Contents

Introduction

Recommender systems such as Spotify or Apple music are designed to offer suggestions to users for artists or songs that they may be interested in. These recommendations are based on previous listening history, or similar users listening habits. Spotify's recommender system uses Collaborative filtering, Natural Language Processing and audio modelling [ProducerHive, 2021]. Spotify leverages another algorithm named Bart which manages the home screen by ranking the cards and the shelves for the best engagement, while trying to provide explanations for the suggestions [McInerney *et al.*, 2018]. Netflix is another example of a widely used recommender system which presents recommendations while explaining the reason for that choice. For example, this can be due to previously watched films or popularity in that region. It does this through a variety of algorithms [Gomez-Uribe and Hunt, 2015]. Recommender systems aren't exclusive to music and streaming sites. They are also used across e-commerce, dating apps, news websites and research articles sites alike.

Approaches to Creating a Recommender System

Collaborative filtering and content based filtering are two starting points when it comes to building a recommender system [Koren *et al.*, 2009].

Collaborative filtering

This method was first proposed in 1992 [Goldberg *et al.*, 1992] and has become a widely used strategy for recommender systems ever since. This relies on the previous ratings of users and their similarity to other users in the past to paint a picture of their potential interests. This method doesn't require any domain information of the product itself as it relies on the premise that users who have similar tastes in music will be a good predictor for a unseen product based on their similarities in the past.

Content based filtering

This method pays more attention to the product itself and can often be of use when the item has a large amount of data available which is linked to the users profile. Products can then be recommended based on previous products that the user has liked.

Our approach

This project seeks to create a music recommender system based on collaborative filtering. The system is created using code available via Google's introduction to recommender systems using Google Colab *here*. It uses matrix factorisation to learn user and artist embeddings and uses stochastic gradient descent (SGD) to minimise the loss function.

Matrix Factorisation

Matrix factorisation models map users and items to a joint latent factor space of dimensionality such that user-item interactions are modeled as inner products in that space [Koren *et al.*, 2009]. In this project, matrix factorisaton characterises users and artists by vectors created using artist weighting patterns where a high correspondance will result in a recommendation.

Dataset

This dataset is from Last.FM and made available thanks to the 2nd International Workshop on Information Heterogeneity and Fusion in Recommender Systems [Cantador, 2011]. It contains 92,800 listening records from 1,892 users across 6 files and can be accessed *here*.

Data Exploration

We will explore the 6 files of the lastfm data set to further understand the data within and to investigate relationships between the users, artists and tags before we proceed with our recommender system.

Install dependencies

```
from __future__ import print_function
import seaborn as sns
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.compat.v1.disable_eager_execution()
```

Import data

Import data from our 6 separate files and do some preliminary analysis to better understand what information is contianed in this dataset.

We have 6 files:

- artists
- tags
- user_artists
- user_friends
- user_taggedartists-timestamp
- user_taggedartists

Artists

```
# Load artists
artists_cols = ['id', 'name', 'url', 'pictureURL']
artists = pd.read_csv('../Data/artists.dat', sep='	', names=artists_cols,
skiprows=1)
artists.head(5)
```

	id	name	url	pictureU
0	1	MALICE MIZER	http://www.last.fm/music/MALICE+MIZER	http://userser ak.last.fm/serve/252/10808
1	2	Diary of Dreams	http://www.last.fm/music/Diary+of+Dreams	http://userser ak.last.fm/serve/252/3052066
2	3	Carpathian Forest	http://www.last.fm/music/Carpathian+Forest	http://userser ak.last.fm/serve/252/402227
3	4	Moi dix Mois	http://www.last.fm/music/Moi+dix+Mois	http://userser ak.last.fm/serve/252/546978
4	5	Bella Morte	http://www.last.fm/music/Bella+Morte	http://userser ak.last.fm/serve/252/147890

```
artists.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17632 entries, 0 to 17631
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   id          17632 non-null  int64
 1   name        17632 non-null  object
 2   url         17632 non-null  object
 3   pictureURL  17188 non-null  object
dtypes: int64(1), object(3)
memory usage: 551.1+ KB
```

The picture URL contains some null values but we will not need this column for our analysis and so we can drop this column and the url column.

```
artists.drop('pictureURL', axis=1, inplace=True)
artists.drop('url', axis=1, inplace=True)
```

```
artists.head()
```

	id	name
0	1	MALICE MIZER
1	2	Diary of Dreams
2	3	Carpathian Forest
3	4	Moi dix Mois
4	5	Bella Morte

This dataframe contains one row for each of the 17,632 artist in this data set with their corresponding id

Tags

```
# Load tags
tags_cols = ['tagID', 'tagValue']
tags = pd.read_csv('../Data/tags.dat', sep='	', encoding='latin-1')
tags.head()
```

	tagID	tagValue
0	1	metal
1	2	alternative metal
2	3	goth rock
3	4	black metal
4	5	death metal

```
tags.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11946 entries, 0 to 11945
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   tagID     11946 non-null  int64
 1   tagValue  11946 non-null  object
dtypes: int64(1), object(1)
memory usage: 186.8+ KB
```

```
tags.describe()
```

	tagID
count	11946.000000
mean	6242.315336
std	3667.498057
min	1.000000
25%	3036.250000
50%	6210.500000
75%	9460.750000
max	12648.000000

```
tags1 = tags[tags['tagValue'].str.endswith('metal')]
tags1.value_counts()
```

```
tagID  tagValue
1      metal                    1
5999   french death metal       1
6318   canadian metal           1
6317   italian metal            1
6311   us metal                 1
                               ..
3135   melodic heavy metal      1
3133   extreme power metal      1
2956   communism death metal    1
2914   transmetal               1
12634  angry metal              1
Length: 306, dtype: int64
```

This dataframe contains one row for each of the 11,946 seperate tags that can be applied to each artist. As we can see from above there can be a wide variety of different genres. There are 306 different tags that all contain the word metal. We will need to be minfuk of this when doing analysis.

User Artists

```
# Load user-artists
user_artists_cols = ['userID', 'artistID', 'weight']
user_artists = pd.read_csv('../Data/user_artists.dat', sep='	')
user_artists.head()
```

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

```
user_artists.describe()
```

	userID	artistID	weight
count	92834.000000	92834.000000	92834.00000
mean	1037.010481	3331.123145	745.24393
std	610.870436	4383.590502	3751.32208
min	2.000000	1.000000	1.00000
25%	502.000000	436.000000	107.00000
50%	1029.000000	1246.000000	260.00000
75%	1568.000000	4350.000000	614.00000
max	2100.000000	18745.000000	352698.00000

```
user_artists.value_counts()
```

```
userID  artistID  weight
2       51        13883     1
1390    964       147       1
        863       687       1
        859       196       1
        709       201       1
                           ..
676     859       168       1
        856       127       1
        854       283       1
        841       198       1
2100    18730     263       1
Length: 92834, dtype: int64
```

The user artists contains users, the artist they listen to, and the weight which is proportional to how much they have listened to the artist. The weight value goes from 1-352,698 with an average weight of 745. Users may have a weighting for multiple artists.

(README.txt) 92834 user-listened artist relations: avg. 49.067 artists most listened by each user avg. 5.265 users who listened each artist

User Friends

```
# Load user-friends
user_friends_cols = ['userID', 'friendID']
user_friends = pd.read_csv('../Data/user_friends.dat', sep='	')
user_friends.head()
```

	userID	friendID
0	2	275
1	2	428
2	2	515
3	2	761
4	2	831

```
user_friends.describe()
```

	userID	friendID
count	25434.000000	25434.000000
mean	992.161437	992.161437
std	603.959049	603.959049
min	2.000000	2.000000
25%	441.000000	441.000000
50%	984.000000	984.000000
75%	1514.000000	1514.000000
max	2100.000000	2100.000000

This dataframe contains: 12717 bi-directional user friend relations, i.e. 25434 (user_i, user_j) pairs avg. 13.443 friend relations per user (taken from README.txt with the data)

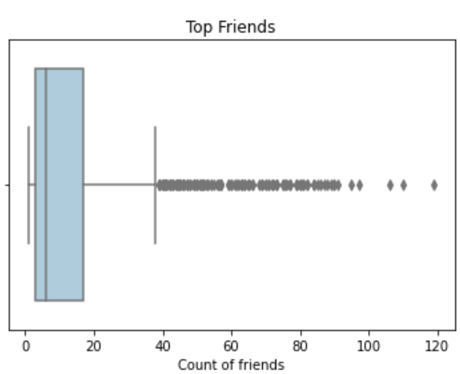
We will explore this data to see if any clear patterns emerge and to see what the distribution of friendship is like in this data.

```
top_friends = user_friends[['userID',
'friendID']].groupby('userID').count().reset_index()
top_friends.rename({'friendID':'count'}, axis=1, inplace=True)

top_friends = top_friends.sort_values('count', ascending=False)
top_friends
```

	userID	count
1394	1543	119
1164	1281	110
772	831	106
169	179	97
1359	1503	95
...	...	...
1693	1874	1
535	573	1
1214	1340	1
145	151	1
1718	1904	1

1892 rows × 2 columns

```
r = sns.color_palette('Paired')
sns.boxplot(x=top_friends['count'], palette=r)

plt.title('Top Friends')
plt.xlabel('Count of friends')
plt.show()
```



This boxplot gives us an idea of the relationships between friends. The majority of the users have < 20 friends however there appear to be some outliers who have upwards of 40 and as many as 119 friends. These are very influential users within this dataset

User Tagged Artists Timestamp

```
# Load user-tagged-artists-timestamps
user_tagged_artists_tstamp_cols = ['userID', 'artistID', 'tagID', 'timestamp']
user_tagged_artists_tstamp = pd.read_csv('../Data/user_taggedartists-timestamps.dat',
sep='	')
user_tagged_artists_tstamp.head()
```

	userID	artistID	tagID	timestamp
0	2	52	13	1238536800000
1	2	52	15	1238536800000
2	2	52	18	1238536800000
3	2	52	21	1238536800000
4	2	52	41	1238536800000

```
user_tagged_artists_tstamp.describe()
```

	userID	artistID	tagID	timestamp
count	186479.000000	186479.000000	186479.000000	1.864790e+05
mean	1035.600137	4375.845328	1439.582913	1.239204e+12
std	622.461272	4897.789595	2775.340279	4.299091e+10
min	2.000000	1.000000	1.000000	-4.287204e+11
25%	488.000000	686.000000	79.000000	1.209593e+12
50%	1021.000000	2203.000000	195.000000	1.243807e+12
75%	1624.000000	6714.000000	887.000000	1.275343e+12
max	2100.000000	18744.000000	12647.000000	1.304941e+12

User Tagged Artists

```
# Load user-tagged-artists
user_tagged_artists_cols = ['userID', 'artistID', 'tagID', 'day', 'month', 'year']
user_tagged_artists = pd.read_csv('../Data/user_taggedartists.dat', sep='	')
user_tagged_artists.head()
```

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

```
user_tagged_artists.describe()
```

	userID	artistID	tagID	day	montl
count	186479.000000	186479.000000	186479.000000	186479.000000	186479.00000(
mean	1035.600137	4375.845328	1439.582913	1.095566	6.52421
std	622.461272	4897.789595	2775.340279	0.712813	3.48685
min	2.000000	1.000000	1.000000	1.000000	1.00000(
25%	488.000000	686.000000	79.000000	1.000000	3.00000(
50%	1021.000000	2203.000000	195.000000	1.000000	7.00000(
75%	1624.000000	6714.000000	887.000000	1.000000	10.00000(
max	2100.000000	18744.000000	12647.000000	9.000000	12.00000(

The users tagged artist and users tagged artists timestamp are the same data across userID, artistID and tagID columns.

Data Manipulation and Visualisation

We will bring all of our data together and create some visualisations to better understand the data.

```
artists.tail()
```

	id	name
17627	18741	Diamanda Galás
17628	18742	Aya RL
17629	18743	Coptic Rain
17630	18744	Oz Alchemist
17631	18745	Grzegorz Tomczak

It is noted the index and the id don't add up at the end of the dataframe so we need to be aware of this when we come to the recommender system.

```
user_artists.tail()
```

	userID	artistID	weight
92829	2100	18726	337
92830	2100	18727	297
92831	2100	18728	281
92832	2100	18729	280
92833	2100	18730	263

```
tags.tail()
```

	tagID	tagValue
11941	12644	suomi
11942	12645	symbiosis
11943	12646	sverige
11944	12647	eire
11945	12648	electro latino

```
user_tagged_artists.tail()
```

	userID	artistID	tagID	day	month	year
186474	2100	16437	4	1	7	2010
186475	2100	16437	292	1	5	2010
186476	2100	16437	2087	1	7	2010
186477	2100	16437	2801	1	5	2010
186478	2100	16437	3335	1	7	2010

```
user_friends.tail()
```

	userID	friendID
25429	2099	1801
25430	2099	2006
25431	2099	2016
25432	2100	586
25433	2100	607

Merge Data / Visualisations

```
merged_uta_t = pd.merge(user_tagged_artists, tags, on = 'tagID')
```

```
merged_uta_t.head()
```

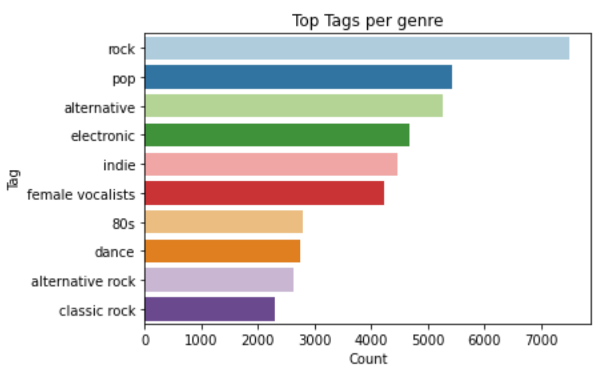
	userID	artistID	tagID	day	month	year	tagValue
0	2	52	13	1	4	2009	chillout
1	2	63	13	1	4	2009	chillout
2	2	73	13	1	4	2009	chillout
3	2	94	13	1	4	2009	chillout
4	2	6177	13	1	5	2009	chillout

We want to create a calculated field counting how many of each unique tag have been applied. This will show us the most listened to genres.

```
top_tag = merged_uta_t[['userID', 'tagValue']].groupby('tagValue').count().reset_index()
top_tag.rename({'userID':'count'}, axis=1, inplace=True)

#limit top tags to top 10
top_tag = top_tag.sort_values('count', ascending=False).head(10)
top_tag
```

	tagValue	count
7473	rock	7503
6802	pop	5418
441	alternative	5251
2709	electronic	4672
4393	indie	4458
3099	female vocalists	4228
174	80s	2791
2120	dance	2739
457	alternative rock	2631
1834	classic rock	2287

```
#Display top ten tags
r = sns.color_palette('Paired')
ax = sns.barplot(x='count', y='tagValue', data=top_tag,
            label="tagValue", palette=r)

plt.title('Top Tags per genre')
plt.ylabel('Tag')
plt.xlabel('Count')
plt.show()
```



We can see rock is the dominant genre. Rock is the number one choice, but as we alluded to earlier, there is also alternative rock and classic rock as the 9th and 10th most popular choices.

```
merged_ua_a = pd.merge(user_artists, artists, how='left', left_on='artistID',
right_on='id')
merged_ua_a.head()
```

	userID	artistID	weight	id	name
0	2	51	13883	51	Duran Duran
1	2	52	11690	52	Morcheeba
2	2	53	11351	53	Air
3	2	54	10300	54	Hooverphonic
4	2	55	8983	55	Kylie Minogue

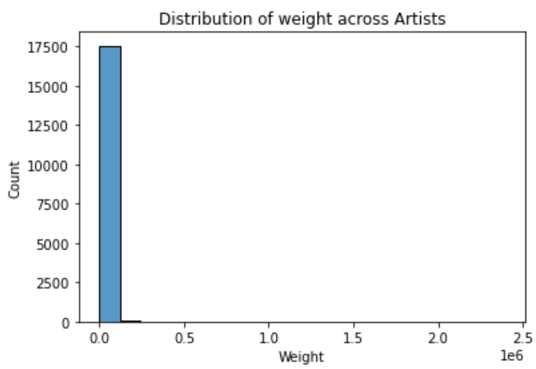
We now look at the different weights given to each of the artists indicating how much each artist has been listened to by users.

```
top_artist = merged_ua_a[['weight', 'name']].groupby('name').sum().reset_index()

top_artist
```

	name	weight
0	!!!	2826
1	!DISTAIN	1257
2	!deladap	65
3	#####	3707
4	#2 Orchestra	144
...	...	...
17627	R E D	373
17628	V a n e s s A	2172
17629	b o o g i e m a n	378
17630	b o r n	2287
17631	m a c h i n e	1338

17632 rows × 2 columns

```
#Display distribution of artists cumulative weights
sns.histplot(x='weight', data=top_artist, palette=r, bins=20)
plt.title('Distribution of weight across Artists')
plt.ylabel('Count')
plt.xlabel('Weight')
plt.show()
```

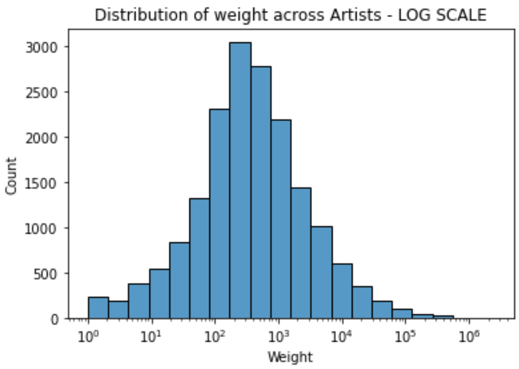



```
top_artist.describe()
```

	weight
count	1.763200e+04
mean	3.923774e+03
std	3.409934e+04
min	1.000000e+00
25%	1.130000e+02
50%	3.500000e+02
75%	1.234250e+03
max	2.393140e+06

This data is extremely right skewed with some large outliers (max value of 2,393,140) which is making it difficult to visualise the distribution. We will use log scale to enable us to do so.

```
#Display artists weights- log
sns.histplot(x='weight', data=top_artist, palette=r, log_scale=True, bins=20)
plt.title('Distribution of weight across Artists - LOG SCALE')
plt.ylabel('Count')
plt.xlabel('Weight')
plt.show()
```



There appears to be a normal distribution around the log scale of weights assigned to each artist.

Conclusion

We have completed our preliminary analysis of the data provided. We understand the weights value varies greatly across the data set. We also have an indication that there are a large number of highly linked users and friends across the data. We are hopeful that with all of this data we will be able to create a useful music recommender system.

Music Recommender System

This will detail all of the steps involved in creating this music recommender system. This code is based on the google colab notebook and educational code available at:

https://colab.research.google.com/github/google/eng-edu/blob/main/ml/recommendation-systems/recommendation-systems.ipynb?utm_source=ss-recommendation-systems&utm_campaign=colab-external&utm_medium=referral&utm_content=recommendation-systems#scrollTo=StMo4lDmLqpc

The notebook is split into 4 sections:

1. Building rating matrix/ Calculating error
2. Training the matrix factorisation model
3. Inspecting embeddings
4. Regularisation in matrix factorisation

Import required dependencies

```python
from __future__ import print_function
import seaborn as sns
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)
```

```
WARNING:tensorflow:From /Users/dockreg/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
```

1) Preliminaries

```python
artists = pd.read_csv('../Data/artists.dat', sep='	')
tags = pd.read_csv('../Data/tags.dat', sep='	', encoding='latin-1')
user_artists = pd.read_csv('../Data/user_artists.dat', sep='	')
user_friends = pd.read_csv('../Data/user_friends.dat', sep='	')
user_tagged_artists_tstamp = pd.read_csv('../Data/user_taggedartists-timestamps.dat',
sep='	')
user_tagged_artists = pd.read_csv('../Data/user_taggedartists.dat', sep='	')
```

We create a function to be used later for splitting the data in testing and training sets

```python
# Split the data into training and test sets.
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
```

Building matrix

We design a function that maps the `user_artists` data to a tensorflow sparsetensor representation. Most users will not have rated each artist so this is full of 0's and hence becomes a very large matrix. This function allows us to efficiently capture this data

```python
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()), len(artists['id'].unique())])
```

Calculating the error

The model approximates the ratings matrix by a low-rank product. We need a way to measure the approximation error. We'll start by using the Mean Squared Error of observed entries only. It is defined as \[\begin{split} \begin{align*} \text{MSE}(A, UV^\top) &= \frac{1}{|\Omega|}\sum_{(i, j) \in\Omega}{(A_{ij} - (UV^\top)_{ij})^2} \\ &= \frac{1}{|\Omega|}\sum_{(i, j) \in\Omega}{(A_{ij} - \langle U_i, V_j\rangle)^2} \end{align*} \end{split}\]
The function below is created to calculate this

```
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
```

2) Training the Matrix Factorization model

Collaborative Filtering Model (CF Model)

This class trains a matrix factorization model. Stochastic gradient descent is used in the function as the optimiser.

```
class CFModel(object):
    def __init__(self, embedding_vars, loss, metrics=None):
        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):
        return self._embeddings

    def train(self, num_iterations=100, learning_rate=1.0, plot_results=True,
            optimizer=tf.train.GradientDescentOptimizer):
        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
```

Build the model

We build the model that uses the `sparse_mean_square_error` function. We write a function that builds a `CFModel` by creating the embedding variables and the train and test losses.

```
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])
```

```
user_artists
```

	userID	artistID	weight
0	2	51	13883
1	2	52	11690
2	2	53	11351
3	2	54	10300
4	2	55	8983
...	...	...	...
92829	2100	18726	337
92830	2100	18727	297
92831	2100	18728	281
92832	2100	18729	280
92833	2100	18730	263

92834 rows × 3 columns

We convert our columns into the appropriate data type before running it through the model

Changing user id and artist id

We get errors when using the user data as is, as the user ID values and artist ID values are not in sequential order and are often higher than the index value so we will change them so they run from 0-1891, and 0-17631 respectively.

```
user_artists.userID.unique().astype(int).max()
```

```
2100
```

```
user_artists.artistID.unique().astype(int).max()
```

```
18745
```

```
def return_inverse(x):
    p = np.zeros(x.max()+1, dtype=bool)
    p[x] = 1

    p2 = np.empty(x.max()+1, dtype=np.uint64)
    c = p.sum()
    p2[p] = np.arange(c)
    out = p2[x]
    return out
```

```
inverse_user_id = return_inverse(user_artists.userID)
inverse_user_id
```

```
array([   0,    0,    0, ..., 1891, 1891, 1891], dtype=uint64)
```

```
inverse_artist_id = return_inverse(user_artists.artistID)
inverse_artist_id
```

```
array([   45,    46,    47, ..., 17617, 17618, 17619], dtype=uint64)
```

```
# Replace id columns
user_artists['userID'] = inverse_user_id
user_artists['artistID'] = inverse_artist_id
```

```
user_artists.describe()
```

	userID	artistID	weight
count	92834.000000	92834.000000	92834.00000
mean	944.222483	3235.736724	745.24393
std	546.751074	4197.216910	3751.32208
min	0.000000	0.000000	1.00000
25%	470.000000	430.000000	107.00000
50%	944.000000	1237.000000	260.00000
75%	1416.000000	4266.000000	614.00000
max	1891.000000	17631.000000	352698.00000

We can see the userID now has a max value of 1,891 and artistID has a max value of 17,631

Normalisation

The variety in weights are very large which will cause issues with our `CFModel` so we normalise these values between a value of 0-1

The code below can be uncommented to try a z score normalisation for different results

```
# copy the data
user_artists_norm = user_artists.copy()

# apply normalization techniques by Column weight
column = 'weight'
user_artists_norm[column] = (user_artists_norm[column] -
user_artists_norm[column].min()) / (user_artists_norm[column].max() -
user_artists_norm[column].min())

# z score normalisation
#user_artists_norm[column] = (user_artists_norm[column] -
user_artists_norm[column].mean()) / user_artists_norm[column].std()

# view normalized data
user_artists_norm.head()
```

	userID	artistID	weight
0	0	45	0.039360
1	0	46	0.033142
2	0	47	0.032181
3	0	48	0.029201
4	0	49	0.025467

Build the CF Model and train it

```
model = build_model(user_artists_norm, embedding_dim=30, init_stddev=0.5)
model.train(num_iterations=1000, learning_rate=10)
```

```
2021-12-02 09:25:17.668856: I tensorflow/core/platform/cpu_feature_guard.cc:151] This
TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use
the following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler
flags.
```

```
iteration 0: train_error=1.888031, test_error=1.851514
iteration 10: train_error=1.534868, test_error=1.621668
iteration 20: train_error=1.287163, test_error=1.459350
```

```
iteration 30: train_error=1.102333, test_error=1.337410
iteration 40: train_error=0.958629, test_error=1.241807
iteration 50: train_error=0.843567, test_error=1.164468
```

```
iteration 60: train_error=0.749380, test_error=1.100390
iteration 70: train_error=0.670939, test_error=1.046289
iteration 80: train_error=0.604698, test_error=0.999911
```

```
iteration 90: train_error=0.548113, test_error=0.959651
iteration 100: train_error=0.499309, test_error=0.924331
iteration 110: train_error=0.456866, test_error=0.893065
```

```
iteration 120: train_error=0.419688, test_error=0.865172
iteration 130: train_error=0.386916, test_error=0.840120
iteration 140: train_error=0.357864, test_error=0.817484
```

```
iteration 150: train_error=0.331980, test_error=0.796923
iteration 160: train_error=0.308812, test_error=0.778157
iteration 170: train_error=0.287990, test_error=0.760956
```

```
iteration 180: train_error=0.269204, test_error=0.745128
iteration 190: train_error=0.252195, test_error=0.730513
iteration 200: train_error=0.236745, test_error=0.716973
```

```
iteration 210: train_error=0.222668, test_error=0.704391
iteration 220: train_error=0.209807, test_error=0.692669
iteration 230: train_error=0.198024, test_error=0.681718
```

```
iteration 240: train_error=0.187204, test_error=0.671465
iteration 250: train_error=0.177243, test_error=0.661842
iteration 260: train_error=0.168053, test_error=0.652794
```

```
iteration 270: train_error=0.159558, test_error=0.644269
iteration 280: train_error=0.151688, test_error=0.636223
iteration 290: train_error=0.144383, test_error=0.628615
```

```
iteration 300: train_error=0.137592, test_error=0.621410
iteration 310: train_error=0.131268, test_error=0.614577
iteration 320: train_error=0.125367, test_error=0.608086
```

```
iteration 330: train_error=0.119855, test_error=0.601913
iteration 340: train_error=0.114696, test_error=0.596034
iteration 350: train_error=0.109863, test_error=0.590428
```

```
iteration 360: train_error=0.105327, test_error=0.585077
iteration 370: train_error=0.101066, test_error=0.579963
iteration 380: train_error=0.097057, test_error=0.575071
```

```
iteration 390: train_error=0.093281, test_error=0.570386
iteration 400: train_error=0.089720, test_error=0.565895
iteration 410: train_error=0.086358, test_error=0.561586
```

```
iteration 420: train_error=0.083182, test_error=0.557449
iteration 430: train_error=0.080176, test_error=0.553472
iteration 440: train_error=0.077330, test_error=0.549647
```

```
iteration 450: train_error=0.074631, test_error=0.545965
iteration 460: train_error=0.072071, test_error=0.542417
iteration 470: train_error=0.069640, test_error=0.538998
```

```
iteration 480: train_error=0.067329, test_error=0.535699
iteration 490: train_error=0.065131, test_error=0.532515
iteration 500: train_error=0.063038, test_error=0.529439
```

```
iteration 510: train_error=0.061043, test_error=0.526466
iteration 520: train_error=0.059141, test_error=0.523591
iteration 530: train_error=0.057326, test_error=0.520808
```

```
iteration 540: train_error=0.055593, test_error=0.518115
iteration 550: train_error=0.053937, test_error=0.515505
iteration 560: train_error=0.052353, test_error=0.512976
```

```
iteration 570: train_error=0.050837, test_error=0.510524
iteration 580: train_error=0.049385, test_error=0.508145
iteration 590: train_error=0.047995, test_error=0.505836
```

```
iteration 600: train_error=0.046661, test_error=0.503593
iteration 610: train_error=0.045382, test_error=0.501415
iteration 620: train_error=0.044154, test_error=0.499298
```

```
iteration 630: train_error=0.042975, test_error=0.497240
iteration 640: train_error=0.041843, test_error=0.495238
iteration 650: train_error=0.040753, test_error=0.493289
```

```
iteration 660: train_error=0.039706, test_error=0.491393
iteration 670: train_error=0.038697, test_error=0.489547
iteration 680: train_error=0.037727, test_error=0.487748
```

```
iteration 690: train_error=0.036791, test_error=0.485995
iteration 700: train_error=0.035890, test_error=0.484287
iteration 710: train_error=0.035021, test_error=0.482621
```

```
iteration 720: train_error=0.034183, test_error=0.480996
iteration 730: train_error=0.033374, test_error=0.479410
iteration 740: train_error=0.032593, test_error=0.477863
```

```
iteration 750: train_error=0.031839, test_error=0.476353
iteration 760: train_error=0.031111, test_error=0.474877
iteration 770: train_error=0.030406, test_error=0.473436
```

```
iteration 780: train_error=0.029725, test_error=0.472028
iteration 790: train_error=0.029067, test_error=0.470652
```

```
iteration 800: train_error=0.028429, test_error=0.469307
iteration 810: train_error=0.027812, test_error=0.467992
```

```
iteration 820: train_error=0.027215, test_error=0.466705
iteration 830: train_error=0.026636, test_error=0.465446
```

```
iteration 840: train_error=0.026075, test_error=0.464215
iteration 850: train_error=0.025532, test_error=0.463009
```

```
iteration 860: train_error=0.025005, test_error=0.461829
iteration 870: train_error=0.024493, test_error=0.460673
```

```
iteration 880: train_error=0.023997, test_error=0.459541
iteration 890: train_error=0.023516, test_error=0.458432
```

```
iteration 900: train_error=0.023049, test_error=0.457346
iteration 910: train_error=0.022595, test_error=0.456281
```

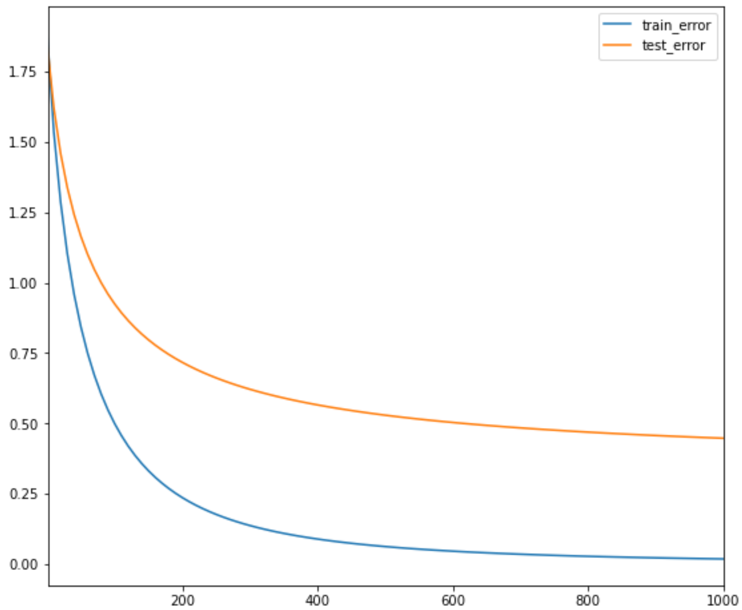
```
iteration 920: train_error=0.022154, test_error=0.455237
iteration 930: train_error=0.021726, test_error=0.454214
iteration 940: train_error=0.021310, test_error=0.453211
```

```
iteration 950: train_error=0.020906, test_error=0.452227
iteration 960: train_error=0.020512, test_error=0.451261
```

```
iteration 970: train_error=0.020130, test_error=0.450314
iteration 980: train_error=0.019758, test_error=0.449385
iteration 990: train_error=0.019395, test_error=0.448472
```

```
iteration 1000: train_error=0.019043, test_error=0.447577
```

```
[{'train_error': 0.01904296, 'test_error': 0.44757697}]
```



3) Inspecting the Embeddings

We look at the recommendations of the system using the dot product and cosine similarity which are two different similarity measures. We create a nearest neighbours function to recommend similar artists.

```
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 artists 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))
```

Testing

We input an artist to see what recommendations our system returns to us

```
artist_neighbors(model, "Johnny Cash", DOT)
artist_neighbors(model, "Johnny Cash", COSINE)
```

```
Nearest neighbors of : Johnny Cash.
[Found more than one matching artist. Other candidates: Johnny Cash & Willie Nelson]
```

	dot score	names
5638	3.441453	Boy Talks Trash
17278	3.338759	Lava
712	3.249975	Johnny Cash
9645	3.093319	Die Fantastischen Vier
2129	3.087614	Георг Корг
6110	3.066176	Negative

```
Nearest neighbors of : Johnny Cash.
[Found more than one matching artist. Other candidates: Johnny Cash & Willie Nelson]
```

	cosine score	names
712	1.000000	Johnny Cash
17278	0.640777	Lava
17086	0.636165	The Post-Modern Cliche
5638	0.619617	Boy Talks Trash
13065	0.603140	Konami
4114	0.594109	The Suicide Machines

These results are interesting but it seems our system could be improved upon.

Model initialisation

It seems the initialisation parameters may play a factor in the results of our system as artists with few ratings may have had their embeddings initialised with a high norm. We use regularisation to combat this by adjusting the value of `init_stdev` (previously at 0.5 now changed to 0.05)

```python
# Solution
model_lowinit = build_model(user_artists_norm, embedding_dim=30, init_stddev=0.05)
model_lowinit.train(num_iterations=1000, learning_rate=10.)
artist_neighbors(model_lowinit, "Johnny Cash", DOT)
artist_neighbors(model_lowinit, "Johnny Cash", COSINE)
#movie_embedding_norm([model, model_lowinit])
```

```
iteration 0: train_error=0.000300, test_error=0.000366
iteration 10: train_error=0.000299, test_error=0.000366
iteration 20: train_error=0.000299, test_error=0.000366
```

```
iteration 30: train_error=0.000298, test_error=0.000365
iteration 40: train_error=0.000298, test_error=0.000365
iteration 50: train_error=0.000297, test_error=0.000365
```

```
iteration 60: train_error=0.000297, test_error=0.000365
iteration 70: train_error=0.000296, test_error=0.000364
iteration 80: train_error=0.000296, test_error=0.000364
```

```
iteration 90: train_error=0.000295, test_error=0.000364
iteration 100: train_error=0.000295, test_error=0.000364
iteration 110: train_error=0.000294, test_error=0.000363
```

```
iteration 120: train_error=0.000294, test_error=0.000363
iteration 130: train_error=0.000293, test_error=0.000363
iteration 140: train_error=0.000293, test_error=0.000363
```

```
iteration 150: train_error=0.000292, test_error=0.000362
iteration 160: train_error=0.000292, test_error=0.000362
iteration 170: train_error=0.000291, test_error=0.000362
```

```
iteration 180: train_error=0.000291, test_error=0.000362
iteration 190: train_error=0.000290, test_error=0.000361
iteration 200: train_error=0.000290, test_error=0.000361
```

```
iteration 210: train_error=0.000289, test_error=0.000361
iteration 220: train_error=0.000289, test_error=0.000361
iteration 230: train_error=0.000288, test_error=0.000360
```

```
iteration 240: train_error=0.000288, test_error=0.000360
iteration 250: train_error=0.000288, test_error=0.000360
iteration 260: train_error=0.000287, test_error=0.000360
```

```
iteration 270: train_error=0.000287, test_error=0.000359
iteration 280: train_error=0.000286, test_error=0.000359
iteration 290: train_error=0.000286, test_error=0.000359
```

```
iteration 300: train_error=0.000285, test_error=0.000359
iteration 310: train_error=0.000285, test_error=0.000358
iteration 320: train_error=0.000284, test_error=0.000358
```

```
iteration 330: train_error=0.000284, test_error=0.000358
iteration 340: train_error=0.000283, test_error=0.000358
iteration 350: train_error=0.000283, test_error=0.000357
```

```
iteration 360: train_error=0.000283, test_error=0.000357
iteration 370: train_error=0.000282, test_error=0.000357
iteration 380: train_error=0.000282, test_error=0.000357
```

```
iteration 390: train_error=0.000281, test_error=0.000356
iteration 400: train_error=0.000281, test_error=0.000356
iteration 410: train_error=0.000280, test_error=0.000356
```

```
iteration 420: train_error=0.000280, test_error=0.000356
iteration 430: train_error=0.000280, test_error=0.000356
```

```
iteration 440: train_error=0.000279, test_error=0.000355
iteration 450: train_error=0.000279, test_error=0.000355
```

```
iteration 460: train_error=0.000278, test_error=0.000355
iteration 470: train_error=0.000278, test_error=0.000355
```

```
iteration 480: train_error=0.000278, test_error=0.000354
iteration 490: train_error=0.000277, test_error=0.000354
iteration 500: train_error=0.000277, test_error=0.000354
```

```
iteration 510: train_error=0.000276, test_error=0.000354
iteration 520: train_error=0.000276, test_error=0.000354
iteration 530: train_error=0.000275, test_error=0.000353
```

```
iteration 540: train_error=0.000275, test_error=0.000353
iteration 550: train_error=0.000275, test_error=0.000353
iteration 560: train_error=0.000274, test_error=0.000353
```

```
iteration 570: train_error=0.000274, test_error=0.000353
iteration 580: train_error=0.000273, test_error=0.000352
iteration 590: train_error=0.000273, test_error=0.000352
```

```
iteration 600: train_error=0.000273, test_error=0.000352
iteration 610: train_error=0.000272, test_error=0.000352
```

```
iteration 620: train_error=0.000272, test_error=0.000351
iteration 630: train_error=0.000271, test_error=0.000351
```

```
iteration 640: train_error=0.000271, test_error=0.000351
iteration 650: train_error=0.000271, test_error=0.000351
```

```
iteration 660: train_error=0.000270, test_error=0.000351
iteration 670: train_error=0.000270, test_error=0.000350
```

```
iteration 680: train_error=0.000269, test_error=0.000350
iteration 690: train_error=0.000269, test_error=0.000350
```

```
iteration 700: train_error=0.000269, test_error=0.000350
iteration 710: train_error=0.000268, test_error=0.000350
```

```
iteration 720: train_error=0.000268, test_error=0.000349
iteration 730: train_error=0.000268, test_error=0.000349
```

```
iteration 740: train_error=0.000267, test_error=0.000349
iteration 750: train_error=0.000267, test_error=0.000349
iteration 760: train_error=0.000266, test_error=0.000349
```

```
iteration 770: train_error=0.000266, test_error=0.000348
iteration 780: train_error=0.000266, test_error=0.000348
iteration 790: train_error=0.000265, test_error=0.000348
```

```
iteration 800: train_error=0.000265, test_error=0.000348
iteration 810: train_error=0.000265, test_error=0.000348
iteration 820: train_error=0.000264, test_error=0.000347
```

```
iteration 830: train_error=0.000264, test_error=0.000347
iteration 840: train_error=0.000263, test_error=0.000347
```

```
iteration 850: train_error=0.000263, test_error=0.000347
iteration 860: train_error=0.000263, test_error=0.000347
iteration 870: train_error=0.000262, test_error=0.000346
```

```
iteration 880: train_error=0.000262, test_error=0.000346
iteration 890: train_error=0.000262, test_error=0.000346
```

```
iteration 900: train_error=0.000261, test_error=0.000346
iteration 910: train_error=0.000261, test_error=0.000346
iteration 920: train_error=0.000261, test_error=0.000346
```

```
 iteration 930: train_error=0.000260, test_error=0.000345
 iteration 940: train_error=0.000260, test_error=0.000345
 iteration 950: train_error=0.000260, test_error=0.000345
```

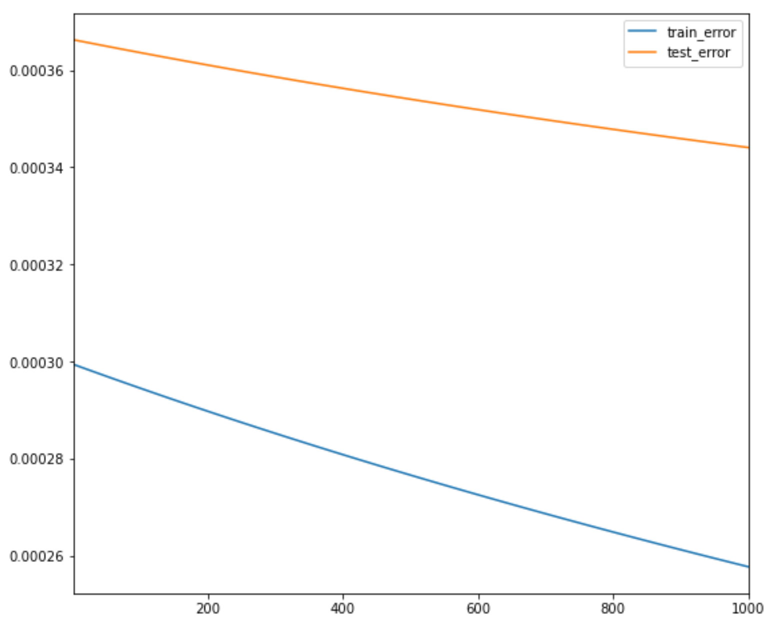
```
 iteration 960: train_error=0.000259, test_error=0.000345
 iteration 970: train_error=0.000259, test_error=0.000345
 iteration 980: train_error=0.000259, test_error=0.000344
```

```
 iteration 990: train_error=0.000258, test_error=0.000344
 iteration 1000: train_error=0.000258, test_error=0.000344Nearest neighbors of : Johnny
Cash.
[Found more than one matching artist. Other candidates: Johnny Cash & Willie Nelson]
```

	dot score	names
712	0.055784	Johnny Cash
16968	0.051251	Riceboy Sleeps
13790	0.042951	Antonello Venditti
3021	0.042440	Polar Bear Club
10392	0.040910	Colette Carr
6218	0.039689	TV-2

```
Nearest neighbors of : Johnny Cash.
[Found more than one matching artist. Other candidates: Johnny Cash & Willie Nelson]
```

	cosine score	names
712	1.000000	Johnny Cash
6218	0.598475	TV-2
3021	0.596209	Polar Bear Club
4737	0.592499	Face to Face
16230	0.575613	The Recoys
10392	0.570717	Colette Carr



4) Regularization In Matrix Factorization

In the code above, loss was defined as the mean squared error on the observed part of the rating matrix. This can often cause issues when the model does not learn how to place the embeddings of irrelevant artists. This is called *folding*.

We add some regularization terms to deal with this problem:

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

Total loss can now be calculated as: $\(\frac{1}{|\Omega|}\sum_{(i, j) \in \Omega} (A_{ij} - \langle U_i, V_j\rangle)^2 + \lambda _r r(U, V) + \lambda_g g(U, V) \)$

```
def gravity(U, V):
  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])
```

We build the regularised model and observe the results

```
reg_model = build_regularized_model(
    user_artists_norm, regularization_coeff=0.1, gravity_coeff=1.0, embedding_dim=35,
    init_stddev=.05)
reg_model.train(num_iterations=2000, learning_rate=20.)
```

```
 iteration 0: train_error_observed=0.000342, test_error_observed=0.000263,
observed_loss=0.000342, regularization_loss=0.017476, gravity_loss=0.000218
 iteration 10: train_error_observed=0.000331, test_error_observed=0.000253,
observed_loss=0.000331, regularization_loss=0.017056, gravity_loss=0.000208
```

```
 iteration 20: train_error_observed=0.000320, test_error_observed=0.000243,
observed_loss=0.000320, regularization_loss=0.016653, gravity_loss=0.000198
 iteration 30: train_error_observed=0.000310, test_error_observed=0.000233,
observed_loss=0.000310, regularization_loss=0.016265, gravity_loss=0.000188
```

```
 iteration 40: train_error_observed=0.000300, test_error_observed=0.000224,
observed_loss=0.000300, regularization_loss=0.015893, gravity_loss=0.000179
 iteration 50: train_error_observed=0.000291, test_error_observed=0.000216,
observed_loss=0.000291, regularization_loss=0.015536, gravity_loss=0.000171
```

```
 iteration 60: train_error_observed=0.000283, test_error_observed=0.000208,
observed_loss=0.000283, regularization_loss=0.015193, gravity_loss=0.000162
 iteration 70: train_error_observed=0.000274, test_error_observed=0.000200,
observed_loss=0.000274, regularization_loss=0.014864, gravity_loss=0.000155
```

```
 iteration 80: train_error_observed=0.000267, test_error_observed=0.000193,
observed_loss=0.000267, regularization_loss=0.014547, gravity_loss=0.000147
 iteration 90: train_error_observed=0.000259, test_error_observed=0.000186,
observed_loss=0.000259, regularization_loss=0.014243, gravity_loss=0.000140
```

```
 iteration 100: train_error_observed=0.000252, test_error_observed=0.000179,
observed_loss=0.000252, regularization_loss=0.013950, gravity_loss=0.000133
 iteration 110: train_error_observed=0.000246, test_error_observed=0.000173,
observed_loss=0.000246, regularization_loss=0.013669, gravity_loss=0.000127
```

```
 iteration 120: train_error_observed=0.000240, test_error_observed=0.000167,
observed_loss=0.000240, regularization_loss=0.013398, gravity_loss=0.000121
 iteration 130: train_error_observed=0.000234, test_error_observed=0.000162,
observed_loss=0.000234, regularization_loss=0.013138, gravity_loss=0.000115
```

```
 iteration 140: train_error_observed=0.000228, test_error_observed=0.000156,
observed_loss=0.000228, regularization_loss=0.012888, gravity_loss=0.000110
 iteration 150: train_error_observed=0.000223, test_error_observed=0.000151,
observed_loss=0.000223, regularization_loss=0.012647, gravity_loss=0.000105
```

```
 iteration 160: train_error_observed=0.000218, test_error_observed=0.000146,
observed_loss=0.000218, regularization_loss=0.012415, gravity_loss=0.000100
 iteration 170: train_error_observed=0.000213, test_error_observed=0.000142,
observed_loss=0.000213, regularization_loss=0.012192, gravity_loss=0.000095
```

```
 iteration 180: train_error_observed=0.000208, test_error_observed=0.000137,
observed_loss=0.000208, regularization_loss=0.011977, gravity_loss=0.000090
 iteration 190: train_error_observed=0.000204, test_error_observed=0.000133,
observed_loss=0.000204, regularization_loss=0.011771, gravity_loss=0.000086
```

```
 iteration 200: train_error_observed=0.000200, test_error_observed=0.000129,
observed_loss=0.000200, regularization_loss=0.011571, gravity_loss=0.000082
 iteration 210: train_error_observed=0.000196, test_error_observed=0.000126,
observed_loss=0.000196, regularization_loss=0.011379, gravity_loss=0.000078
```

```
 iteration 220: train_error_observed=0.000192, test_error_observed=0.000122,
observed_loss=0.000192, regularization_loss=0.011194, gravity_loss=0.000074
 iteration 230: train_error_observed=0.000189, test_error_observed=0.000119,
observed_loss=0.000189, regularization_loss=0.011016, gravity_loss=0.000071
```

```
 iteration 240: train_error_observed=0.000185, test_error_observed=0.000115,
observed_loss=0.000185, regularization_loss=0.010844, gravity_loss=0.000067
 iteration 250: train_error_observed=0.000182, test_error_observed=0.000112,
observed_loss=0.000182, regularization_loss=0.010678, gravity_loss=0.000064
```

```
 iteration 260: train_error_observed=0.000179, test_error_observed=0.000109,
observed_loss=0.000179, regularization_loss=0.010517, gravity_loss=0.000061
 iteration 270: train_error_observed=0.000176, test_error_observed=0.000107,
observed_loss=0.000176, regularization_loss=0.010363, gravity_loss=0.000058
```

```
 iteration 280: train_error_observed=0.000174, test_error_observed=0.000104,
observed_loss=0.000174, regularization_loss=0.010214, gravity_loss=0.000055
 iteration 290: train_error_observed=0.000171, test_error_observed=0.000102,
observed_loss=0.000171, regularization_loss=0.010070, gravity_loss=0.000053
```

```
 iteration 300: train_error_observed=0.000169, test_error_observed=0.000099,
observed_loss=0.000169, regularization_loss=0.009930, gravity_loss=0.000050
 iteration 310: train_error_observed=0.000166, test_error_observed=0.000097,
observed_loss=0.000166, regularization_loss=0.009796, gravity_loss=0.000048
```

```
 iteration 320: train_error_observed=0.000164, test_error_observed=0.000095,
observed_loss=0.000164, regularization_loss=0.009666, gravity_loss=0.000046
 iteration 330: train_error_observed=0.000162, test_error_observed=0.000093,
observed_loss=0.000162, regularization_loss=0.009540, gravity_loss=0.000043
```

```
 iteration 340: train_error_observed=0.000160, test_error_observed=0.000091,
observed_loss=0.000160, regularization_loss=0.009419, gravity_loss=0.000041
 iteration 350: train_error_observed=0.000158, test_error_observed=0.000089,
observed_loss=0.000158, regularization_loss=0.009301, gravity_loss=0.000039
```

```
 iteration 360: train_error_observed=0.000156, test_error_observed=0.000087,
observed_loss=0.000156, regularization_loss=0.009187, gravity_loss=0.000038
 iteration 370: train_error_observed=0.000155, test_error_observed=0.000085,
observed_loss=0.000155, regularization_loss=0.009077, gravity_loss=0.000036
```

```
 iteration 380: train_error_observed=0.000153, test_error_observed=0.000084,
observed_loss=0.000153, regularization_loss=0.008970, gravity_loss=0.000034
 iteration 390: train_error_observed=0.000151, test_error_observed=0.000082,
observed_loss=0.000151, regularization_loss=0.008867, gravity_loss=0.000032
```

```
 iteration 400: train_error_observed=0.000150, test_error_observed=0.000081,
observed_loss=0.000150, regularization_loss=0.008766, gravity_loss=0.000031
 iteration 410: train_error_observed=0.000149, test_error_observed=0.000079,
observed_loss=0.000149, regularization_loss=0.008669, gravity_loss=0.000029
```

```
 iteration 420: train_error_observed=0.000147, test_error_observed=0.000078,
observed_loss=0.000147, regularization_loss=0.008575, gravity_loss=0.000028
 iteration 430: train_error_observed=0.000146, test_error_observed=0.000077,
observed_loss=0.000146, regularization_loss=0.008484, gravity_loss=0.000027
```

```
 iteration 440: train_error_observed=0.000145, test_error_observed=0.000076,
observed_loss=0.000145, regularization_loss=0.008395, gravity_loss=0.000025
 iteration 450: train_error_observed=0.000144, test_error_observed=0.000075,
observed_loss=0.000144, regularization_loss=0.008309, gravity_loss=0.000024
```

```
 iteration 460: train_error_observed=0.000143, test_error_observed=0.000073,
observed_loss=0.000143, regularization_loss=0.008225, gravity_loss=0.000023
 iteration 470: train_error_observed=0.000141, test_error_observed=0.000072,
observed_loss=0.000141, regularization_loss=0.008144, gravity_loss=0.000022
```

```
 iteration 480: train_error_observed=0.000141, test_error_observed=0.000071,
observed_loss=0.000141, regularization_loss=0.008065, gravity_loss=0.000021
 iteration 490: train_error_observed=0.000140, test_error_observed=0.000071,
observed_loss=0.000140, regularization_loss=0.007988, gravity_loss=0.000020
```

```
 iteration 500: train_error_observed=0.000139, test_error_observed=0.000070,
observed_loss=0.000139, regularization_loss=0.007913, gravity_loss=0.000019
 iteration 510: train_error_observed=0.000138, test_error_observed=0.000069,
observed_loss=0.000138, regularization_loss=0.007840, gravity_loss=0.000018
```

```
 iteration 520: train_error_observed=0.000137, test_error_observed=0.000068,
observed_loss=0.000137, regularization_loss=0.007769, gravity_loss=0.000017
 iteration 530: train_error_observed=0.000136, test_error_observed=0.000067,
observed_loss=0.000136, regularization_loss=0.007700, gravity_loss=0.000016
```

```
 iteration 540: train_error_observed=0.000136, test_error_observed=0.000067,
observed_loss=0.000136, regularization_loss=0.007633, gravity_loss=0.000016
 iteration 550: train_error_observed=0.000135, test_error_observed=0.000066,
observed_loss=0.000135, regularization_loss=0.007568, gravity_loss=0.000015
```

```
 iteration 560: train_error_observed=0.000134, test_error_observed=0.000065,
observed_loss=0.000134, regularization_loss=0.007504, gravity_loss=0.000014
 iteration 570: train_error_observed=0.000134, test_error_observed=0.000065,
observed_loss=0.000134, regularization_loss=0.007442, gravity_loss=0.000014
```

```
 iteration 580: train_error_observed=0.000133, test_error_observed=0.000064,
observed_loss=0.000133, regularization_loss=0.007381, gravity_loss=0.000013
 iteration 590: train_error_observed=0.000132, test_error_observed=0.000063,
observed_loss=0.000132, regularization_loss=0.007321, gravity_loss=0.000012
```

```
 iteration 600: train_error_observed=0.000132, test_error_observed=0.000063,
observed_loss=0.000132, regularization_loss=0.007263, gravity_loss=0.000012
 iteration 610: train_error_observed=0.000131, test_error_observed=0.000062,
observed_loss=0.000131, regularization_loss=0.007207, gravity_loss=0.000011
```

```
 iteration 620: train_error_observed=0.000131, test_error_observed=0.000062,
observed_loss=0.000131, regularization_loss=0.007151, gravity_loss=0.000011
 iteration 630: train_error_observed=0.000130, test_error_observed=0.000061,
observed_loss=0.000130, regularization_loss=0.007097, gravity_loss=0.000010
```

```
 iteration 640: train_error_observed=0.000130, test_error_observed=0.000061,
observed_loss=0.000130, regularization_loss=0.007044, gravity_loss=0.000010
 iteration 650: train_error_observed=0.000130, test_error_observed=0.000061,
observed_loss=0.000130, regularization_loss=0.006992, gravity_loss=0.000009
```

```
 iteration 660: train_error_observed=0.000129, test_error_observed=0.000060,
observed_loss=0.000129, regularization_loss=0.006942, gravity_loss=0.000009
 iteration 670: train_error_observed=0.000129, test_error_observed=0.000060,
observed_loss=0.000129, regularization_loss=0.006892, gravity_loss=0.000008
```

```
 iteration 680: train_error_observed=0.000128, test_error_observed=0.000059,
observed_loss=0.000128, regularization_loss=0.006843, gravity_loss=0.000008
 iteration 690: train_error_observed=0.000128, test_error_observed=0.000059,
observed_loss=0.000128, regularization_loss=0.006796, gravity_loss=0.000008
```

```
 iteration 700: train_error_observed=0.000128, test_error_observed=0.000059,
observed_loss=0.000128, regularization_loss=0.006749, gravity_loss=0.000007
 iteration 710: train_error_observed=0.000128, test_error_observed=0.000058,
observed_loss=0.000128, regularization_loss=0.006703, gravity_loss=0.000007
```

```
 iteration 720: train_error_observed=0.000127, test_error_observed=0.000058,
observed_loss=0.000127, regularization_loss=0.006658, gravity_loss=0.000007
 iteration 730: train_error_observed=0.000127, test_error_observed=0.000058,
observed_loss=0.000127, regularization_loss=0.006614, gravity_loss=0.000006
```

```
 iteration 740: train_error_observed=0.000127, test_error_observed=0.000058,
observed_loss=0.000127, regularization_loss=0.006571, gravity_loss=0.000006
 iteration 750: train_error_observed=0.000126, test_error_observed=0.000057,
observed_loss=0.000126, regularization_loss=0.006528, gravity_loss=0.000006
```

```
 iteration 760: train_error_observed=0.000126, test_error_observed=0.000057,
observed_loss=0.000126, regularization_loss=0.006486, gravity_loss=0.000005
 iteration 770: train_error_observed=0.000126, test_error_observed=0.000057,
observed_loss=0.000126, regularization_loss=0.006445, gravity_loss=0.000005
```

```
 iteration 780: train_error_observed=0.000126, test_error_observed=0.000057,
observed_loss=0.000126, regularization_loss=0.006405, gravity_loss=0.000005
 iteration 790: train_error_observed=0.000126, test_error_observed=0.000056,
observed_loss=0.000126, regularization_loss=0.006365, gravity_loss=0.000005
```

```
 iteration 800: train_error_observed=0.000125, test_error_observed=0.000056,
observed_loss=0.000125, regularization_loss=0.006326, gravity_loss=0.000004
 iteration 810: train_error_observed=0.000125, test_error_observed=0.000056,
observed_loss=0.000125, regularization_loss=0.006287, gravity_loss=0.000004
```

```
 iteration 820: train_error_observed=0.000125, test_error_observed=0.000056,
observed_loss=0.000125, regularization_loss=0.006250, gravity_loss=0.000004
 iteration 830: train_error_observed=0.000125, test_error_observed=0.000056,
observed_loss=0.000125, regularization_loss=0.006212, gravity_loss=0.000004
```

```
 iteration 840: train_error_observed=0.000125, test_error_observed=0.000055,
observed_loss=0.000125, regularization_loss=0.006175, gravity_loss=0.000004
 iteration 850: train_error_observed=0.000125, test_error_observed=0.000055,
observed_loss=0.000125, regularization_loss=0.006139, gravity_loss=0.000004
```

```
 iteration 860: train_error_observed=0.000124, test_error_observed=0.000055,
observed_loss=0.000124, regularization_loss=0.006104, gravity_loss=0.000003
 iteration 870: train_error_observed=0.000124, test_error_observed=0.000055,
observed_loss=0.000124, regularization_loss=0.006068, gravity_loss=0.000003
```

```
 iteration 880: train_error_observed=0.000124, test_error_observed=0.000055,
observed_loss=0.000124, regularization_loss=0.006034, gravity_loss=0.000003
 iteration 890: train_error_observed=0.000124, test_error_observed=0.000055,
observed_loss=0.000124, regularization_loss=0.005999, gravity_loss=0.000003
```

```
 iteration 900: train_error_observed=0.000124, test_error_observed=0.000055,
observed_loss=0.000124, regularization_loss=0.005966, gravity_loss=0.000003
 iteration 910: train_error_observed=0.000124, test_error_observed=0.000054,
observed_loss=0.000124, regularization_loss=0.005932, gravity_loss=0.000003
```

```
 iteration 920: train_error_observed=0.000124, test_error_observed=0.000054,
observed_loss=0.000124, regularization_loss=0.005899, gravity_loss=0.000003
 iteration 930: train_error_observed=0.000124, test_error_observed=0.000054,
observed_loss=0.000124, regularization_loss=0.005867, gravity_loss=0.000002
```

```
 iteration 940: train_error_observed=0.000124, test_error_observed=0.000054,
observed_loss=0.000124, regularization_loss=0.005835, gravity_loss=0.000002
 iteration 950: train_error_observed=0.000123, test_error_observed=0.000054,
observed_loss=0.000123, regularization_loss=0.005803, gravity_loss=0.000002
```

```
 iteration 960: train_error_observed=0.000123, test_error_observed=0.000054,
observed_loss=0.000123, regularization_loss=0.005772, gravity_loss=0.000002
 iteration 970: train_error_observed=0.000123, test_error_observed=0.000054,
observed_loss=0.000123, regularization_loss=0.005741, gravity_loss=0.000002
```

```
 iteration 980: train_error_observed=0.000123, test_error_observed=0.000054,
observed_loss=0.000123, regularization_loss=0.005710, gravity_loss=0.000002
 iteration 990: train_error_observed=0.000123, test_error_observed=0.000054,
observed_loss=0.000123, regularization_loss=0.005680, gravity_loss=0.000002
```

```
 iteration 1000: train_error_observed=0.000123, test_error_observed=0.000054,
observed_loss=0.000123, regularization_loss=0.005650, gravity_loss=0.000002
 iteration 1010: train_error_observed=0.000123, test_error_observed=0.000054,
observed_loss=0.000123, regularization_loss=0.005620, gravity_loss=0.000002
```

```
 iteration 1020: train_error_observed=0.000123, test_error_observed=0.000054,
observed_loss=0.000123, regularization_loss=0.005591, gravity_loss=0.000002
 iteration 1030: train_error_observed=0.000123, test_error_observed=0.000053,
observed_loss=0.000123, regularization_loss=0.005561, gravity_loss=0.000001
```

```
 iteration 1040: train_error_observed=0.000123, test_error_observed=0.000053,
observed_loss=0.000123, regularization_loss=0.005533, gravity_loss=0.000001
 iteration 1050: train_error_observed=0.000123, test_error_observed=0.000053,
observed_loss=0.000123, regularization_loss=0.005504, gravity_loss=0.000001
```

```
 iteration 1060: train_error_observed=0.000123, test_error_observed=0.000053,
observed_loss=0.000123, regularization_loss=0.005476, gravity_loss=0.000001
 iteration 1070: train_error_observed=0.000123, test_error_observed=0.000053,
observed_loss=0.000123, regularization_loss=0.005448, gravity_loss=0.000001
```

```
 iteration 1080: train_error_observed=0.000123, test_error_observed=0.000053,
observed_loss=0.000123, regularization_loss=0.005420, gravity_loss=0.000001
 iteration 1090: train_error_observed=0.000123, test_error_observed=0.000053,
observed_loss=0.000123, regularization_loss=0.005393, gravity_loss=0.000001
```

```
 iteration 1100: train_error_observed=0.000123, test_error_observed=0.000053,
observed_loss=0.000123, regularization_loss=0.005366, gravity_loss=0.000001
 iteration 1110: train_error_observed=0.000123, test_error_observed=0.000053,
observed_loss=0.000123, regularization_loss=0.005339, gravity_loss=0.000001
```

```
 iteration 1120: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005312, gravity_loss=0.000001
 iteration 1130: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005286, gravity_loss=0.000001
```

```
 iteration 1140: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005259, gravity_loss=0.000001
 iteration 1150: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005233, gravity_loss=0.000001
```

```
 iteration 1160: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005208, gravity_loss=0.000001
 iteration 1170: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005182, gravity_loss=0.000001
```

```
 iteration 1180: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005157, gravity_loss=0.000001
 iteration 1190: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005131, gravity_loss=0.000001
```

```
 iteration 1200: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005106, gravity_loss=0.000001
 iteration 1210: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005082, gravity_loss=0.000001
```

```
 iteration 1220: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005057, gravity_loss=0.000001
 iteration 1230: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005033, gravity_loss=0.000001
```

```
 iteration 1240: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.005008, gravity_loss=0.000001
 iteration 1250: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.004984, gravity_loss=0.000001
```

```
 iteration 1260: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.004960, gravity_loss=0.000001
 iteration 1270: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.004936, gravity_loss=0.000000
```

```
 iteration 1280: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.004913, gravity_loss=0.000000
 iteration 1290: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.004889, gravity_loss=0.000000
```

```
 iteration 1300: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.004866, gravity_loss=0.000000
 iteration 1310: train_error_observed=0.000122, test_error_observed=0.000053,
observed_loss=0.000122, regularization_loss=0.004843, gravity_loss=0.000000
```

```
 iteration 1320: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004820, gravity_loss=0.000000
 iteration 1330: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004797, gravity_loss=0.000000
```

```
 iteration 1340: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004774, gravity_loss=0.000000
 iteration 1350: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004752, gravity_loss=0.000000
```

```
 iteration 1360: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004730, gravity_loss=0.000000
 iteration 1370: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004707, gravity_loss=0.000000
```

```
 iteration 1380: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004685, gravity_loss=0.000000
 iteration 1390: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004663, gravity_loss=0.000000
```

```
 iteration 1400: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004641, gravity_loss=0.000000
 iteration 1410: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004620, gravity_loss=0.000000
```

```
 iteration 1420: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004598, gravity_loss=0.000000
 iteration 1430: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004577, gravity_loss=0.000000
```

```
 iteration 1440: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004555, gravity_loss=0.000000
 iteration 1450: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004534, gravity_loss=0.000000
```

```
 iteration 1460: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004513, gravity_loss=0.000000
 iteration 1470: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004492, gravity_loss=0.000000
```

```
 iteration 1480: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004471, gravity_loss=0.000000
 iteration 1490: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004450, gravity_loss=0.000000
```

```
 iteration 1500: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004430, gravity_loss=0.000000
 iteration 1510: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004409, gravity_loss=0.000000
```

```
 iteration 1520: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004389, gravity_loss=0.000000
 iteration 1530: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004369, gravity_loss=0.000000
```

```
 iteration 1540: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004348, gravity_loss=0.000000
 iteration 1550: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004328, gravity_loss=0.000000
```

```
 iteration 1560: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004308, gravity_loss=0.000000
 iteration 1570: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004288, gravity_loss=0.000000
```

```
 iteration 1580: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004269, gravity_loss=0.000000
 iteration 1590: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004249, gravity_loss=0.000000
```

```
 iteration 1600: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004230, gravity_loss=0.000000
 iteration 1610: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004210, gravity_loss=0.000000
```

```
 iteration 1620: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004191, gravity_loss=0.000000
 iteration 1630: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004171, gravity_loss=0.000000
```

```
 iteration 1640: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004152, gravity_loss=0.000000
 iteration 1650: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004133, gravity_loss=0.000000
```

```
 iteration 1660: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004114, gravity_loss=0.000000
 iteration 1670: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004096, gravity_loss=0.000000
```

```
 iteration 1680: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004077, gravity_loss=0.000000
 iteration 1690: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004058, gravity_loss=0.000000
```

```
 iteration 1700: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004040, gravity_loss=0.000000
 iteration 1710: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004021, gravity_loss=0.000000
```

```
 iteration 1720: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.004003, gravity_loss=0.000000
 iteration 1730: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003984, gravity_loss=0.000000
```

```
 iteration 1740: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003966, gravity_loss=0.000000
 iteration 1750: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003948, gravity_loss=0.000000
```

```
 iteration 1760: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003930, gravity_loss=0.000000
 iteration 1770: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003912, gravity_loss=0.000000
```

```
 iteration 1780: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003894, gravity_loss=0.000000
 iteration 1790: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003876, gravity_loss=0.000000
```

```
 iteration 1800: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003859, gravity_loss=0.000000
 iteration 1810: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003841, gravity_loss=0.000000
```

```
 iteration 1820: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003824, gravity_loss=0.000000
 iteration 1830: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003806, gravity_loss=0.000000
```

```
 iteration 1840: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003789, gravity_loss=0.000000
 iteration 1850: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003772, gravity_loss=0.000000
```

```
 iteration 1860: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003755, gravity_loss=0.000000
 iteration 1870: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003737, gravity_loss=0.000000
```

```
 iteration 1880: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003720, gravity_loss=0.000000
 iteration 1890: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003704, gravity_loss=0.000000
```

```
 iteration 1900: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003687, gravity_loss=0.000000
 iteration 1910: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003670, gravity_loss=0.000000
```

```
 iteration 1920: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003653, gravity_loss=0.000000
 iteration 1930: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003637, gravity_loss=0.000000
```

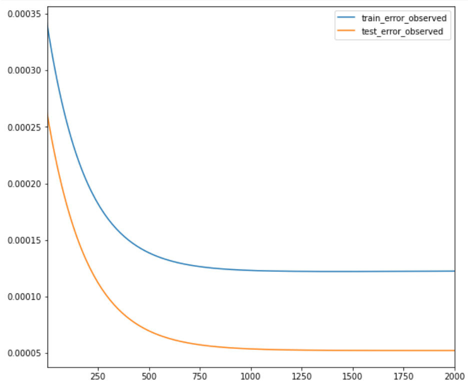
```
 iteration 1940: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003620, gravity_loss=0.000000
 iteration 1950: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003604, gravity_loss=0.000000
```

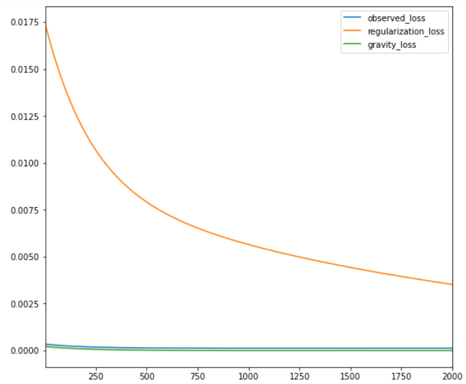
```
 iteration 1960: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003587, gravity_loss=0.000000
 iteration 1970: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003571, gravity_loss=0.000000
```

```
 iteration 1980: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003555, gravity_loss=0.000000
 iteration 1990: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003539, gravity_loss=0.000000
```

```
 iteration 2000: train_error_observed=0.000122, test_error_observed=0.000052,
observed_loss=0.000122, regularization_loss=0.003523, gravity_loss=0.000000
```

```
[{'train_error_observed': 0.00012242547, 'test_error_observed': 5.2216194e-05},
 {'observed_loss': 0.00012242547,
  'regularization_loss': 0.0035225393,
  'gravity_loss': 2.7017721e-08}]
```





Testing

```
artist_neighbors(model_lowinit, "Johnny Cash", DOT)
artist_neighbors(model_lowinit, "Johnny Cash", COSINE)
```

```
Nearest neighbors of : Johnny Cash.
[Found more than one matching artist. Other candidates: Johnny Cash & Willie Nelson]
```

	dot score	names
712	0.055784	Johnny Cash
16968	0.051251	Riceboy Sleeps
13790	0.042951	Antonello Venditti
3021	0.042440	Polar Bear Club
10392	0.040910	Colette Carr
6218	0.039689	TV-2

```
Nearest neighbors of : Johnny Cash.
[Found more than one matching artist. Other candidates: Johnny Cash & Willie Nelson]
```

	cosine score	names
712	1.000000	Johnny Cash
6218	0.598475	TV-2
3021	0.596209	Polar Bear Club
4737	0.592499	Face to Face
16230	0.575613	The Recoys
10392	0.570717	Colette Carr

Results

Our recommender system is fully functional and outputs artists based on similarity metrics to whatever artist the user enters. The system appears to have some issues as there are often useful recommendations alongside other, not so useful recommendations. The inner workings of the systems need some work before this would be deemed acceptable however as a starting point it is a useful recommender system to be further fine tuned.

Novel Lyrics Display

This section details the novel lyrics aspect of this system. A user can input a favourite artist and song and will be returned some classic lyrics from that artist on screen.

```
pip install lyricsgenius
```

```
Requirement already satisfied: lyricsgenius in
/Users/dockreg/anaconda3/lib/python3.7/site-packages (3.0.1)
Requirement already satisfied: beautifulsoup4>=4.6.0 in
/Users/dockreg/anaconda3/lib/python3.7/site-packages (from lyricsgenius) (4.10.0)
Requirement already satisfied: requests>=2.20.0 in
/Users/dockreg/anaconda3/lib/python3.7/site-packages (from lyricsgenius) (2.26.0)
Requirement already satisfied: soupsieve>1.2 in
/Users/dockreg/anaconda3/lib/python3.7/site-packages (from beautifulsoup4>=4.6.0-
>lyricsgenius) (2.2.1)
```

```
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/Users/dockreg/anaconda3/lib/python3.7/site-packages (from requests>=2.20.0-
>lyricsgenius) (1.26.6)
Requirement already satisfied: charset-normalizer~=2.0.0 in
/Users/dockreg/anaconda3/lib/python3.7/site-packages (from requests>=2.20.0-
>lyricsgenius) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in
/Users/dockreg/anaconda3/lib/python3.7/site-packages (from requests>=2.20.0-
>lyricsgenius) (3.2)
Requirement already satisfied: certifi>=2017.4.17 in
/Users/dockreg/anaconda3/lib/python3.7/site-packages (from requests>=2.20.0-
>lyricsgenius) (2021.10.8)
```

```
Note: you may need to restart the kernel to use updated packages.
```

```
import os
import json
import time
```

Token has been removed below after successfully running the API call

```
token = 'xxxxxxxxxxxxxxxxxxxxxxxxxxx'
```

```
import lyricsgenius as lg
genius = lg.Genius(token)
```

```
song_title = "Walk the line"
artist_name = "Johnny Cash"
```

```
song = genius.search_song(title=song_title, artist=artist_name)
```

```
Searching for "Walk the line" by Johnny Cash...
```

```
Done.
```

```
lyrics = song.lyrics
```

```
l=lyrics.split('\n')
```

```python
for line in l:
    print(line)

    # uncomment for interactive notebook running
    #time.sleep(2)
```

```
[Verse 1]
I keep a close watch on this heart of mine
I keep my eyes wide open all the time
I keep the ends out for the tie that binds
Because you're mine, I walk the line

[Verse 2]
I find it very, very easy to be true
I find myself alone when each day is through
Yes, I'll admit that I'm a fool for you
Because you're mine, I walk the line

[Verse 3]
As sure as night is dark and day is light
I keep you on my mind both day and night
And happiness I've known proves that it's right
Because you're mine, I walk the line

[Verse 4]
You've got a way to keep me on your side
You give me cause for love that I can't hide
For you, I know I'd even try to turn the tide
Because you're mine, I walk the line

[Verse 1]
I keep a close watch on this heart of mine
I keep my eyes wide open all the time
I keep the ends out for the tie that binds
Because you're mine, I walk the line22EmbedShare URLCopyEmbedCopy
```

Bibliography

[Can11] Iván Cantador. *Proceedings of the 2nd International Workshop on Information Heterogeneity and Fusion in Recommender Systems (HetRec 2011): 27th October 2011, Chicago, IL, USA*. ACM, 2011.

[GNOT92] David Goldberg, David Nichols, Brian M Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[GUH15] Carlos A Gomez-Uribe and Neil Hunt. The netflix recommender system: algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):1–19, 2015.

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

[MLH+18] James McInerney, Benjamin Lacker, Samantha Hansen, Karl Higley, Hugues Bouchard, Alois Gruson, and Rishabh Mehrotra. Explore, exploit, and explain: personalizing explainable recommendations with bandits. In *Proceedings of the 12th ACM conference on recommender systems*, 31–39. 2018.

[Pro21] ProducerHive. A.i. & the future of music. 2021. URL: https://producerhive.com/editorial/artificial-intelligence-and-the-future-of-music/ (visited on 2021-11-23).
