17627	9544	1
17628	9546	1
17629	9548	1
17630	9549	1
17631	18730	1

17632 rows × 2 columns

```python
# Rename artistID column name to Total_user_listens
artist_count = artist_count.rename(columns={"artistID" : "Total_user_listens"})

# Rename Index to artistid
artist_count = artist_count.rename(columns={"index" : "id"})
```

Now we can use the artistID in the artist_count dataframe to find the corresponding artist name in the artists data frame

```python
# Find corresponding artist name
artist_name_counts = artist_count.merge(artists, on='id')
artist_name_counts.head()
```

	id	Total_user_listens	name	url
0	89	611	Lady Gaga	http://www.last.fm/music/Lady+Gaga
1	289	522	Britney Spears	http://www.last.fm/music/Britney+Spears
2	288	484	Rihanna	http://www.last.fm/music/Rihanna
3	227	480	The Beatles	http://www.last.fm/music/The+Beatles
4	300	473	Katy Perry	http://www.last.fm/music/Katy+Perry

```python
# Exporting artist_name_counts to csv file for analysis in a future section
artist_name_counts.to_csv('clean_data/artist_name_counts.csv', index=False)
```

```python
# Show artists with over 100 user listens
artist_name_counts.loc[artist_name_counts['Total_user_listens'] > 100]
```

	id	Total_user_listens	name	url
0	89	611	Lady Gaga	http://www.last.fm/music/Lady+Gaga
1	289	522	Britney Spears	http://www.last.fm/music/Britney+Spears
2	288	484	Rihanna	http://www.last.fm/music/Rihanna
3	227	480	The Beatles	http://www.last.fm/music/The+Beatles
4	300	473	Katy Perry	http://www.last.fm/music/Katy+Perry
...	...	...	...	...
120	543	105	Nicole Scherzinger	http://www.last.fm/music/Nicole+Scherzinger
121	918	104	Megadeth	http://www.last.fm/music/Megadeth
122	1513	104	Ramones	http://www.last.fm/music/Ramones
123	907	104	Timbaland	http://www.last.fm/music/Timbaland
124	930	103	Nightwish	http://www.last.fm/music/Nightwish

125 rows × 4 columns

```
# Find sum of user listens top 100 artists
artist_over_100 = artist_name_counts.loc[artist_name_counts['Total_user_listens'] > 100]
artist_over_100["Total_user_listens"].sum()
#artist_name_counts["Total_user_listens"].sum()
```

```
24809
```

```
# Show sum of all user listens
artist_name_counts["Total_user_listens"].sum()
```

```
92834
```

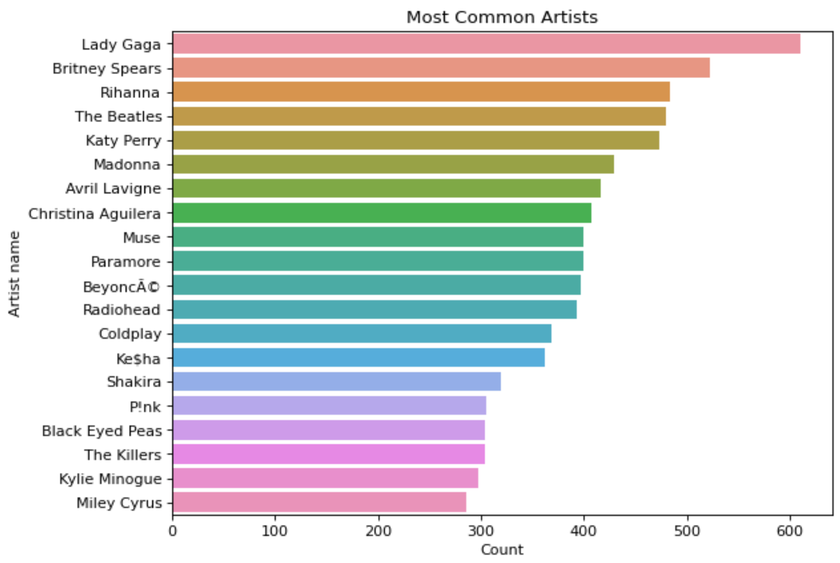
There are very few artists in the data frame that have had over 100 users listen to them

Out of 17632 artists, only 125 artists have had over 100 user listens. This is only 0.71% of the data frame

Although only a small proportion of artists have over 100 user listens, they account for 26.7% of the total user listens in this data frame. Over a quarter of the user listens are represented in the 125 artists that have over 100 user listens.

Visual representation of the top 10 artists

```
# Plot the 10 most listened to artists
from matplotlib.pyplot import figure
plot_artists = artist_name_counts.head(20)

figure(figsize=(8, 6), dpi=80)


sns.barplot(x="Total_user_listens", y="name", data=plot_artists)

plt.title('Most Common Artists')
plt.ylabel('Artist name')
plt.xlabel('Count')
plt.show()
```



From the graph above it can be see that lady gaga is has the most user listens in the data set

The same approach will be followed for identifying the most popular genres as was used for finding the most popular artists

```
# Most popular genre analysis
tag_counts = user_taggedartists["tagID"].value_counts().reset_index()
tag_counts.head()
```

	index	tagID
0	73	7503
1	24	5418
2	79	5251
3	18	4672
4	81	4458

```
# Rename index column name to tagID and rename tagID to Count_Artist_Genre
tag_counts = tag_counts.rename(columns={"index" : "tagID", "tagID" :
"Count_Artist_Genre"})
tag_counts.head()
```

	tagID	Count_Artist_Genre
0	73	7503
1	24	5418
2	79	5251
3	18	4672
4	81	4458

```
# Find corresponding artist name
tags_name_counts = tag_counts.merge(tags, on='tagID')
tags_name_counts.head()
```

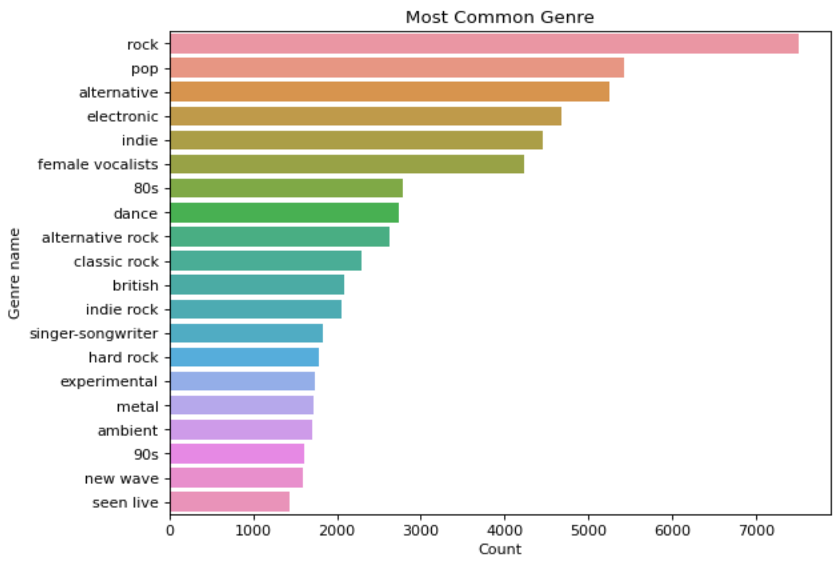
	tagID	Count_Artist_Genre	tagValue
0	73	7503	rock
1	24	5418	pop
2	79	5251	alternative
3	18	4672	electronic
4	81	4458	indie

```
# Renaming tagValue to Genre
tags_name_counts = tags_name_counts.rename(columns={"tagValue" : "Genre"})
tags_name_counts.head(1)
```

	tagID	Count_Artist_Genre	Genre
0	73	7503	rock

```
# Export the tags_name_counts for use in the clustering segment of this report
tags_name_counts.to_csv('clean_data/tags_name_counts.csv', index=False)
```

```
# Plot the 10 most popular to genres
from matplotlib.pyplot import figure
plot_tags = tags_name_counts.head(20)

figure(figsize=(8, 6), dpi=80)


sns.barplot(x="Count_Artist_Genre", y="Genre", data=plot_tags)

plt.title('Most Common Genre')
plt.ylabel('Genre name')
plt.xlabel('Count')
plt.show()
```



Final data preparation steps

After performing some simple data cleaning and data analysis, one final step is needed in this setion. We will reindex the IDs in our data frames. By reindexing these IDs now it will make things simpler for us going forward in this project

```
# Reindex user_artists
user_artists["userID"] = user_artists["userID"].apply(lambda x: str(x-2))
user_artists["artistID"] = user_artists["artistID"].apply(lambda x: str(x-1))

# Reindex artists
artists["id"] = artists["id"].apply(lambda x: str(x-1))

# Reindex tags
tags["tagID"] = tags["tagID"].apply(lambda x: str(x-1))

# Reindex user_friends
user_friends["userID"] = user_friends["userID"].apply(lambda x: str(x-2))
user_friends["friendID"] = user_friends["friendID"].apply(lambda x: str(x-2))

# Reindex user_taggedartists
user_taggedartists["userID"] = user_taggedartists["userID"].apply(lambda x: str(x-2))
user_taggedartists["artistID"] = user_taggedartists["artistID"].apply(lambda x: str(x-
1))
user_taggedartists["tagID"] = user_taggedartists["tagID"].apply(lambda x: str(x-1))
```

Different kind of re-indexing were performed in this section due to problems that were faced during the building of the recommender system. Problems were being encountered due to index being out of range so further cleaning of the data will be performed

```python
# Converting userID and artistID to numbers as they are currently object values and
reset index to start at 0
userids = np.asarray(user_artists.userID)
u_mapper, u_ind = np.unique(userids, return_inverse=True)

artistids = np.asarray(user_artists.artistID)
a_mapper, a_ind = np.unique(artistids, return_inverse=True)

user_artists["userID"] = u_ind
user_artists["artistID"] = a_ind
```

After performing the reindexing of IDs, we will now look at the other columns to see if there are any other discrepancies in our data

```python
# Generating summary statistics to discover potential discrepancies in user_artists
user_artists.describe()
```

	userID	artistID	weight
count	92834.000000	92834.000000	92834.00000
mean	947.082911	9967.995314	745.24393
std	546.354069	4416.483620	3751.32208
min	0.000000	0.000000	1.00000
25%	473.000000	7791.000000	107.00000
50%	950.000000	10305.500000	260.00000
75%	1419.000000	12989.000000	614.00000
max	1891.000000	17631.000000	352698.00000

```python
# Generating summary statistics to discover potential discrepancies in artists
artists.describe()
```

	id	name	url
count	17632	17632	17632
unique	17632	17632	17632
top	0	MALICE MIZER	http://www.last.fm/music/MALICE+MIZER
freq	1	1	1

```python
# Generating summary statistics to discover potential discrepancies in tags
tags.describe()
```

	tagID	tagValue
count	11946	11946
unique	11946	11946
top	0	metal
freq	1	1

```python
# Generating summary statistics to discover potential discrepancies in user_friends
user_friends.describe()
```

	userID	friendID
count	25434	25434
unique	1892	1892
top	1541	1541
freq	119	119

```
# Generating summary statistics to discover potential discrepancies in
user_taggedartists
user_taggedartists.describe()
```

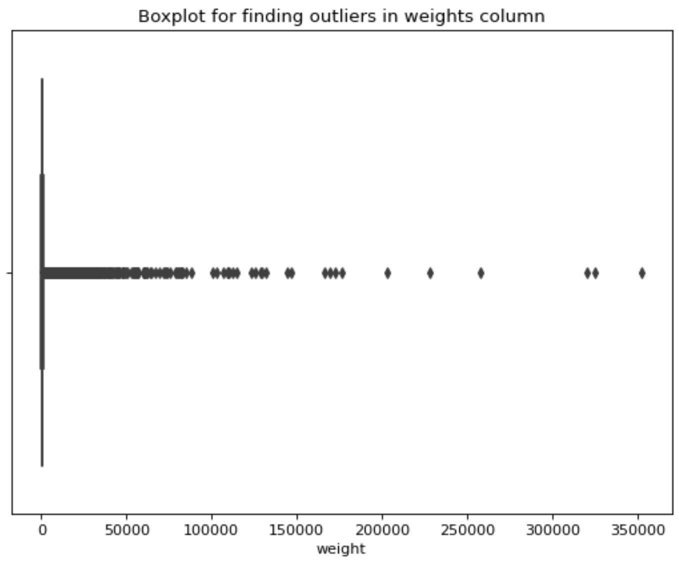
	day	month	year
count	186479.000000	186479.000000	186479.000000
mean	1.095566	6.524215	2008.809791
std	0.712813	3.486855	1.410062
min	1.000000	1.000000	1956.000000
25%	1.000000	3.000000	2008.000000
50%	1.000000	7.000000	2009.000000
75%	1.000000	10.000000	2010.000000
max	9.000000	12.000000	2011.000000

From generating summary statists using describe() as seen above, there is only one column that sticks out to me. That column is the weights column in the user_artists data frame. We will investigate this column further to develop a better understanding for it

The weights column in ther user_artists data frame represents the amount of times a user has listened to a particular artist. It seems unlikely that a user would listen to an artist 352,698 times but we will investigate further to see if there is an answer to this problem

```
# Creating a box plot for weights column to find outliers
figure(figsize=(8, 6), dpi=80)
sns.boxplot(data=user_artists, x="weight")
plt.title("Boxplot for finding outliers in weights column")
```

```
Text(0.5, 1.0, 'Boxplot for finding outliers in weights column')
```



From analysing this box plot it is clear to see that there are some large outliers. As a result of this we will need to perform some sort of standardisation on the data to prevent potential errors in the future

The ranges for the weight column vary greatly. The minimum weights value is one with the maximum value being 352,698. If we were to leave this our recommendations could be thrown off significantly

For normalising the weights volumn, we will use [Keras normalisation](). Keras is a tensorflow package used for machine learning. It is used for to standaraize the inputs for a deep neural network. By using keras we can also reduce to number of epochs needed during the training phase of this project

```
# Performing Keras normalisation on weights column
'''
import tensorflow.compat.v1 as tf
user_artists.weight = tf.keras.utils.normalize(np.asarray(user_artists.weight), order=2)
[0]
'''
```

```
'\nimport tensorflow.compat.v1 as tf\nuser_artists.weight =
tf.keras.utils.normalize(np.asarray(user_artists.weight), order=2)[0]\n'
```

This was a good thought at the time of normalising. When it came to my system I felt as if the recommendations were very bad and didn't give a good recommendation for users. Another appoach will be used in order to fix the weights issue. The weights column is probably the most important column when it comes to the recommendations so we must ensure we do a more approiate form of normalisation.

```
'''
user_artists["weight"]=(user_artists["weight"]-
user_artists["weight"].min())/(user_artists["weight"].max()-
user_artists["weight"].min())
'''
```

```
'\nuser_artists["weight"]=(user_artists["weight"]-
user_artists["weight"].min())/(user_artists["weight"].max()-
user_artists["weight"].min())\n'
```

The weights have now been normalised using a min-max scalar. The new weight values range between 0 and 1

Min-Max Scalar is not an appropriate method to perform on our data. It doesn't give any more insightful findings compared to Keras normalisation. For example in our recommender when we look at Lady Gaga who had the most user listens in the data, one would expect another popular artist to be recommended. Any of the suggestions given for Lady Gaga I had never heard of before. For both DOT and COSINE the nearest neighbour was Secret Lives of the Freemasons. I had to google who they were. They were a punk rock band. This is a clear indicator that the recommendation system that has been built isn't good enough.

Rather then normalising or standardising our data it might be best to scale our data instead

Due to the massive outliers in the weights data it will be beneficial to manually set an upper bound limit to the weight column. I will set this value to 614 as it is the value for the 75% in the column

```
user_artists_weights_outlier = user_artists[user_artists["weight"] > 614]
user_artists_weights_outlier.describe()
```

	userID	artistID	weight
count	23182.000000	23182.000000	23182.000000
mean	940.614442	10211.632085	2352.778923
std	545.403059	4205.498502	7268.661759
min	0.000000	1.000000	615.000000
25%	465.000000	8919.000000	813.000000
50%	953.000000	10339.000000	1161.000000
75%	1413.000000	13001.000000	1984.750000
max	1891.000000	17625.000000	352698.000000

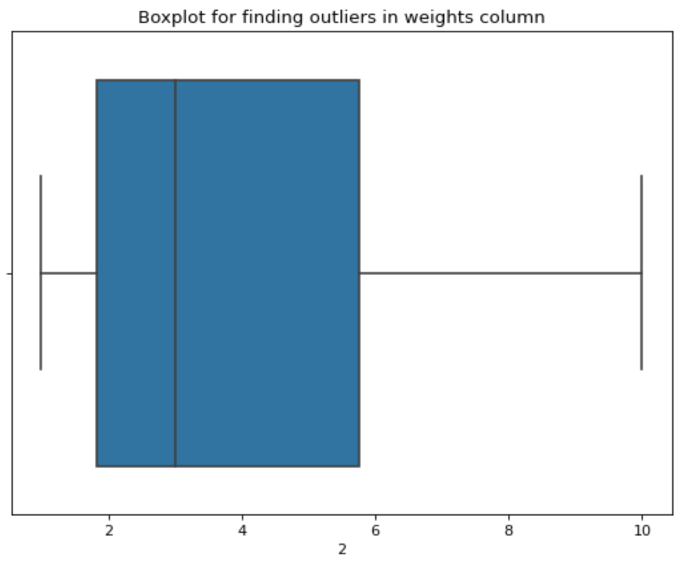
```
# Manually setting all values above 614 to be equal to 1161 as it is the median of our
outliers
user_artists['weight'].values[user_artists['weight'].values > 1161] = 1161
```

After we have adjusted this number we can now scale our data between 1 and 10

```
# Using Min Max scalar to scale features to a given range (1,10)
from sklearn import preprocessing
minmax_scale = preprocessing.MinMaxScaler(feature_range=(1,10))

x_scale = pd.DataFrame(minmax_scale.fit_transform(user_artists))

# Set new scaled value to be the new weight column
user_artists["weight"] = x_scale[2]
```

```
# Creating a box plot for weights column to find outliers
figure(figsize=(8, 6), dpi=80)
sns.boxplot(data=x_scale, x=2)
plt.title("Boxplot for finding outliers in weights column")
```

```
Text(0.5, 1.0, 'Boxplot for finding outliers in weights column')
```



```
# See our new and improved weights column
user_artists["weight"].describe()
```

```
count    92834.000000
mean         4.154168
std          2.964601
min          1.000000
25%          1.822414
50%          3.009483
75%          5.756034
max         10.000000
Name: weight, dtype: float64
```

After our analysis and preparation we are able to export our data for use in the next part of the project

```
# Export Clean data to Clean data folder


user_artists.to_csv('clean_data/user_artists.csv', index=False)
artists.to_csv('clean_data/artists.csv', index=False)
tags.to_csv('clean_data/tags.csv', index=False)
user_friends.to_csv('clean_data/user_friends.csv', index=False)
user_taggedartists.to_csv('clean_data/user_taggedartists.csv', index=False)
```

Conclusion

From performing data analyis on artists and genres I have come to some conclusions. With regards to the artists, even though there is only a small representation with artists with over 100 user listens, they account for over a quarter of the entire user listens. With the genres, it is no surpise really that artists like pop and rock are the most popular. These genres are very broad and can be brokem up into sub categories but they will remain the same for now.

By performing these simple data analytics procedures we can better understand the data we are working with. The genres and artists that people listen to might be very useful when providing recommendations in the future.

Building our Recommender System

The work done in this section is based off the movieLens recommender system adapted to use for LastFM dataset. The original work an be found here

In this setion we will be building a TensorFlow.SparseTensor represntation of our weights matrix, we will be calculating our Mean Squared error, training a matrix factorisation model, creating a CFModel helper class and inspecting our embeddings.

Firstly we will have to read in our modules and just for our sake add some convenience pandas functions

```python
# @title Imports (run this cell)
from __future__ import print_function

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

from sklearn.manifold import TSNE
from sklearn.decomposition import PCA

# 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

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

```
WARNING:tensorflow:From /home/michael/anaconda3/lib/python3.7/site-
packages/tensorflow/python/compat/v2_compat.py:111: disable_resource_variables (from
tensorflow.python.ops.variable_scope) is deprecated and will be removed in a future
version.
Instructions for updating:
non-resource variables are not supported in the long term
```

```
Installing Altair...
```

```
Collecting git+git://github.com/altair-viz/altair.git
  Cloning git://github.com/altair-viz/altair.git to /tmp/pip-req-build-no_tocll
  Running command git clone --filter=blob:none -q git://github.com/altair-
viz/altair.git /tmp/pip-req-build-no_tocll
```

```
  Resolved git://github.com/altair-viz/altair.git to commit
a987d04e276106f62d4247ea48a1fcead2d06636
```

```
  Installing build dependencies ... ?25l-
```

```
 \
```

```
 |
```

```
 done
```

```
?25h  Getting requirements to build wheel ... ?25l-
```

```
 done
```

```
?25h  Preparing metadata (pyproject.toml) ... ?25l-
```

```
 done
?25hRequirement already satisfied: numpy in /home/michael/anaconda3/lib/python3.7/site-
packages (from altair==4.2.0.dev0) (1.19.5)
Requirement already satisfied: jinja2 in /home/michael/anaconda3/lib/python3.7/site-
packages (from altair==4.2.0.dev0) (2.11.2)
Requirement already satisfied: entrypoints in
/home/michael/anaconda3/lib/python3.7/site-packages (from altair==4.2.0.dev0) (0.3)
Requirement already satisfied: toolz in /home/michael/anaconda3/lib/python3.7/site-
packages (from altair==4.2.0.dev0) (0.11.1)
Requirement already satisfied: pandas>=0.18 in
/home/michael/anaconda3/lib/python3.7/site-packages (from altair==4.2.0.dev0) (1.3.4)
Requirement already satisfied: jsonschema<4.0,>=3.0 in
/home/michael/anaconda3/lib/python3.7/site-packages (from altair==4.2.0.dev0) (3.2.0)
```

```
Requirement already satisfied: setuptools in
/home/michael/anaconda3/lib/python3.7/site-packages (from jsonschema<4.0,>=3.0-
>altair==4.2.0.dev0) (49.6.0.post20200925)
Requirement already satisfied: importlib-metadata in
/home/michael/anaconda3/lib/python3.7/site-packages (from jsonschema<4.0,>=3.0-
>altair==4.2.0.dev0) (1.7.0)
Requirement already satisfied: pyrsistent>=0.14.0 in
/home/michael/anaconda3/lib/python3.7/site-packages (from jsonschema<4.0,>=3.0-
>altair==4.2.0.dev0) (0.17.3)
Requirement already satisfied: six>=1.11.0 in
/home/michael/anaconda3/lib/python3.7/site-packages (from jsonschema<4.0,>=3.0-
>altair==4.2.0.dev0) (1.15.0)
Requirement already satisfied: attrs>=17.4.0 in
/home/michael/anaconda3/lib/python3.7/site-packages (from jsonschema<4.0,>=3.0-
>altair==4.2.0.dev0) (21.2.0)
Requirement already satisfied: pytz>=2017.3 in
/home/michael/anaconda3/lib/python3.7/site-packages (from pandas>=0.18-
>altair==4.2.0.dev0) (2020.1)
Requirement already satisfied: python-dateutil>=2.7.3 in
/home/michael/anaconda3/lib/python3.7/site-packages (from pandas>=0.18-
>altair==4.2.0.dev0) (2.8.1)
Requirement already satisfied: MarkupSafe>=0.23 in
/home/michael/anaconda3/lib/python3.7/site-packages (from jinja2->altair==4.2.0.dev0)
(1.1.1)
```

```
Requirement already satisfied: zipp>=0.5 in /home/michael/anaconda3/lib/python3.7/site-
packages (from importlib-metadata->jsonschema<4.0,>=3.0->altair==4.2.0.dev0) (3.1.0)
```

```
Done installing Altair.
```

Read in clean data that we prepared earlier

```
# Reading in data from Last.FM clean data
user_artists = pd.read_csv('clean_data/user_artists.csv')
artists = pd.read_csv('clean_data/artists.csv')
tags = pd.read_csv('clean_data/tags.csv')
user_friends = pd.read_csv('clean_data/user_friends.csv')
user_taggedartists = pd.read_csv('clean_data/user_taggedartists.csv')
artist_name_counts = pd.read_csv('clean_data/artist_name_counts.csv')
```

Before we start to build our system we need to reset the index of our columns to revent errors in the future

Building a Sparse Representation Model

In this part we will be factorising our weights matrix into a product of user embeddings and matrix embeddings. We do this because our weights matrix could could be very large, so some users will only listen to a small subset of artists. For effecient representation we will be using tf.SparseTensor to represent our matrix.

```
"""
  Args:
    ratings_df: a pd.DataFrame with `user_id`, `artist_id` and `weights` columns.
  Returns:
    a tf.SparseTensor representing the weights matrix.
"""

def build_rating_sparse_tensor(user_artists_df):

  indices = user_artists_df[['userID', 'artistID']].values
  values = user_artists_df['weight'].values

  return tf.SparseTensor(
      indices=indices,
      values=values,
      dense_shape=[len(user_artists.userID.unique()), artists.shape[0]])
```

Mean Squared Error

Mean Square Error will be used to measure an approximate error with our recommender model It is defined in the formula: MSE = $\frac{1}{n}\Sigma_{i=1}^{n}(y-\hat{y})^2$

```
"""
  Args:
    sparse_ratings: A SparseTensor rating matrix, of dense_shape [N, M]
    user_embeddings: A dense Tensor U of shape [N, k] where k is the embedding
      dimension, such that U_i is the embedding of user i.
    artist_embeddings: A dense Tensor V of shape [M, k] where k is the embedding
      dimension, such that V_j is the embedding of artist j.
  Returns:
    A scalar Tensor representing the MSE between the true ratings and the
      model's predictions.
"""

def sparse_mean_square_error(sparse_ratings, user_embeddings, music_embeddings):

  predictions = tf.reduce_sum(
      tf.gather(user_embeddings, sparse_ratings.indices[:, 0]) *
      tf.gather(music_embeddings, sparse_ratings.indices[:, 1]),
      axis=1)
  loss = tf.losses.mean_squared_error(sparse_ratings.values, predictions)
  return loss
```

CFModel Helper

This is a simple class to train a matrix factorization model using stochastic gradient descent.

Collaborative filtering: Uses similarities between queries and items simultaneously to provide recommendations. If user A is similar to user B, and user B likes video 1, then the system can recommend video 1 to user A (even if user A hasn't seen any videos similar to video 1).

```
class CFModel(object):
  """Simple class that represents a collaborative filtering model"""
  def __init__(self, embedding_vars, loss, metrics=None):
    """Initializes a CFModel.
    Args:
      embedding_vars: A dictionary of tf.Variables.
      loss: A float Tensor. The loss to optimize.
      metrics: optional list of dictionaries of Tensors. The metrics in each
        dictionary will be plotted in a separate figure during training.
    """
    self._embedding_vars = embedding_vars
    self._loss = loss
    self._metrics = metrics
    self._embeddings = {k: None for k in embedding_vars}
    self._session = None

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

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

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

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

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

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

Building the model

Using your sparse_mean_square_error function, write a function that builds a CFModel by creating the embedding variables and the train and test losses.

```
# Splitting our data into test and train dataframes

def split_dataframe(df, holdout_fraction=0.1):

  test = df.sample(frac=holdout_fraction, replace=False)
  train = df[~df.index.isin(test.index)]
  return train, test
```

```
"""
  Args:
    ratings: a DataFrame of the weights
    embedding_dim: the dimension of the embedding vectors.
    init_stddev: float, the standard deviation of the random initial embeddings.
  Returns:
    model: a CFModel.
"""

def build_model(ratings, embedding_dim=3, init_stddev=1.):

  # Split the ratings DataFrame into train and test.
  train_ratings, test_ratings = split_dataframe(ratings)
  # SparseTensor representation of the train and test datasets.
  A_train = build_rating_sparse_tensor(train_ratings)
  A_test = build_rating_sparse_tensor(test_ratings)
  # Initialize the embeddings using a normal distribution.
  U = tf.Variable(tf.random.normal(
      [A_train.dense_shape[0], embedding_dim], stddev=init_stddev))
  V = tf.Variable(tf.random.normal(
      [A_train.dense_shape[1], embedding_dim], stddev=init_stddev))
  train_loss = sparse_mean_square_error(A_train, U, V)
  test_loss = sparse_mean_square_error(A_test, U, V)
  metrics = {
      'train_error': train_loss,
      'test_error': test_loss
  }
  embeddings = {
      "userID": U,
      "artistID": V
  }
  return CFModel(embeddings, train_loss, [metrics])
```

```
# Make sure all column data types are correct before training model!

user_artists['userID'] = user_artists['userID'].astype(str)
user_artists['artistID'] = user_artists['artistID'].astype(str)
user_artists['weight'] = user_artists['weight'].astype(float)
user_artists = user_artists.sample(frac=1).reset_index(drop=True)
```

Great, now it’s time to train the model!

```
# Training our Model!
model = build_model(user_artists, embedding_dim=30, init_stddev=0.5)
model.train(num_iterations=1000, learning_rate=10.)
```

```
iteration 0: train_error=27.876650, test_error=28.287300
iteration 10: train_error=25.841696, test_error=28.020741
```

```
iteration 20: train_error=23.548723, test_error=27.250408
```

```
iteration 30: train_error=20.174698, test_error=25.138470
```

```
iteration 40: train_error=16.760786, test_error=22.696613
```

```
iteration 50: train_error=13.858467, test_error=20.611059
```

```
iteration 60: train_error=11.438461, test_error=18.814426
```

```
iteration 70: train_error=9.528853, test_error=17.381752
```

```
iteration 80: train_error=8.051212, test_error=16.295622
```

```
iteration 90: train_error=6.899530, test_error=15.485218
```

```
iteration 100: train_error=5.987197, test_error=14.879247
```

```
iteration 110: train_error=5.252829, test_error=14.422573
```

```
iteration 120: train_error=4.653678, test_error=14.075703
```

```
iteration 130: train_error=4.159267, test_error=13.810741
```

```
iteration 140: train_error=3.747157, test_error=13.607823
```

```
iteration 150: train_error=3.400398, test_error=13.452544
```

```
iteration 160: train_error=3.105986, test_error=13.334266
```

```
iteration 170: train_error=2.853845, test_error=13.245006
```

```
iteration 180: train_error=2.636119, test_error=13.178689
```

```
iteration 190: train_error=2.446647, test_error=13.130627
```

```
iteration 200: train_error=2.280563, test_error=13.097180
```

```
iteration 210: train_error=2.133995, test_error=13.075476
```

```
iteration 220: train_error=2.003838, test_error=13.063252
```

```
iteration 230: train_error=1.887582, test_error=13.058693
```

```
iteration 240: train_error=1.783181, test_error=13.060349
```

```
iteration 250: train_error=1.688956, test_error=13.067045
```

```
iteration 260: train_error=1.603519, test_error=13.077834
```

```
iteration 270: train_error=1.525712, test_error=13.091939
iteration 280: train_error=1.454569, test_error=13.108727
```

```
iteration 290: train_error=1.389274, test_error=13.127684
```

```
iteration 300: train_error=1.329134, test_error=13.148379
```

```
iteration 310: train_error=1.273563, test_error=13.170463
```

```
iteration 320: train_error=1.222054, test_error=13.193642
```

```
iteration 330: train_error=1.174176, test_error=13.217681
```

```
iteration 340: train_error=1.129554, test_error=13.242383
```

```
iteration 350: train_error=1.087862, test_error=13.267575
```

```
iteration 360: train_error=1.048818, test_error=13.293130
```

```
iteration 370: train_error=1.012174, test_error=13.318930
```

```
iteration 380: train_error=0.977712, test_error=13.344880
```

```
iteration 390: train_error=0.945241, test_error=13.370905
```

```
iteration 400: train_error=0.914591, test_error=13.396938
```

```
iteration 410: train_error=0.885611, test_error=13.422925
```

```
iteration 420: train_error=0.858168, test_error=13.448827
```

```
iteration 430: train_error=0.832141, test_error=13.474603
```

```
iteration 440: train_error=0.807423, test_error=13.500223
```

```
iteration 450: train_error=0.783917, test_error=13.525663
```

```
iteration 460: train_error=0.761537, test_error=13.550904
```

```
iteration 470: train_error=0.740203, test_error=13.575931
```

```
iteration 480: train_error=0.719845, test_error=13.600732
```

```
iteration 490: train_error=0.700396, test_error=13.625295
```

```
iteration 500: train_error=0.681799, test_error=13.649611
```

```
iteration 510: train_error=0.663999, test_error=13.673679
```

```
iteration 520: train_error=0.646948, test_error=13.697493
```

```
iteration 530: train_error=0.630598, test_error=13.721047
```

```
iteration 540: train_error=0.614910, test_error=13.744346
iteration 550: train_error=0.599844, test_error=13.767384
```

```
iteration 560: train_error=0.585365, test_error=13.790165
```

```
iteration 570: train_error=0.571441, test_error=13.812689
```

```
iteration 580: train_error=0.558041, test_error=13.834953
```

```
iteration 590: train_error=0.545137, test_error=13.856967
```

```
iteration 600: train_error=0.532704, test_error=13.878728
```

```
iteration 610: train_error=0.520716, test_error=13.900243
```

```
iteration 620: train_error=0.509151, test_error=13.921512
```

```
iteration 630: train_error=0.497989, test_error=13.942537
```

```
iteration 640: train_error=0.487209, test_error=13.963326
```

```
iteration 650: train_error=0.476794, test_error=13.983880
```

```
iteration 660: train_error=0.466725, test_error=14.004203
```

```
iteration 670: train_error=0.456986, test_error=14.024298
```

```
iteration 680: train_error=0.447564, test_error=14.044170
```

```
iteration 690: train_error=0.438442, test_error=14.063824
```

```
iteration 700: train_error=0.429608, test_error=14.083261
```

```
iteration 710: train_error=0.421050, test_error=14.102487
```

```
iteration 720: train_error=0.412754, test_error=14.121504
```

```
iteration 730: train_error=0.404711, test_error=14.140317
```

```
iteration 740: train_error=0.396909, test_error=14.158933
```

```
iteration 750: train_error=0.389339, test_error=14.177350
```

```
iteration 760: train_error=0.381990, test_error=14.195573
```

```
iteration 770: train_error=0.374854, test_error=14.213610
```

```
iteration 780: train_error=0.367923, test_error=14.231462
```

```
iteration 790: train_error=0.361189, test_error=14.249128
```

```
iteration 800: train_error=0.354643, test_error=14.266620
```

```
iteration 810: train_error=0.348279, test_error=14.283933
iteration 820: train_error=0.342089, test_error=14.301075
```

```
iteration 830: train_error=0.336067, test_error=14.318048
```

```
iteration 840: train_error=0.330208, test_error=14.334856
```

```
iteration 850: train_error=0.324504, test_error=14.351501
```

```
iteration 860: train_error=0.318951, test_error=14.367988
```

```
iteration 870: train_error=0.313542, test_error=14.384316
```

```
iteration 880: train_error=0.308273, test_error=14.400489
```

```
iteration 890: train_error=0.303140, test_error=14.416511
```

```
iteration 900: train_error=0.298136, test_error=14.432384
```

```
iteration 910: train_error=0.293258, test_error=14.448111
```

```
iteration 920: train_error=0.288502, test_error=14.463694
```

```
iteration 930: train_error=0.283863, test_error=14.479136
```

```
iteration 940: train_error=0.279337, test_error=14.494435
```

```
iteration 950: train_error=0.274922, test_error=14.509601
```

```
iteration 960: train_error=0.270612, test_error=14.524629
```

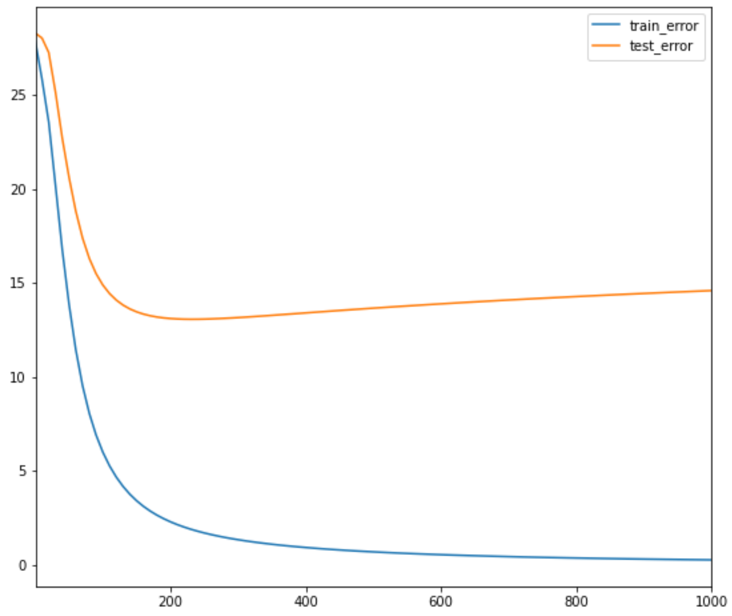
```
iteration 970: train_error=0.266406, test_error=14.539528
```

```
iteration 980: train_error=0.262298, test_error=14.554293
```

```
iteration 990: train_error=0.258287, test_error=14.568930
```

```
iteration 1000: train_error=0.254370, test_error=14.583439
```

```
[{'train_error': 0.25436953, 'test_error': 14.583439}]
```



Inspecting the Embeddings

In this section, we take a closer look at the learned embeddings, by

- computing your recommendations
- looking at the nearest neighbors of some artists,
- looking at the norms of the movie embeddings,
- visualizing the embedding in a projected embedding space.

A mapping from a discrete set (in this case, the set of queries, or the set of items to recommend) to a vector space called the embedding space. Many recommendation systems rely on learning an appropriate embedding representation of the queries and items.

Our recommendations are based on DOT and COSINE scores.

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

DOT = 'dot'
COSINE = 'cosine'
def compute_scores(query_embedding, item_embeddings, measure=DOT):

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

```
def artist_neighbors(model, title_substring, measure=DOT, k=6):
  ids =  artists[artists['name'].str.contains(title_substring)].index.values
  titles = artists.iloc[ids]['name'].values
  if len(titles) == 0:
    raise ValueError("Found no artist with title %s" % title_substring)
  print("Nearest neighbors of : %s." % titles[0])
  if len(titles) > 1:
    print("[Found more than one matching artist. Other candidates: {}]".format(
        ", ".join(titles[1:])))
  artistID = ids[0]
  scores = compute_scores(
      model.embeddings["artistID"][artistID], model.embeddings["artistID"],
      measure)
  score_key = measure + ' score'
  df = pd.DataFrame({
      score_key: list(scores),
      'names': artists['name'],
  })
  display.display(df.sort_values([score_key], ascending=False).head(k))
```

Now we can check out our first artist Recommedations!

We can pass a few artists name’s in to check out our first implementation

```
artist_neighbors(model, "Miley Cyrus", DOT)
artist_neighbors(model, "Miley Cyrus", COSINE)
```

```
Nearest neighbors of : Miley Cyrus.
[Found more than one matching artist. Other candidates: Miley Cyrusæ□□, Demi Lovato Ft.
Miley Cyrus Ft. Selena Gomez Ft. Jonas Brothers, Miley Cyrus and Billy Ray Cyrus, Miley
Cyrus and John Travolta, Hannah Montana and Miley Cyrus]
```

	dot score	names
455	5.878	Miley Cyrus
14357	5.107	Caravan Palace
7687	4.475	Alisa Jones
12329	4.473	Arnab (aimraj.com)
14154	4.464	Settle the Score
11564	4.452	Pg.99

```
Nearest neighbors of : Miley Cyrus.
[Found more than one matching artist. Other candidates: Miley Cyrusæ□□, Demi Lovato Ft.
Miley Cyrus Ft. Selena Gomez Ft. Jonas Brothers, Miley Cyrus and Billy Ray Cyrus, Miley
Cyrus and John Travolta, Hannah Montana and Miley Cyrus]
```

	cosine score	names
455	1.000	Miley Cyrus
7981	0.589	Bad Balance
14154	0.584	Settle the Score
15541	0.582	Tubelord
5404	0.582	Black Stone Cherry
1349	0.577	Judas Priest

```
artist_neighbors(model, "Coldplay", DOT)
artist_neighbors(model, "Coldplay", COSINE)
```

```
Nearest neighbors of : Coldplay.
[Found more than one matching artist. Other candidates: Jay-Z & Coldplay, Coldplay/U2]
```

	dot score	names
59	5.170	Coldplay
15987	4.699	The Exploding Hearts
9920	4.611	Joseph LoDuca
16864	4.568	Blue Merle
9953	4.421	Giacomo Puccini
10078	4.187	Martin Denny

```
Nearest neighbors of : Coldplay.
[Found more than one matching artist. Other candidates: Jay-Z & Coldplay, Coldplay/U2]
```

	cosine score	names
59	1.000	Coldplay
15987	0.625	The Exploding Hearts
2548	0.619	Billy Connolly
12539	0.607	Honor
8349	0.594	Lauren Nichols
5709	0.584	Lion

```python
model_lowinit = build_model(user_artists, embedding_dim=30, init_stddev=0.05)
model_lowinit.train(num_iterations=1000, learning_rate=10.)
artist_neighbors(model_lowinit, "The Beatles", DOT)
artist_neighbors(model_lowinit, "The Beatles", COSINE)
```

```
iteration 0: train_error=26.002579, test_error=26.444706
```

```
iteration 10: train_error=25.982233, test_error=26.443041
```

```
iteration 20: train_error=25.943939, test_error=26.425926
```

```
iteration 30: train_error=25.792734, test_error=26.304892
```

```
iteration 40: train_error=25.021013, test_error=25.592434
```

```
iteration 50: train_error=22.743046, test_error=23.363958
```

```
iteration 60: train_error=20.431293, test_error=21.067860
```

```
iteration 70: train_error=18.046412, test_error=18.814585
```

```
iteration 80: train_error=15.623238, test_error=16.541756
```

```
iteration 90: train_error=13.572453, test_error=14.647625
```

```
iteration 100: train_error=11.946621, test_error=13.186904
```

```
iteration 110: train_error=10.664365, test_error=12.074239
```

```
iteration 120: train_error=9.630803, test_error=11.210501
```

```
iteration 130: train_error=8.775890, test_error=10.522435
```

```
iteration 140: train_error=8.053210, test_error=9.961926
```

```
iteration 150: train_error=7.432146, test_error=9.497918
```

```
iteration 160: train_error=6.891699, test_error=9.109549
```

```
iteration 170: train_error=6.416732, test_error=8.781923
```

```
iteration 180: train_error=5.995811, test_error=8.503820
```

```
iteration 190: train_error=5.620031, test_error=8.266509
```

```
iteration 200: train_error=5.282333, test_error=8.063087
```

```
iteration 210: train_error=4.977081, test_error=7.888067
```

```
iteration 220: train_error=4.699740, test_error=7.737051
```

```
iteration 230: train_error=4.446614, test_error=7.606472
```

```
iteration 240: train_error=4.214639, test_error=7.493399
```

```
iteration 250: train_error=4.001239, test_error=7.395392
```

```
iteration 260: train_error=3.804225, test_error=7.310396
iteration 270: train_error=3.621735, test_error=7.236676
```

```
iteration 280: train_error=3.452188, test_error=7.172760
```

```
iteration 290: train_error=3.294242, test_error=7.117407
```

```
iteration 300: train_error=3.146752, test_error=7.069567
```

```
iteration 310: train_error=3.008730, test_error=7.028355
```

```
iteration 320: train_error=2.879311, test_error=6.993021
```

```
iteration 330: train_error=2.757727, test_error=6.962930
```

```
iteration 340: train_error=2.643288, test_error=6.937538
```

```
iteration 350: train_error=2.535372, test_error=6.916371
```

```
iteration 360: train_error=2.433419, test_error=6.899023
```

```
iteration 370: train_error=2.336927, test_error=6.885127
```

```
iteration 380: train_error=2.245450, test_error=6.874360
```

```
iteration 390: train_error=2.158591, test_error=6.866426
```

```
iteration 400: train_error=2.076000, test_error=6.861063
```

```
iteration 410: train_error=1.997367, test_error=6.858026
```

```
iteration 420: train_error=1.922417, test_error=6.857090
```

```
iteration 430: train_error=1.850908, test_error=6.858053
```

```
iteration 440: train_error=1.782623, test_error=6.860727
```

```
iteration 450: train_error=1.717368, test_error=6.864940
```

```
iteration 460: train_error=1.654966, test_error=6.870535
```

```
iteration 470: train_error=1.595259, test_error=6.877368
```

```
iteration 480: train_error=1.538102, test_error=6.885310
```

```
iteration 490: train_error=1.483362, test_error=6.894239
```

```
iteration 500: train_error=1.430917, test_error=6.904047
```

```
iteration 510: train_error=1.380653, test_error=6.914634
```

```
iteration 520: train_error=1.332466, test_error=6.925913
```

```
iteration 530: train_error=1.286259, test_error=6.937799
iteration 540: train_error=1.241940, test_error=6.950219
```

```
iteration 550: train_error=1.199424, test_error=6.963105
```

```
iteration 560: train_error=1.158632, test_error=6.976396
```

```
iteration 570: train_error=1.119487, test_error=6.990037
```

```
iteration 580: train_error=1.081919, test_error=7.003978
```

```
iteration 590: train_error=1.045860, test_error=7.018171
```

```
iteration 600: train_error=1.011246, test_error=7.032578
```

```
iteration 610: train_error=0.978016, test_error=7.047159
```

```
iteration 620: train_error=0.946112, test_error=7.061881
```

```
iteration 630: train_error=0.915478, test_error=7.076715
```

```
iteration 640: train_error=0.886062, test_error=7.091630
```

```
iteration 650: train_error=0.857812, test_error=7.106604
```

```
iteration 660: train_error=0.830679, test_error=7.121613
```

```
iteration 670: train_error=0.804616, test_error=7.136636
```

```
iteration 680: train_error=0.779579, test_error=7.151657
```

```
iteration 690: train_error=0.755523, test_error=7.166657
```

```
iteration 700: train_error=0.732407, test_error=7.181637
```

```
iteration 710: train_error=0.710191, test_error=7.196543
```

```
iteration 720: train_error=0.688836, test_error=7.211403
```

```
iteration 730: train_error=0.668306, test_error=7.226192
```

```
iteration 740: train_error=0.648566, test_error=7.240903
```

```
iteration 750: train_error=0.629580, test_error=7.255527
```

```
iteration 760: train_error=0.611317, test_error=7.270058
```

```
iteration 770: train_error=0.593744, test_error=7.284487
```

```
iteration 780: train_error=0.576834, test_error=7.298810
```

```
iteration 790: train_error=0.560556, test_error=7.313022
```

```
iteration 800: train_error=0.544883, test_error=7.327120
iteration 810: train_error=0.529790, test_error=7.341098
```

```
iteration 820: train_error=0.515252, test_error=7.354957
```

```
iteration 830: train_error=0.501243, test_error=7.368693
```

```
iteration 840: train_error=0.487743, test_error=7.382302
```

```
iteration 850: train_error=0.474729, test_error=7.395783
```

```
iteration 860: train_error=0.462180, test_error=7.409138
```

```
iteration 870: train_error=0.450077, test_error=7.422363
```

```
iteration 880: train_error=0.438401, test_error=7.435458
```

```
iteration 890: train_error=0.427135, test_error=7.448425
```

```
iteration 900: train_error=0.416260, test_error=7.461260
```

```
iteration 910: train_error=0.405761, test_error=7.473966
```

```
iteration 920: train_error=0.395622, test_error=7.486543
```

```
iteration 930: train_error=0.385828, test_error=7.498992
```

```
iteration 940: train_error=0.376366, test_error=7.511312
```

```
iteration 950: train_error=0.367221, test_error=7.523508
```

```
iteration 960: train_error=0.358381, test_error=7.535576
```

```
iteration 970: train_error=0.349834, test_error=7.547518
```

```
iteration 980: train_error=0.341567, test_error=7.559339
```

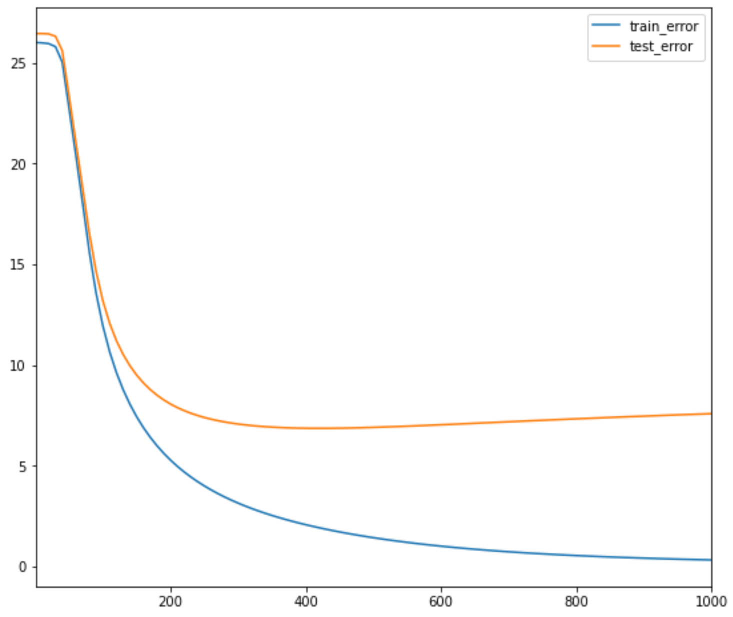
```
iteration 990: train_error=0.333570, test_error=7.571035
```

```
 iteration 1000: train_error=0.325832, test_error=7.582612Nearest neighbors of : The
Beatles.
[Found more than one matching artist. Other candidates: The Beatles with Billy Preston]
```

	dot score	names
8142	0.444	egredior
15641	0.438	Dwarr
10287	0.412	Righteous Jams
8919	0.406	David Helpling & Jon Jenkins
11832	0.396	Cocoa Tea
1848	0.394	YUI

```
Nearest neighbors of : The Beatles.
[Found more than one matching artist. Other candidates: The Beatles with Billy Preston]
```

	cosine score	names
221	1.000	The Beatles
7232	0.604	Scratch Acid
15458	0.596	Restless
7246	0.595	Madeline Powell
7222	0.579	S.O.M.A.
2539	0.573	Kajagoogoo



Regularization in Matrix Factorization

In the previous section, our loss was defined as the mean squared error on the observed part of the rating matrix. This can be problematic as the model does not learn how to place the embeddings of irrelevant artists. This phenomenon is known as folding.

We will add regularization terms that will address this issue

```
"""
  Args:
    ratings: the DataFrame of movie ratings.
    embedding_dim: The dimension of the embedding space.
    regularization_coeff: The regularization coefficient lambda.
    gravity_coeff: The gravity regularization coefficient lambda_g.
  Returns:
    A CFModel object that uses a regularized loss.
"""

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

def build_regularized_model(
    ratings, embedding_dim=3, regularization_coeff=.1, gravity_coeff=1.,
    init_stddev=0.1):
  # Split the ratings DataFrame into train and test.
  train_ratings, test_ratings = split_dataframe(ratings)
  # SparseTensor representation of the train and test datasets.
  A_train = build_rating_sparse_tensor(train_ratings)
  A_test = build_rating_sparse_tensor(test_ratings)
  U = tf.Variable(tf.random_normal(
      [A_train.dense_shape[0], embedding_dim], stddev=init_stddev))
  V = tf.Variable(tf.random_normal(
      [A_train.dense_shape[1], embedding_dim], stddev=init_stddev))

  error_train = sparse_mean_square_error(A_train, U, V)
  error_test = sparse_mean_square_error(A_test, U, V)
  gravity_loss = gravity_coeff * gravity(U, V)
  regularization_loss = regularization_coeff * (
      tf.reduce_sum(U*U)/U.shape[0].value + tf.reduce_sum(V*V)/V.shape[0].value)
  total_loss = error_train + regularization_loss + gravity_loss
  losses = {
      'train_error_observed': error_train,
      'test_error_observed': error_test,
  }
  loss_components = {
      'observed_loss': error_train,
      'regularization_loss': regularization_loss,
      'gravity_loss': gravity_loss,
  }
  embeddings = {"userId": U, "artistID": V}

  return CFModel(embeddings, total_loss, [losses, loss_components]), U, V
```

```
reg_model, u, v = build_regularized_model(
    user_artists, regularization_coeff=0.1, gravity_coeff=1.0, embedding_dim=35,
    init_stddev=.05)
reg_model.train(num_iterations=1000, learning_rate=20.)
```

```
 iteration 0: train_error_observed=26.094936, test_error_observed=25.604876,
observed_loss=26.094936, regularization_loss=0.017498, gravity_loss=0.000219
```

```
 iteration 10: train_error_observed=26.034243, test_error_observed=25.590090,
observed_loss=26.034243, regularization_loss=0.017630, gravity_loss=0.000222
```

```
 iteration 20: train_error_observed=25.325861, test_error_observed=24.964613,
observed_loss=25.325861, regularization_loss=0.022670, gravity_loss=0.000480
```

```
 iteration 30: train_error_observed=20.846054, test_error_observed=20.733400,
observed_loss=20.846054, regularization_loss=0.067324, gravity_loss=0.018447
```

```
 iteration 40: train_error_observed=16.239809, test_error_observed=16.631695,
observed_loss=16.239809, regularization_loss=0.135587, gravity_loss=0.093673
```

```
 iteration 50: train_error_observed=12.560231, test_error_observed=13.341138,
observed_loss=12.560231, regularization_loss=0.215193, gravity_loss=0.254236
```

```
 iteration 60: train_error_observed=10.253874, test_error_observed=11.370034,
observed_loss=10.253874, regularization_loss=0.284669, gravity_loss=0.455121
```

```
 iteration 70: train_error_observed=8.740067, test_error_observed=10.180342,
observed_loss=8.740067, regularization_loss=0.342006, gravity_loss=0.655284
```

```
 iteration 80: train_error_observed=7.659132, test_error_observed=9.413710,
observed_loss=7.659132, regularization_loss=0.390411, gravity_loss=0.837758
```

```
 iteration 90: train_error_observed=6.842360, test_error_observed=8.896672,
observed_loss=6.842360, regularization_loss=0.432161, gravity_loss=0.993988
```

```
 iteration 100: train_error_observed=6.198863, test_error_observed=8.538752,
observed_loss=6.198863, regularization_loss=0.468811, gravity_loss=1.120254
```

```
 iteration 110: train_error_observed=5.673008, test_error_observed=8.287972,
observed_loss=5.673008, regularization_loss=0.501630, gravity_loss=1.216755
```

```
 iteration 120: train_error_observed=5.229159, test_error_observed=8.112465,
observed_loss=5.229159, regularization_loss=0.531668, gravity_loss=1.286180
```

```
 iteration 130: train_error_observed=4.844241, test_error_observed=7.991718,
observed_loss=4.844241, regularization_loss=0.559738, gravity_loss=1.332366
```

```
 iteration 140: train_error_observed=4.503096, test_error_observed=7.911713,
observed_loss=4.503096, regularization_loss=0.586419, gravity_loss=1.359385
```

```
 iteration 150: train_error_observed=4.195695, test_error_observed=7.862312,
observed_loss=4.195695, regularization_loss=0.612102, gravity_loss=1.371051
```

```
 iteration 160: train_error_observed=3.915470, test_error_observed=7.836016,
observed_loss=3.915470, regularization_loss=0.637028, gravity_loss=1.370710
```

```
 iteration 170: train_error_observed=3.658155, test_error_observed=7.827332,
observed_loss=3.658155, regularization_loss=0.661321, gravity_loss=1.361216
```

```
 iteration 180: train_error_observed=3.420907, test_error_observed=7.832309,
observed_loss=3.420907, regularization_loss=0.685019, gravity_loss=1.344960
```

```
 iteration 190: train_error_observed=3.201679, test_error_observed=7.848094,
observed_loss=3.201679, regularization_loss=0.708102, gravity_loss=1.323917
```

```
 iteration 200: train_error_observed=2.998847, test_error_observed=7.872585,
observed_loss=2.998847, regularization_loss=0.730522, gravity_loss=1.299697
```

```
 iteration 210: train_error_observed=2.811043, test_error_observed=7.904152,
observed_loss=2.811043, regularization_loss=0.752217, gravity_loss=1.273577
```

```
 iteration 220: train_error_observed=2.637091, test_error_observed=7.941486,
observed_loss=2.637091, regularization_loss=0.773128, gravity_loss=1.246552
```

```
 iteration 230: train_error_observed=2.475980, test_error_observed=7.983490,
observed_loss=2.475980, regularization_loss=0.793200, gravity_loss=1.219373
```

```
 iteration 240: train_error_observed=2.326831, test_error_observed=8.029225,
observed_loss=2.326831, regularization_loss=0.812385, gravity_loss=1.192591
```

```
 iteration 250: train_error_observed=2.188856, test_error_observed=8.077885,
observed_loss=2.188856, regularization_loss=0.830644, gravity_loss=1.166593
```

```
 iteration 260: train_error_observed=2.061326, test_error_observed=8.128763,
observed_loss=2.061326, regularization_loss=0.847951, gravity_loss=1.141643
```

```
 iteration 270: train_error_observed=1.943545, test_error_observed=8.181257,
observed_loss=1.943545, regularization_loss=0.864285, gravity_loss=1.117904
```

```
 iteration 280: train_error_observed=1.834839, test_error_observed=8.234841,
observed_loss=1.834839, regularization_loss=0.879639, gravity_loss=1.095466
```

```
 iteration 290: train_error_observed=1.734553, test_error_observed=8.289075,
observed_loss=1.734553, regularization_loss=0.894016, gravity_loss=1.074365
```

```
 iteration 300: train_error_observed=1.642051, test_error_observed=8.343583,
observed_loss=1.642051, regularization_loss=0.907427, gravity_loss=1.054596
```

```
 iteration 310: train_error_observed=1.556726, test_error_observed=8.398047,
observed_loss=1.556726, regularization_loss=0.919892, gravity_loss=1.036129
```

```
 iteration 320: train_error_observed=1.477999, test_error_observed=8.452214,
observed_loss=1.477999, regularization_loss=0.931438, gravity_loss=1.018914
```

```
 iteration 330: train_error_observed=1.405326, test_error_observed=8.505864,
observed_loss=1.405326, regularization_loss=0.942099, gravity_loss=1.002888
```

```
 iteration 340: train_error_observed=1.338201, test_error_observed=8.558830,
observed_loss=1.338201, regularization_loss=0.951910, gravity_loss=0.987984
```

```
 iteration 350: train_error_observed=1.276157, test_error_observed=8.610969,
observed_loss=1.276157, regularization_loss=0.960911, gravity_loss=0.974130
```

```
 iteration 360: train_error_observed=1.218762, test_error_observed=8.662175,
observed_loss=1.218762, regularization_loss=0.969144, gravity_loss=0.961255
```

```
 iteration 370: train_error_observed=1.165623, test_error_observed=8.712361,
observed_loss=1.165623, regularization_loss=0.976651, gravity_loss=0.949289
```

```
 iteration 380: train_error_observed=1.116380, test_error_observed=8.761468,
observed_loss=1.116380, regularization_loss=0.983475, gravity_loss=0.938162
```

```
 iteration 390: train_error_observed=1.070704, test_error_observed=8.809445,
observed_loss=1.070704, regularization_loss=0.989656, gravity_loss=0.927812
```

```
 iteration 400: train_error_observed=1.028297, test_error_observed=8.856263,
observed_loss=1.028297, regularization_loss=0.995236, gravity_loss=0.918176
```

```
 iteration 410: train_error_observed=0.988887, test_error_observed=8.901902,
observed_loss=0.988887, regularization_loss=1.000254, gravity_loss=0.909199
```

```
 iteration 420: train_error_observed=0.952225, test_error_observed=8.946354,
observed_loss=0.952225, regularization_loss=1.004748, gravity_loss=0.900828
```

```
 iteration 430: train_error_observed=0.918087, test_error_observed=8.989614,
observed_loss=0.918087, regularization_loss=1.008755, gravity_loss=0.893012
```

```
 iteration 440: train_error_observed=0.886267, test_error_observed=9.031692,
observed_loss=0.886267, regularization_loss=1.012308, gravity_loss=0.885709
```

```
 iteration 450: train_error_observed=0.856577, test_error_observed=9.072593,
observed_loss=0.856577, regularization_loss=1.015441, gravity_loss=0.878876
```

```
 iteration 460: train_error_observed=0.828845, test_error_observed=9.112336,
observed_loss=0.828845, regularization_loss=1.018185, gravity_loss=0.872475
```

```
 iteration 470: train_error_observed=0.802917, test_error_observed=9.150940,
observed_loss=0.802917, regularization_loss=1.020568, gravity_loss=0.866473
```

```
 iteration 480: train_error_observed=0.778649, test_error_observed=9.188424,
observed_loss=0.778649, regularization_loss=1.022620, gravity_loss=0.860836
```

```
 iteration 490: train_error_observed=0.755912, test_error_observed=9.224814,
observed_loss=0.755912, regularization_loss=1.024365, gravity_loss=0.855537
```

```
 iteration 500: train_error_observed=0.734585, test_error_observed=9.260138,
observed_loss=0.734585, regularization_loss=1.025828, gravity_loss=0.850548
```

```
 iteration 510: train_error_observed=0.714560, test_error_observed=9.294419,
observed_loss=0.714560, regularization_loss=1.027032, gravity_loss=0.845846
```

```
 iteration 520: train_error_observed=0.695739, test_error_observed=9.327687,
observed_loss=0.695739, regularization_loss=1.027998, gravity_loss=0.841409
```

```
 iteration 530: train_error_observed=0.678029, test_error_observed=9.359970,
observed_loss=0.678029, regularization_loss=1.028747, gravity_loss=0.837218
```

```
 iteration 540: train_error_observed=0.661347, test_error_observed=9.391296,
observed_loss=0.661347, regularization_loss=1.029296, gravity_loss=0.833252
```

```
 iteration 550: train_error_observed=0.645618, test_error_observed=9.421697,
observed_loss=0.645618, regularization_loss=1.029663, gravity_loss=0.829497
```

```
 iteration 560: train_error_observed=0.630771, test_error_observed=9.451198,
observed_loss=0.630771, regularization_loss=1.029864, gravity_loss=0.825937
```

```
 iteration 570: train_error_observed=0.616742, test_error_observed=9.479830,
observed_loss=0.616742, regularization_loss=1.029914, gravity_loss=0.822557
```

```
 iteration 580: train_error_observed=0.603473, test_error_observed=9.507619,
observed_loss=0.603473, regularization_loss=1.029827, gravity_loss=0.819346
```

```
 iteration 590: train_error_observed=0.590910, test_error_observed=9.534596,
observed_loss=0.590910, regularization_loss=1.029616, gravity_loss=0.816292
```

```
 iteration 600: train_error_observed=0.579002, test_error_observed=9.560786,
observed_loss=0.579002, regularization_loss=1.029293, gravity_loss=0.813383
```

```
 iteration 610: train_error_observed=0.567706, test_error_observed=9.586218,
observed_loss=0.567706, regularization_loss=1.028868, gravity_loss=0.810611
```

```
 iteration 620: train_error_observed=0.556979, test_error_observed=9.610915,
observed_loss=0.556979, regularization_loss=1.028352, gravity_loss=0.807966
```

```
 iteration 630: train_error_observed=0.546783, test_error_observed=9.634906,
observed_loss=0.546783, regularization_loss=1.027754, gravity_loss=0.805440
```

```
 iteration 640: train_error_observed=0.537083, test_error_observed=9.658211,
observed_loss=0.537083, regularization_loss=1.027084, gravity_loss=0.803026
```

```
 iteration 650: train_error_observed=0.527845, test_error_observed=9.680857,
observed_loss=0.527845, regularization_loss=1.026348, gravity_loss=0.800717
```

```
 iteration 660: train_error_observed=0.519040, test_error_observed=9.702867,
observed_loss=0.519040, regularization_loss=1.025554, gravity_loss=0.798506
```

```
 iteration 670: train_error_observed=0.510641, test_error_observed=9.724262,
observed_loss=0.510641, regularization_loss=1.024709, gravity_loss=0.796387
```

```
 iteration 680: train_error_observed=0.502621, test_error_observed=9.745063,
observed_loss=0.502621, regularization_loss=1.023819, gravity_loss=0.794355
```

```
 iteration 690: train_error_observed=0.494957, test_error_observed=9.765293,
observed_loss=0.494957, regularization_loss=1.022890, gravity_loss=0.792405
```

```
 iteration 700: train_error_observed=0.487628, test_error_observed=9.784971,
observed_loss=0.487628, regularization_loss=1.021926, gravity_loss=0.790533
```

```
 iteration 710: train_error_observed=0.480612, test_error_observed=9.804118,
observed_loss=0.480612, regularization_loss=1.020934, gravity_loss=0.788733
```

```
 iteration 720: train_error_observed=0.473891, test_error_observed=9.822749,
observed_loss=0.473891, regularization_loss=1.019916, gravity_loss=0.787002
```

```
 iteration 730: train_error_observed=0.467449, test_error_observed=9.840884,
observed_loss=0.467449, regularization_loss=1.018878, gravity_loss=0.785336
```

```
 iteration 740: train_error_observed=0.461268, test_error_observed=9.858541,
observed_loss=0.461268, regularization_loss=1.017822, gravity_loss=0.783732
```

```
 iteration 750: train_error_observed=0.455334, test_error_observed=9.875738,
observed_loss=0.455334, regularization_loss=1.016752, gravity_loss=0.782186
```

```
 iteration 760: train_error_observed=0.449632, test_error_observed=9.892487,
observed_loss=0.449632, regularization_loss=1.015672, gravity_loss=0.780695
```

```
 iteration 770: train_error_observed=0.444151, test_error_observed=9.908805,
observed_loss=0.444151, regularization_loss=1.014583, gravity_loss=0.779257
```

```
 iteration 780: train_error_observed=0.438877, test_error_observed=9.924709,
observed_loss=0.438877, regularization_loss=1.013488, gravity_loss=0.777869
```

```
 iteration 790: train_error_observed=0.433800, test_error_observed=9.940209,
observed_loss=0.433800, regularization_loss=1.012390, gravity_loss=0.776529
```

```
 iteration 800: train_error_observed=0.428909, test_error_observed=9.955324,
observed_loss=0.428909, regularization_loss=1.011291, gravity_loss=0.775233
```

```
 iteration 810: train_error_observed=0.424195, test_error_observed=9.970064,
observed_loss=0.424195, regularization_loss=1.010192, gravity_loss=0.773981
```

```
 iteration 820: train_error_observed=0.419647, test_error_observed=9.984441,
observed_loss=0.419647, regularization_loss=1.009095, gravity_loss=0.772769
```

```
 iteration 830: train_error_observed=0.415258, test_error_observed=9.998468,
observed_loss=0.415258, regularization_loss=1.008002, gravity_loss=0.771596
```

```
 iteration 840: train_error_observed=0.411019, test_error_observed=10.012156,
observed_loss=0.411019, regularization_loss=1.006914, gravity_loss=0.770461
```

```
 iteration 850: train_error_observed=0.406924, test_error_observed=10.025521,
observed_loss=0.406924, regularization_loss=1.005833, gravity_loss=0.769361
```

```
 iteration 860: train_error_observed=0.402965, test_error_observed=10.038568,
observed_loss=0.402965, regularization_loss=1.004759, gravity_loss=0.768294
```

```
 iteration 870: train_error_observed=0.399135, test_error_observed=10.051309,
observed_loss=0.399135, regularization_loss=1.003693, gravity_loss=0.767261
```

```
 iteration 880: train_error_observed=0.395428, test_error_observed=10.063754,
observed_loss=0.395428, regularization_loss=1.002637, gravity_loss=0.766258
```

```
 iteration 890: train_error_observed=0.391839, test_error_observed=10.075915,
observed_loss=0.391839, regularization_loss=1.001591, gravity_loss=0.765285
```

```
 iteration 900: train_error_observed=0.388362, test_error_observed=10.087797,
observed_loss=0.388362, regularization_loss=1.000555, gravity_loss=0.764340
```

```
 iteration 910: train_error_observed=0.384991, test_error_observed=10.099412,
observed_loss=0.384991, regularization_loss=0.999531, gravity_loss=0.763422
```

```
 iteration 920: train_error_observed=0.381723, test_error_observed=10.110769,
observed_loss=0.381723, regularization_loss=0.998519, gravity_loss=0.762530
```

```
 iteration 930: train_error_observed=0.378552, test_error_observed=10.121874,
observed_loss=0.378552, regularization_loss=0.997519, gravity_loss=0.761664
```

```
 iteration 940: train_error_observed=0.375474, test_error_observed=10.132736,
observed_loss=0.375474, regularization_loss=0.996533, gravity_loss=0.760821
```

```
 iteration 950: train_error_observed=0.372485, test_error_observed=10.143364,
observed_loss=0.372485, regularization_loss=0.995559, gravity_loss=0.760001
```

```
 iteration 960: train_error_observed=0.369582, test_error_observed=10.153762,
observed_loss=0.369582, regularization_loss=0.994599, gravity_loss=0.759203
```

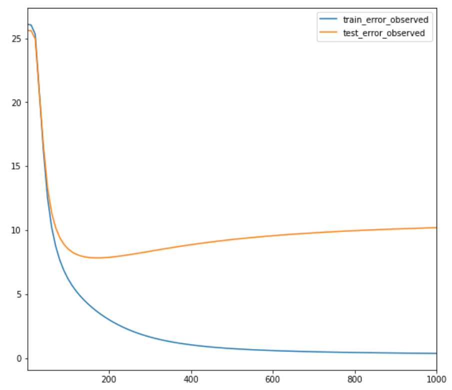
```
 iteration 970: train_error_observed=0.366760, test_error_observed=10.163940,
observed_loss=0.366760, regularization_loss=0.993652, gravity_loss=0.758426
```

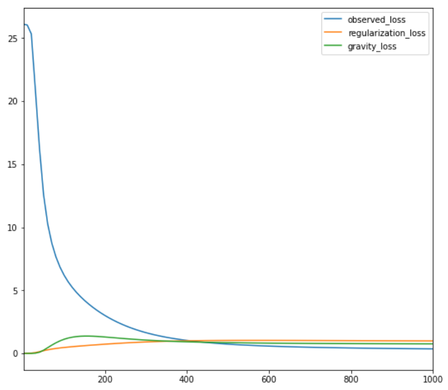
```
 iteration 980: train_error_observed=0.364016, test_error_observed=10.173903,
observed_loss=0.364016, regularization_loss=0.992719, gravity_loss=0.757670
```

```
 iteration 990: train_error_observed=0.361347, test_error_observed=10.183660,
observed_loss=0.361347, regularization_loss=0.991801, gravity_loss=0.756933
```

```
 iteration 1000: train_error_observed=0.358750, test_error_observed=10.193214,
observed_loss=0.358750, regularization_loss=0.990896, gravity_loss=0.756215
```

```
[{'train_error_observed': 0.35874966, 'test_error_observed': 10.193214},
 {'observed_loss': 0.35874966,
  'regularization_loss': 0.9908964,
  'gravity_loss': 0.75621516}]
```





Observe that adding the regularization terms results in a higher MSE, both on the training and test set. However, as we will see, the quality of the recommendations improves.

```
artist_neighbors(reg_model, "Madonna", DOT)
artist_neighbors(reg_model, "Madonna", COSINE)
```

```
Nearest neighbors of : Madonna.
[Found more than one matching artist. Other candidates: Madonna feat. Gogol Bordello,
Lady GaGa vs. Madonna, Madonna feat. Justin Timberlake, Madonna & Justin, Madonna &
Justin Timberlake & Timbaland, Britney, Christina, Madonna, Madonna & Antonio Banderas]
```

	dot score	names
9304	2.563	John Prine
14518	2.345	Electrobelle
10436	2.249	Ð Ð°Ð¹Ð½Ð°
13152	2.065	Xavier CafÃ©Ã¯ne
12268	2.061	Orange Juice
12060	1.970	Faraquet

```
Nearest neighbors of : Madonna.
[Found more than one matching artist. Other candidates: Madonna feat. Gogol Bordello,
Lady GaGa vs. Madonna, Madonna feat. Justin Timberlake, Madonna & Justin, Madonna &
Justin Timberlake & Timbaland, Britney, Christina, Madonna, Madonna & Antonio Banderas]
```

	cosine score	names
61	1.000	Madonna
50	0.976	Daft Punk
53	0.976	New Order
49	0.975	Kylie Minogue
52	0.974	Goldfrapp
17473	0.972	Ronnie Day

Before going onto check our model we will export our user and artist embedings to be used in the clustering section of our paper

```
# Assign artist_embeddings to a pandas dataframe
artist_embeddings = pd.DataFrame(reg_model.embeddings["artistID"])

# Assign user_embeddings to a pandas dataframe
user_embeddings = pd.DataFrame(reg_model.embeddings["userId"])
```

```
artist_embeddings.head(1)
```

	0	1	2	3	4	5	6	7	8	9	...	25	2
0	-0.052	0.187	-0.033	0.143	0.124	-0.003	-0.026	-0.204	-0.117	0.005	...	0.051	0.35

1 rows × 35 columns

```
user_embeddings.head(1)
```

	0	1	2	3	4	5	6	7	8	9	...	25	2
0	-1.210	0.946	0.216	0.420	-1.090	-0.233	0.470	-0.508	-0.727	-0.466	...	-0.864	0.93

1 rows × 35 columns

Conclusion

Here we have built two recommender systems and it is clear which one is better. By regularising our matrix we are able to find more insightful recommendation for our users.

Checking Most popular artists recommendations for top 10 genres

If capturing popularity information is desirable, then you should prefer dot product. However, if you're not careful, the popular items may end up dominating the recommendations. It is for this resaon we will be using COSINE distance as our recommendation metric for our artists

After the many iterations of running our system and observing our system it is clear that COSINE is the best for recommendations.

Before we continue we will also need to associate an artists to a genre.

```
# Join user_taggedartists andd tags based on the tagID column to give each artists a
genre

user_taggedartists["artistID"].value_counts()

user_artists_genre = pd.merge(user_taggedartists, tags, on=["tagID"], how='left')

# Check to make sure there is no loss of rows and one column has been added
user_artists_genre.shape
```

```
(186479, 7)
```

In this section we will be able to see the recommendations for artists for the top 10 Genres and the corresponding count values

```
# Read in data set with count of genres
tags_name_counts = pd.read_csv('clean_data/tags_name_counts.csv')
```

```
# This will allow us to see what the most popular genres are
genre_to_include =
list(zip(tags_name_counts['Genre'],tags_name_counts['Count_Artist_Genre']))[:10]
genre_to_include
```

```
[('rock', 7503),
 ('pop', 5418),
 ('alternative', 5251),
 ('electronic', 4672),
 ('indie', 4458),
 ('female vocalists', 4228),
 ('80s', 2791),
 ('dance', 2739),
 ('alternative rock', 2631),
 ('classic rock', 2287)]
```

For the top 10 genres we will now be able to isolate all genres into their own dataframe for further analysis. Once we isolate genres, we will be able to identify which are the most popular artists in each genre for use in our recommedation system.

```
# Isolating all artists based on their genre for the top 10 genres
artist_rock = user_artists_genre[user_artists_genre['tagValue'] == "rock"]
artist_pop = user_artists_genre[user_artists_genre['tagValue'] == "pop"]
artist_alternative = user_artists_genre[user_artists_genre['tagValue'] == "alternative"]
artist_electronic = user_artists_genre[user_artists_genre['tagValue'] == "electronic"]
artist_indie = user_artists_genre[user_artists_genre['tagValue'] == "indie"]
artist_female_vocalists = user_artists_genre[user_artists_genre['tagValue'] == "female
vocalists"]
artist_80s = user_artists_genre[user_artists_genre['tagValue'] == "80s"]
artist_dance = user_artists_genre[user_artists_genre['tagValue'] == "dance"]
artist_alternative_rock = user_artists_genre[user_artists_genre['tagValue'] == "80"]
artist_classic_rock = user_artists_genre[user_artists_genre['tagValue'] == "rock"]
```

Here we can do a simple join for each artists to see which is the most popular for each genre

```
artist_name_counts = artist_name_counts.rename(columns={"id":"artistID"})
```

```
# Show corresponding listener counts for each artist and their genre
artists_rock_genre = pd.merge(artist_rock, artist_name_counts, on=["artistID"],
how='left')
artists_pop_genre = pd.merge(artist_pop, artist_name_counts, on=["artistID"],
how='left')
artists_alternative_genre = pd.merge(artist_alternative, artist_name_counts, on=
["artistID"], how='left')
artists_electronic_genre = pd.merge(artist_electronic, artist_name_counts, on=
["artistID"], how='left')
artists_indie_genre = pd.merge(artist_indie, artist_name_counts, on=["artistID"],
how='left')
artists_female_vocalists_genre = pd.merge(artist_female_vocalists, artist_name_counts,
on=["artistID"], how='left')
artists_80s_genre = pd.merge(artist_80s, artist_name_counts, on=["artistID"],
how='left')
artists_dance_genre = pd.merge(artist_dance, artist_name_counts, on=["artistID"],
how='left')
artists_alternative_rock_genre = pd.merge(artist_alternative_rock, artist_name_counts,
on=["artistID"], how='left')
artists_classic_rock_genre = pd.merge(artist_classic_rock, artist_name_counts, on=
["artistID"], how='left')
```

Now that each artists has a corresponding genre and listener count we can put the artist through our system to see what we get!

```
artists_rock_genre.iloc[artists_rock_genre['Total_user_listens'].idxmax()]
```

```
userID                                               19
artistID                                            288
tagID                                                72
day                                                   1
month                                                 9
year                                               2006
tagValue                                           rock
Total_user_listens                              484.000
name                                            Rihanna
url                   http://www.last.fm/music/Rihanna
Name: 131, dtype: object
```

For the rock genre, Rihanna has the most user listens

```
artist_neighbors(reg_model, "Rihanna", COSINE)
```

```
Nearest neighbors of : Rihanna.
[Found more than one matching artist. Other candidates: Rihanna (feat. Drake), Jay-Z,
Bono, The Edge & Rihanna, RihannaÌ⁻, Sean Paul ft. Rihanna, Rihanna-remixado REnan,
\Eminem f_ Rihanna]
```

	cosine score	names
282	1.000	Rihanna
269	0.891	Angie Stone
267	0.890	Avant
265	0.888	Mos Def
9366	0.882	Scrapy
272	0.876	2Pac

From first look Britney Spears is one of Rihannas nearest neighbors. This is a very good recommendation for the rock genre

```
artists_pop_genre.iloc[artists_pop_genre['Total_user_listens'].idxmax()]
```

```
userID                                                 131
artistID                                               289
tagID                                                   23
day                                                      1
month                                                    5
year                                                  2010
tagValue                                               pop
Total_user_listens                                 522.000
name                                        Britney Spears
url                   http://www.last.fm/music/Britney+Spears
Name: 457, dtype: object
```

```
artist_neighbors(reg_model, "Britney Spears", COSINE)
```

```
Nearest neighbors of : Britney Spears.
[Found more than one matching artist. Other candidates: Michael Jackson & Britney
Spears, Britney Spears vs Avril Lavigne, Britney Spearsã  ¦, Britney Spearsâ¼, Britney
Spearsì¨, Panic! at the Disco feat. Britney Spears and Gwen Stefani]
```

	cosine score	names
283	1.000	Britney Spears
265	0.911	Mos Def
290	0.880	Sugababes
301	0.872	Kate Voegele
269	0.872	Angie Stone
287	0.866	Ashlee Simpson

Janet Jackson and Rihanna are very good recommendations for Britney Spears who is the most popular pop artist

```
artists_alternative_genre.iloc[artists_alternative_genre['Total_user_listens'].idxmax()]
```

```
userID                                          157
artistID                                         89
tagID                                            78
day                                               1
month                                            10
year                                           2008
tagValue                                alternative
Total_user_listens                          611.000
name                                      Lady Gaga
url                   http://www.last.fm/music/Lady+Gaga
Name: 509, dtype: object
```

```
artist_neighbors(reg_model, "Lady Gaga", COSINE)
```

```
Nearest neighbors of : Lady Gaga.
[Found more than one matching artist. Other candidates: Lady Gaga VS Christina
Aguilera, BeyoncÃ© e Lady Gaga, Lady Gaga feat BeyoncÃ©]
```

	cosine score	names
83	1.000	Lady Gaga
4036	0.689	VETO
10162	0.681	Peter Pan Speedrock
16502	0.662	Jacaszek & MiÅ‚ka
15384	0.651	Babangida
16401	0.643	Jomed

These are weird ones for Lady Gaga….

```
artists_electronic_genre.iloc[artists_electronic_genre['Total_user_listens'].idxmax()]
```

```
userID                                                 157
artistID                                                89
tagID                                                   17
day                                                      1
month                                                   10
year                                                  2008
tagValue                                        electronic
Total_user_listens                                 611.000
name                                             Lady Gaga
url                      http://www.last.fm/music/Lady+Gaga
Name: 383, dtype: object
```

```
artists_indie_genre.iloc[artists_indie_genre['Total_user_listens'].idxmax()]
```

```
userID                                                   102
artistID                                                 227
tagID                                                     80
day                                                        1
month                                                      3
year                                                    2010
tagValue                                               indie
Total_user_listens                                   480.000
name                                             The Beatles
url                      http://www.last.fm/music/The+Beatles
Name: 225, dtype: object
```

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

```
Nearest neighbors of : The Beatles.
[Found more than one matching artist. Other candidates: The Beatles with Billy Preston]
```

	cosine score	names
221	1.000	The Beatles
223	0.987	The Killers
225	0.984	Devendra Banhart
230	0.983	Clap Your Hands Say Yeah
228	0.981	Nirvana
224	0.980	Green Day

This is probably the best recommendation so far. The system has recommend several other big bands for users who like the Beatles

```
artists_female_vocalists_genre.iloc[artists_female_vocalists_genre['Total_user_listens']
.idxmax()]
```

```
userID                                                      468
artistID                                                    289
tagID                                                       129
day                                                           1
month                                                        10
year                                                       2010
tagValue                                       female vocalists
Total_user_listens                                      522.000
name                                             Britney Spears
url                      http://www.last.fm/music/Britney+Spears
Name: 1099, dtype: object
```

```
artists_80s_genre.iloc[artists_80s_genre['Total_user_listens'].idxmax()]
```

```
userID                                               478
artistID                                              67
tagID                                                 24
day                                                    1
month                                                  3
year                                                2009
tagValue                                             80s
Total_user_listens                               429.000
name                                             Madonna
url                      http://www.last.fm/music/Madonna
Name: 593, dtype: object
```

```
artist_neighbors(reg_model, "Madonna", COSINE)
```

```
Nearest neighbors of : Madonna.
[Found more than one matching artist. Other candidates: Madonna feat. Gogol Bordello,
Lady GaGa vs. Madonna, Madonna feat. Justin Timberlake, Madonna & Justin, Madonna &
Justin Timberlake & Timbaland, Britney, Christina, Madonna, Madonna & Antonio Banderas]
```

	cosine score	names
61	1.000	Madonna
50	0.976	Daft Punk
53	0.976	New Order
49	0.975	Kylie Minogue
52	0.974	Goldfrapp
17473	0.972	Ronnie Day

Kylie Minogue and Daft Punk are 2 good recommendations for Madonnna

```
artists_dance_genre.iloc[artists_dance_genre['Total_user_listens'].idxmax()]
```

```
userID                                                 157
artistID                                                89
tagID                                                   38
day                                                      1
month                                                   10
year                                                  2008
tagValue                                             dance
Total_user_listens                                 611.000
name                                             Lady Gaga
url                     http://www.last.fm/music/Lady+Gaga
Name: 298, dtype: object
```

```
artists_alternative_rock_genre.iloc[artists_alternative_rock_genre['Total_user_listens']
.idxmax()]
```

```
userID                                                                1503
artistID                                                              1835
tagID                                                                 9693
day                                                                      1
month                                                                    1
year                                                                  2008
tagValue                                                                80
Total_user_listens                                                      13
name                                               The Cinematic Orchestra
url                     http://www.last.fm/music/The+Cinematic+Orchestra
Name: 0, dtype: object
```

```
artist_neighbors(reg_model, "The Cinematic Orchestra", COSINE)
```

```
Nearest neighbors of : The Cinematic Orchestra.
```

	cosine score	names
1826	1.000	The Cinematic Orchestra
6093	0.849	The Bony King of Nowhere
6094	0.822	Sad Day For Puppets
6096	0.817	Jeremy Messersmith
6097	0.808	Guillemots
6101	0.796	N*E*R*D

To be honest I am not sure if this is a good recommendation or not as I have never heard of any of these people….

```
artists_classic_rock_genre.iloc[artists_classic_rock_genre['Total_user_listens'].idxmax(
)]
```

```
userID                                              19
artistID                                           288
tagID                                               72
day                                                  1
month                                                9
year                                              2006
tagValue                                          rock
Total_user_listens                             484.000
name                                           Rihanna
url                      http://www.last.fm/music/Rihanna
Name: 131, dtype: object
```

Overall I think the system works well! Usually giving 2/3 solid recommendations based on the artist. Although, I think the system struggles when it comes to artists who have a lot of genres attached to their name eg Lady Gaga. From Spotify it never gets 100% of recommendations right and the same with YouTube. I often can find recommendations poor and am lucky to find one/two songs that can be good from their recommendations. Although recommender systems are subjective to the user, I think it has performed relatively well. No system is ever going to get a 100% approval rate from their user. It is a shame that it struggles with artists that have many genres associated to their name.

Clustering

Just as an extra piece of work I was interested in how the genres would cluster together. I did some research online and came across an interesting article that can be found here. This article was based on a recommender system also. The clustering that was used in this project was for books and aticles on wikipedia. By performing some clustering on our data we might be able to validate the results that are found from our recommendation system. The clustering methods that will be used in this section are Uniform Manifold Approximation and Projection (UMAP) and t-Stochastic Distributed Neighbors Embedding (TSNE)

I saw some very interesting similar projects online that adopted using TSNE and UMAP for their clusters. The projects in question are Visualizing Embeddings With t-SNE and Building a Recommendation System Using Neural Network Embeddings. The use of clusters in both of these projects were very interesting and helped to provide some very good insights for their systems and data

```python
# Import our modules for clustering
import time
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
```

```python
# Create function to reduce our dimensions
def reduce_dim(weights, components = 3, method = 'tsne'):
    """Reduce dimensions of embeddings"""
    if method == 'tsne':
        return TSNE(components, metric = 'cosine').fit_transform(weights)
    elif method == 'umap':
        # Might want to try different parameters for UMAP
        return UMAP(n_components=components, metric = 'cosine',
                    init = 'random', n_neighbors = 5).fit_transform(weights)
```

Firstly we will look at user and artist embeddings by using TSNE clustering

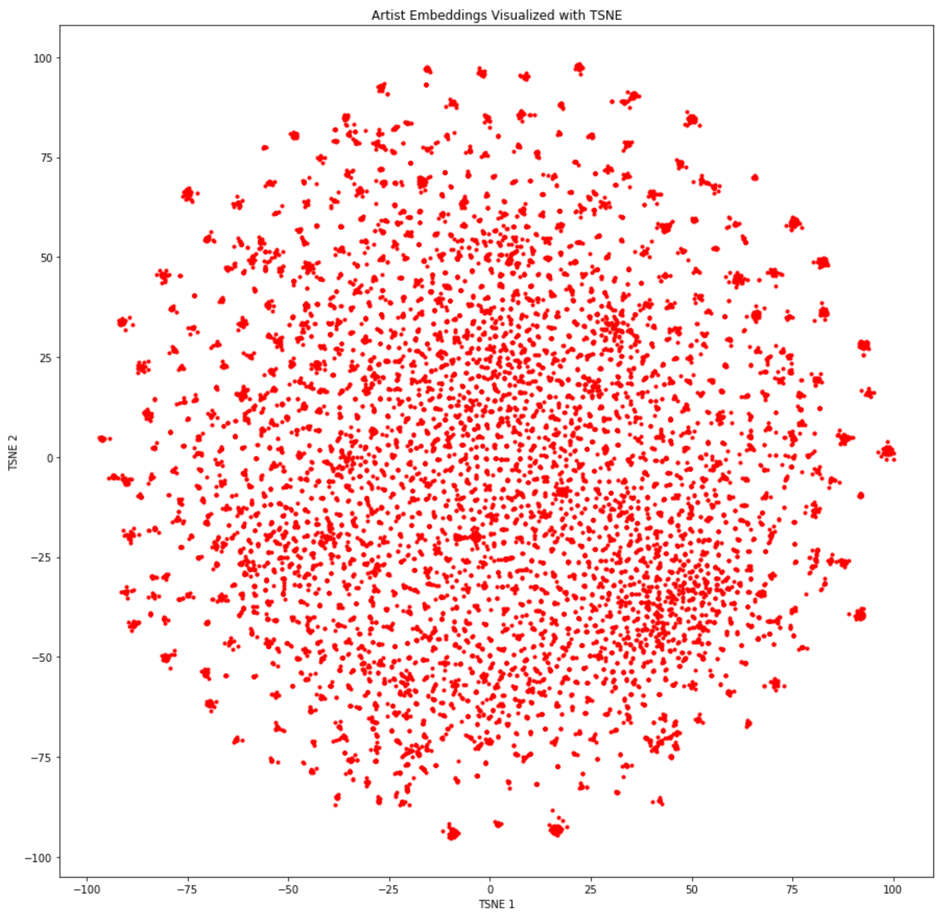
Since it is hard to visualize embeddings in a higher-dimensional space (when the embedding dimension k>3), one approach is to project the embeddings to a lower dimensional space. T-SNE (T-distributed Stochastic Neighbor Embedding) is an algorithm that projects the embeddings while attempting to preserve their pariwise distances.

```python
# Reduce Artist embeddings to 2 dimensions
artist_r = reduce_dim(artist_embeddings, components = 2, method = 'tsne')
artist_r.shape
```

```
/home/michael/anaconda3/lib/python3.7/site-packages/sklearn/manifold/_t_sne.py:699:
FutureWarning: 'square_distances' has been introduced in 0.24 to help phase out legacy
squaring behavior. The 'legacy' setting will be removed in 1.1 (renaming of 0.26), and
the default setting will be changed to True. In 1.3, 'square_distances' will be removed
altogether, and distances will be squared by default. Set 'square_distances'=True to
silence this warning.
  FutureWarning
```

```
(17632, 2)
```

```python
from IPython.core.interactiveshell import InteractiveShell

InteractiveShell.ast_node_interactivity = 'all'
#InteractiveShell.ast_node_interactivity = 'last'

plt.figure(figsize = (15, 15))
plt.plot(artist_r[:, 0], artist_r[:, 1], 'r.')
plt.xlabel('TSNE 1'); plt.ylabel('TSNE 2'); plt.title('Artist Embeddings Visualized with
TSNE');
```



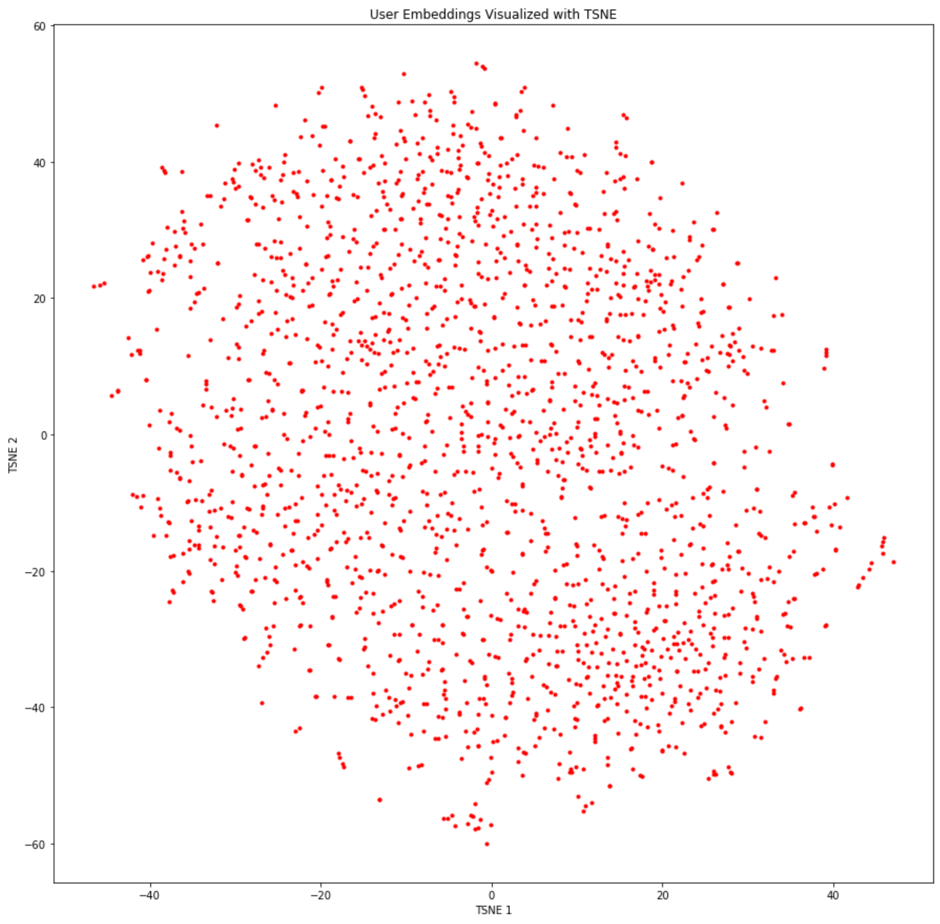
From looking at this cluster you can see there are lost of clusters forming arounf the outide of the big cluster. These might be interesting to investigate

```python
# Reduce user embeddings to 2 demensions
user_r = reduce_dim(user_embeddings, components = 2, method = 'tsne')
user_r.shape
```

```
/home/michael/anaconda3/lib/python3.7/site-packages/sklearn/manifold/_t_sne.py:699:
FutureWarning: 'square_distances' has been introduced in 0.24 to help phase out legacy
squaring behavior. The 'legacy' setting will be removed in 1.1 (renaming of 0.26), and
the default setting will be changed to True. In 1.3, 'square_distances' will be removed
altogether, and distances will be squared by default. Set 'square_distances'=True to
silence this warning.
  FutureWarning
```

```
(1892, 2)
```

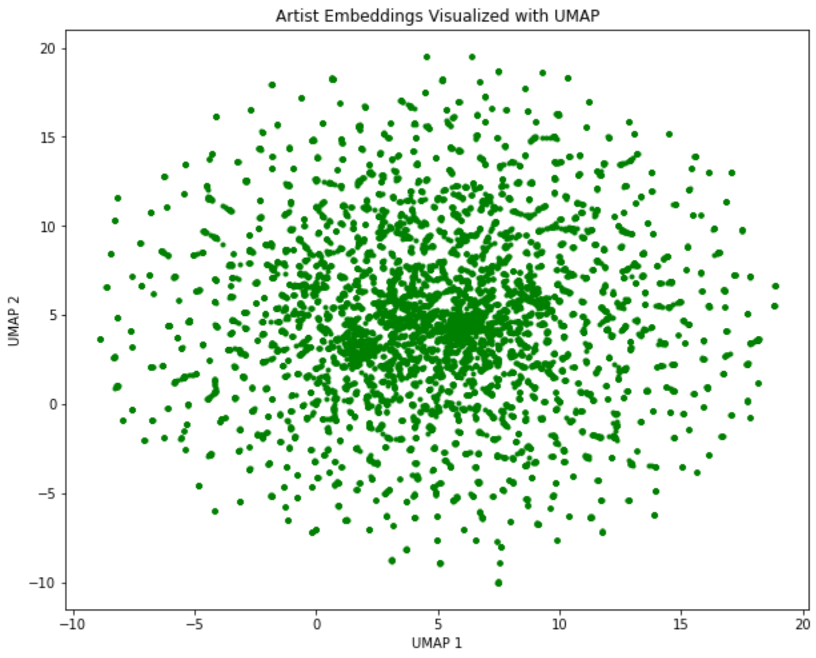
```
from IPython.core.interactiveshell import InteractiveShell

InteractiveShell.ast_node_interactivity = 'all'
#InteractiveShell.ast_node_interactivity = 'last'

plt.figure(figsize = (15, 15))
plt.plot(user_r[:, 0], user_r[:, 1], 'r.')
plt.xlabel('TSNE 1'); plt.ylabel('TSNE 2'); plt.title('User Embeddings Visualized with
TSNE');
```



When TSNE clustering is performed on the User embeddings there is nothing that seems intersting from my initial inspection. Artist embeddings will be investigated further!

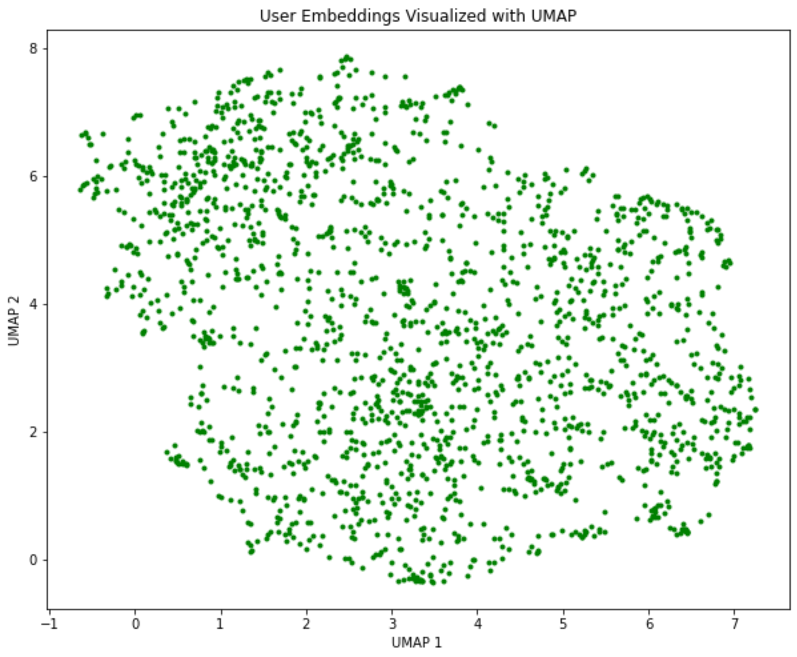
There were some potentially interesting clusters in the artist embeddings by using TSNE. We will now use UMAP to see if there are any significant changes

```
from umap import UMAP
artist_ru = reduce_dim(artist_embeddings, components = 2, method = 'umap')

plt.figure(figsize = (10, 8))
plt.plot(artist_ru[:, 0], artist_ru[:, 1], 'g.');
plt.xlabel('UMAP 1'); plt.ylabel('UMAP 2'); plt.title('Artist Embeddings Visualized with
UMAP');
```



There is an obvious cluster of the artist embedings all in the middle when we use the UMAP approach for clustering

```
user_ru = reduce_dim(user_embeddings, components = 2, method = 'umap')

plt.figure(figsize = (10, 8))
plt.plot(user_ru[:, 0], user_ru[:, 1], 'g.');
plt.xlabel('UMAP 1'); plt.ylabel('UMAP 2'); plt.title('User Embeddings Visualized with
UMAP');
```



Going forward I think it will be interesting to use the UMAP and TSNE approach for the artist embeddings only. It will be interesting to see if these clusters can back up our claim for the recommender system that we built. I only be looking at the artists embeddings as from a glance at the user embeddings they don’t look as if they will provide us with any meaningful findings on our data.

Firstly from our UMAP cluster there is a big cluster of nodes in the centre of the graph, we can isolate these clusters and see if we can make any conclusions on them!

By restricting the x and y labels we will be able to have a closer look at our graphs. After we have our desired ranges, we can export our data and see if they have anything in common.

```
plt.figure(figsize = (10, 8))
plt.plot(artist_ru[:, 0], artist_ru[:, 1], 'g.');
plt.xlabel('UMAP 1'); plt.ylabel('UMAP 2'); plt.title('Artist Embeddings Visualized with
UMAP');
plt.ylim(0,10)
plt.xlim(0,10)
```

```
<Figure size 720x576 with 0 Axes>
```

```
[<matplotlib.lines.Line2D at 0x7f0868ddc6d0>]
```

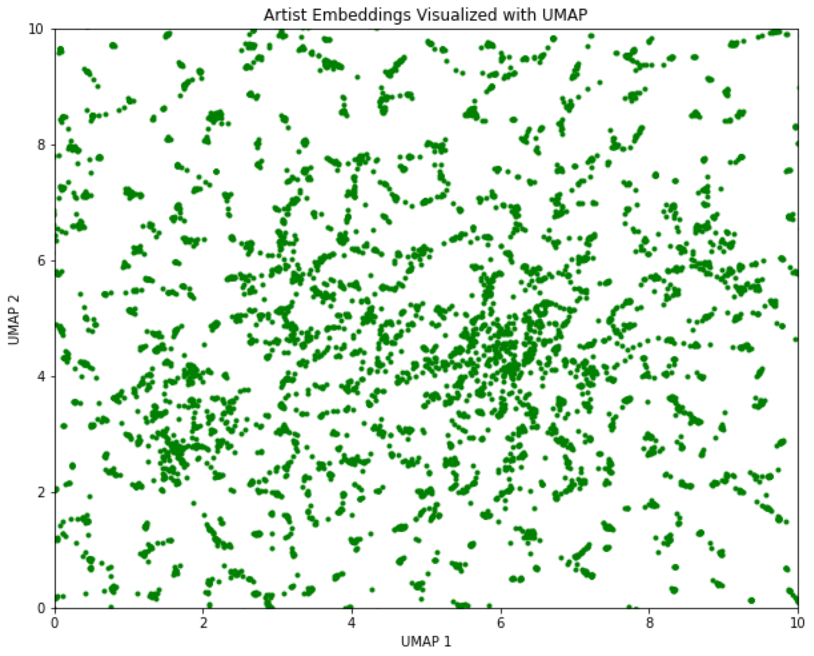
```
Text(0.5, 0, 'UMAP 1')
```

```
Text(0, 0.5, 'UMAP 2')
```

```
Text(0.5, 1.0, 'Artist Embeddings Visualized with UMAP')
```

```
(0.0, 10.0)
```

```
(0.0, 10.0)
```



At first glance the big cluster we had in UMAP is made up of lots of little clusters around the middle

Now we can attempt to adopt the methods used in the [Visualizing Embeddings With t-SNE](#) project. The author uses TSNE to find interesting clusters based on genres and artists names. Before we continue we will need to create a new table for use in our new function.

```
# Making a combination of genres and artists data frame!

artist_test = artist_name_counts[["artistID", "name"]]
```

```python
# Some helper functions for plotting annotated t-SNE visualizations

artist_test['x'] = artist_r[:, 0]
artist_test['y'] = artist_r[:, 1]

try:
    from adjustText import adjust_text
except ImportError:
    def adjust_text(*args, **kwargs):
        pass

def adjust_text(*args, **kwargs):
    pass

def plot_bg(bg_alpha=.01, figsize=(13, 9), emb_2d=None):
    """Create and return a plot of all our artist embeddings with very low opacity.
    (Intended to be used as a basis for further - more prominent - plotting of a
    subset of artists. Having the overall shape of the map space in the background is
    useful for context.)
    """
    if emb_2d is None:
        emb_2d = artist_r
    fig, ax = plt.subplots(figsize=figsize)
    X = emb_2d[:, 0]
    Y = emb_2d[:, 1]
    ax.scatter(X, Y, alpha=bg_alpha)
    return ax

def annotate_sample(n, n_listens_thresh=0):
    """Plot our embeddings with a random sample of n artists listens.
    Only selects artists where the number of ratings is at least n_listens_thresh.
    """
    sample = artist_name_counts[artist_name_counts.Total_user_listens >=
n_listens_thresh].sample(
        n, random_state=1)
    plot_with_annotations(sample.index)

def plot_by_title_pattern(pattern, **kwargs):
    """Plot all artist whose titles match the given regex pattern.
    """
    match = artist_test[artist_test.name.str.contains(pattern)]
    return plot_with_annotations(match.index, **kwargs)

def add_annotations(ax, label_indices, emb_2d=None, **kwargs):
    if emb_2d is None:
        emb_2d = artist_r
    X = emb_2d[label_indices, 0]
    Y = emb_2d[label_indices, 1]
    ax.scatter(X, Y, **kwargs)

def plot_with_annotations(label_indices, text=True, labels=None, alpha=1, **kwargs):
    ax = plot_bg(**kwargs)
    Xlabeled = artist_r[label_indices, 0]
    Ylabeled = artist_r[label_indices, 1]
    if labels is not None:
        for x, y, label in zip(Xlabeled, Ylabeled, labels):
            ax.scatter(x, y, alpha=alpha, label=label, marker='1',
                       s=90,
                      )
        fig.legend()
    else:
        ax.scatter(Xlabeled, Ylabeled, alpha=alpha, color='green')

    if text:
        # TODO: Add abbreviated title column
        titles = artist_test.loc[label_indices, 'name'].values
        texts = []
        for label, x, y in zip(titles, Xlabeled, Ylabeled):
            t = ax.annotate(label, xy=(x, y))
            texts.append(t)
        adjust_text(texts,
                    #expand_text=(1.01, 1.05),
                    arrowprops=dict(arrowstyle='->', color='red'),
                   )
    return ax


FS = (13, 9)
def plot_region(x0, x1, y0, y1, text=True):
    """Plot the region of the mapping space bounded by the given x and y limits.
    """
    fig, ax = plt.subplots(figsize=FS)
    pts = artist_test[
        (artist_test.x >= x0) & (artist_test.x <= x1)
        & (artist_test.y >= y0) & (artist_test.y <= y1)
    ]
    ax.scatter(pts.x, pts.y, alpha=.6)
    ax.set_xlim(x0, x1)
    ax.set_ylim(y0, y1)
    if text:
        texts = []
        for label, x, y in zip(pts.name.values, pts.x.values, pts.y.values):
            t = ax.annotate(label, xy=(x, y))
            texts.append(t)
        adjust_text(texts, expand_text=(1.01, 1.05))
    return ax


def plot_region_around(title, margin=5, **kwargs):
    """Plot the region of the mapping space in the neighbourhood of the the artist with
    the given title. The margin parameter controls the size of the neighbourhood around
    the artist.
    """
    xmargin = ymargin = margin
    match = artist_test[artist_test.name == title]
    assert len(match) == 1
    row = match.iloc[0]
    return plot_region(row.x-xmargin, row.x+xmargin, row.y-ymargin, row.y+ymargin,
**kwargs)
```

```
/home/michael/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:3:
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
  This is separate from the ipykernel package so we can avoid doing imports until
```

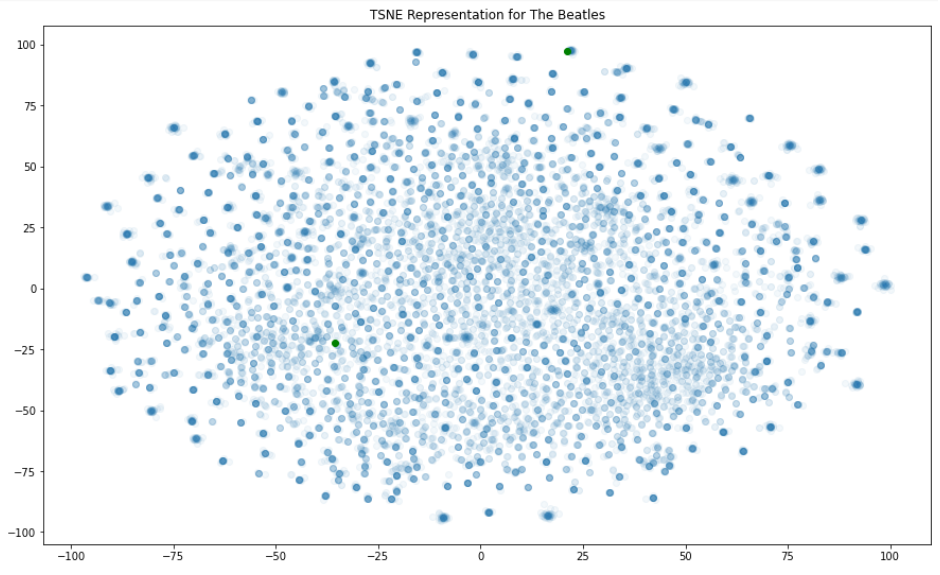
```
plt.figure(figsize = (10, 6))

plot_by_title_pattern('The Beatles', figsize=(15, 9), bg_alpha=.05, text=False);
plt.title("TSNE Representation for The Beatles")
```

```
<Figure size 720x432 with 0 Axes>
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0869060ad0>
```

```
Text(0.5, 1.0, 'TSNE Representation for The Beatles')
```

```
<Figure size 720x432 with 0 Axes>
```



```
plt.figure(figsize = (20, 20))

plot_region_around('The Beatles', 4);
plt.title("Nearest Neighbors for The Beatles")
```

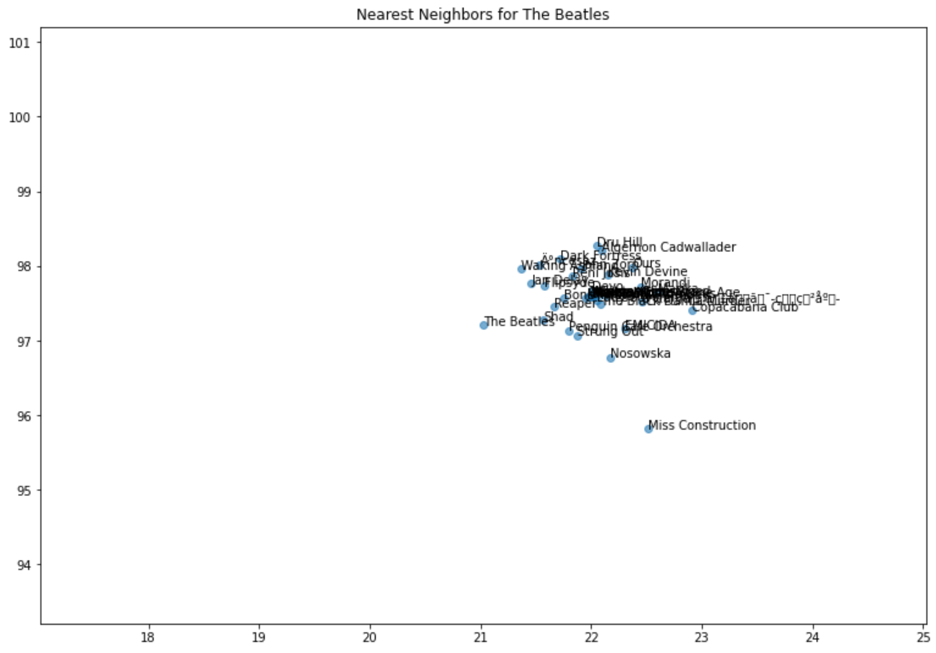
```
<Figure size 1440x1440 with 0 Axes>
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f07f109d610>
```

```
Text(0.5, 1.0, 'Nearest Neighbors for The Beatles')
```

```
<Figure size 1440x1440 with 0 Axes>
```

```
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 130 missing from
current font.
  font.set_text(s, 0.0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 131 missing from
current font.
  font.set_text(s, 0.0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 134 missing from
current font.
  font.set_text(s, 0.0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 143 missing from
current font.
  font.set_text(s, 0.0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 136 missing from
current font.
  font.set_text(s, 0.0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 144 missing from
current font.
  font.set_text(s, 0.0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:214: RuntimeWarning: Glyph 151 missing from
current font.
  font.set_text(s, 0.0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 130 missing from
current font.
  font.set_text(s, 0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 131 missing from
current font.
  font.set_text(s, 0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 134 missing from
current font.
  font.set_text(s, 0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 143 missing from
current font.
  font.set_text(s, 0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 136 missing from
current font.
  font.set_text(s, 0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 144 missing from
current font.
  font.set_text(s, 0, flags=flags)
/home/michael/anaconda3/lib/python3.7/site-
packages/matplotlib/backends/backend_agg.py:183: RuntimeWarning: Glyph 151 missing from
current font.
  font.set_text(s, 0, flags=flags)
```



Conclusions

Although I am unable to acieve my desired results for discovering clusters based on genre for our artists embeddings for UMAP and TSNE clusters due to time contraints, I do believe that we would discover some interesting results. I would think that most of our genres would cluster relatively tightly together. This would mean that recommendations would be based around artist genres. The projects that I got received inspiration from above were definitly very interesting and if I had a bit more time it would definitly something I would like to implement in the future.

Our two main UMAP and TSNE clusters have some ineresting patterns but without being able to isolate genres it made it difficult to uncover anything interesting about these clusters.

Testing System on my Spotify account

As part of this section I will be scraping data from my own Spotify account and see how the Recommendations produced compare to what I actually like. This will be a good way to see if my recommender system is good and gives accurate recommendations. Spotify is known for recommending songs to its users. They do this by the disover weekly playlist that come out for each user every Monday. This is just one of the many cases of Spotify using recommendation systems for its users. Spotify prides itself on their recommendations allowing users to uncover new music tastes. Personally I think this is the ultimate test of the recommender system!

The Spotify's Web API lets your applications fetch data from the Spotify music catalog and manage user's playlists and saved music. "Based on simple REST principles, our Web API endpoints return metadata in JSON forma

With Spotify wrapped 2021 being released on December 1st, I would be interested to see how may music from the past year would give me as suggestions if it was 2011. First things first was to connect to the spotify web API.

```python
# Importing the relevant python packages in order to interact with the Spotify API
# Spotipy allows us to interact with the Spotify web API to extract our data.
import json
import spotipy
import pandas as pd
from spotipy.oauth2 import SpotifyClientCredentials
```

```python
client_id = 'a7b3106b9dbd42cabb70096e1ed5cb60' #insert your client id
client_secret = '04a471a49143428d943827874630fcc7' # insert your client secret id here

client_credentials_manager = SpotifyClientCredentials(client_id, client_secret)
sp = spotipy.Spotify(client_credentials_manager=client_credentials_manager)

playlist_id='spotify:playlist:37i9dQZF1ELWSvGWJD6EgH?si=94d4b37ba8354def' #insert your
playlist id
results = sp.playlist(playlist_id)
```

```
# create a list of song ids
ids=[]

for item in results['tracks']['items']:
        track = item['track']['id']
        ids.append(track)

song_meta={'id':[],'album':[], 'name':[],
           'artist':[],'explicit':[],'popularity':[]}

for song_id in ids:
    # get song's meta data
    meta = sp.track(song_id)

    # song id
    song_meta['id'].append(song_id)

    # album name
    album=meta['album']['name']
    song_meta['album']+=[album]

    # song name
    song=meta['name']
    song_meta['name']+=[song]

    # artists name
    s = ', '
    artist=s.join([singer_name['name'] for singer_name in meta['artists']])
    song_meta['artist']+=[artist]

    # explicit: lyrics could be considered offensive or unsuitable for children
    explicit=meta['explicit']
    song_meta['explicit'].append(explicit)

    # song popularity
    popularity=meta['popularity']
    song_meta['popularity'].append(popularity)

song_meta_df=pd.DataFrame.from_dict(song_meta)

# check the song feature
features = sp.audio_features(song_meta['id'])
# change dictionary to dataframe
features_df=pd.DataFrame.from_dict(features)

# convert milliseconds to mins
# duration_ms: The duration of the track in milliseconds.
# 1 minute = 60 seconds = 60 × 1000 milliseconds = 60,000 ms
features_df['duration_ms']=features_df['duration_ms']/60000

# combine two dataframe
final_df=song_meta_df.merge(features_df)
```

The playlsit that has been taken from my pesonal spotify account. It is a collection of my songs from my spotify wrapped 2020. It was made by spotify based on my listening habits of 2020. As it is from 2020, it will require some data cleaning. Nine years is a long time in the musi world so obviously not every artist in this playlist will be in the LastFM data set. We will need to remove any unnescary artists.

Spotify obviously provides excellent recommendations and it is clear once we see the data. Not only can we see genre and artist name, but we can all see the popularity and danceability of each song which I think is a really cool feature. It is no wonder that Spotify is the music giant that it is today.

```
final_df.head(1)
```

	id	album	name	artist	explicit	popularity	danceability	en
0	2wAJTrFhCnQyNSD3oUgTZO	Cole World: The Sideline Story	Work Out	J. Cole	True	82	0.831	0

1 rows × 23 columns

We will compare names from artist_name_counts to our spotify data and only keep artists who appead in artist_name_counts

Firstly we will change the name of the artist name in our spotify data frame to match that of our artist_name_counts data frame

```
# Changin names to allow us to perform a join
final_df = final_df.rename(columns={"name":"song","artist":"name"})
final_df.head(1)
```

	id	album	song	name	explicit	popularity	danceability	ene
0	2wAJTrFhCnQyNSD3oUgTZO	Cole World: The Sideline Story	Work Out	J. Cole	True	82	0.831	0

1 rows × 23 columns

Before we isolate only artists that are in the LastFM data set we can first check and see how many unique artists are in this playlist.

```
# Checking number of unique artists
len(final_df['name'].unique().tolist())
```

```
84
```

Of the 100 entries, 84 artists are unique

Our next step is to only keep artists that are in the LasFM data set. We can do this by using the isin function

```
final_df = final_df[final_df.name.isin(artist_name_counts["name"])]
```

Now we can check the number of unique artists again. This time all the artists will be from 2011 or before

```
# Checking number of unique artists
len(final_df['name'].unique().tolist())
```

```
37
```

```
final_df.head(1)
```

	id	album	song	name	explicit	popularity	danceability	ene
0	2wAJTrFhCnQyNSD3oUgTZO	Cole World: The Sideline Story	Work Out	J. Cole	True	82	0.831	0

1 rows × 23 columns

Really we are only interested in the artists in our data frame, therfore we can simply just isolate them in their own dataframe

```
# Retrieving artists from spotify data
spotify_artists_2011  = final_df["name"]
spotify_artists_2011.head(1)
```

```
0    J. Cole
Name: name, dtype: object
```

```
# Returning data frame as opposed to displaying it
def artist_neighbors_spotify(model, title_substring, measure=DOT, k=6):
  ids =  artists[artists['name'].str.contains(title_substring)].index.values
  titles = artists.iloc[ids]['name'].values
  if len(titles) == 0:
    raise ValueError("Found no artist with title %s" % title_substring)
  print("Nearest neighbors of : %s." % titles[0])
  if len(titles) > 1:
    print("[Found more than one matching artist. Other candidates: {}]".format(
        ", ".join(titles[1:])))
  artistID = ids[0]
  scores = compute_scores(
      model.embeddings["artistID"][artistID], model.embeddings["artistID"],
      measure)
  score_key = measure + ' score'
  df = pd.DataFrame({
      score_key: list(scores),
      'names': artists['name'],
  })
  return df.sort_values([score_key], ascending=False).head(k)
```

Now we will be able to get our Cosine scores for all entries of our spotify data in order to receive our very own recommendation!

```
# Here we make a list of all the cosine nearest neighbor recommendations for analysis
Cosine_recommendataions = []
error_list = []
for artist in range (0, len(spotify_artists_2011)):
    try:
        Cosine_recommendataions.append((artist_neighbors_spotify(reg_model,
spotify_artists_2011[artist], COSINE)))
# An unknown error was occuring when this ran so I added in a try/except clause to
combat it!
    except:
        pass
;
```

```
Nearest neighbors of : J. Cole.
Nearest neighbors of : The White Stripes.
Nearest neighbors of : Fleetwood Mac.
Nearest neighbors of : Aloe Blacc.
Nearest neighbors of : Elbow.
Nearest neighbors of : MGMT.
Nearest neighbors of : Stromae.
Nearest neighbors of : Stromae.
Nearest neighbors of : MGMT.
Nearest neighbors of : Robbie Williams.
Nearest neighbors of : TLC.
Nearest neighbors of : Dizzee Rascal.
[Found more than one matching artist. Other candidates: Florence + The Machine & Dizzee
Rascal]
Nearest neighbors of : Elton John.
[Found more than one matching artist. Other candidates: Elton John & Hans Zimmer & Lebo
M & South African Chorus, Elton John & Leon Russell]
Nearest neighbors of : Ludovico Einaudi.
Nearest neighbors of : Coldplay.
[Found more than one matching artist. Other candidates: Jay-Z & Coldplay, Coldplay/U2]
Nearest neighbors of : The Script.
Nearest neighbors of : Bee Gees.
Nearest neighbors of : Whigfield.
```

```
''
```

```
# Making a pandas data frame of our cosine recommendations list
cos_df = pd.concat(Cosine_recommendataions)
```

joint_df.pivot(index='name', columns='artistID', values='tagValue') Now we can remove duplicate names and then show the top 5 recommendations for our spotify data

```
# Order entries based on Cosine Score
cos_df = cos_df.sort_values(by=['cosine score'], ascending=False)

# Remove Duplicate Entries
cos_df = cos_df.drop_duplicates(subset=["names"], keep='last')
```

```
# Making a dataframe with the ttop 10 recommendations
results = cos_df.head(10)
```

```python
# Print results in Markdown format to show in a table
with pd.option_context('display.max_rows', None, 'display.max_columns', None): \
    print(results["names"].to_markdown(tablefmt="grid"))
```

```
+-------+---------------+
|       | names         |
+=======+===============+
|  3884 | Whigfield     |
+-------+---------------+
|  2117 | Dizzee Rascal |
+-------+---------------+
|  1391 | MGMT          |
+-------+---------------+
| 11026 | J. Cole       |
+-------+---------------+
|    90 | Fleetwood Mac |
+-------+---------------+
|  2587 | The Script    |
+-------+---------------+
|   186 | Elton John    |
+-------+---------------+
|  4773 | TLC           |
+-------+---------------+
|   655 | Stromae       |
+-------+---------------+
|  1958 | Aloe Blacc    |
+-------+---------------+
```

Even though the only available to test my spotify data on was data from 2011, I have to say I am quite pleased with the recommendations from our system.

Conclusions

Overall, this was certainly a very interesting project. Recommendation systems are a very useful tool in the Machine Learning world and it was very interesting to get hands on experience with them. Perosnally I think that the recommendation system worked well. As metioned above each recommendation that is given is completly subjective to the user involved. Interacting with my own personal data from Spotify was another interesting aspect of this assignment.

It is clear that there are many relationships in this data as seen in the cluster graphs. Although for now the information around what those clusters means is a mystery. This would have been able to backup my recommendations and make them explainable.

In my opinion the main struggle that the system had was with artists with many genres associated to them. This was seen with “Lady Gaga” as the recommendations were quite bad for her.

Apart from this I am happy with the recommendations of the model and think it works quite well!

Future Work

Due to the time onsuming nature of this project I didn’t get to full achieve everything I wanted to, mainly in the clustering area.

When the LastFM data becomes available for 2021 it would definitly be good to test the system again and see do the added artists provide better recommendations then the ones from 2011.

Interactive TSNE plots for clusters among artists genre embeddings to fully understand our data

It might be interesting to fully implement the clustering project work onto my own to get some concrete answeres with regards to my clusters in the future.

Another thing I would have liked to have done is compare recommendations for my entire spotify listening history. I requested my listening history from spotify but its takes up to 30 days for you to receive it and during my time completing this assignment I never received it.

Refrences

“1” Google Developers. 2021. Introduction | Recommendation Systems | Google Developers. [online] Available at: https://developers.google.com/machine-learning/recommendation/ [Accessed 2 December 2021].