Book Recommendation Project

Recommendation Systems are one of the largest application areas of Machine Learning. They enable tailoring personalized content for users, thereby generating revenue for businesses

There are 2 main types of personalized recommendation systems:

Content based filtering

Recommendations are based on user's past likes/ dislikes & item feature space. The system makes recommendations which are similar to items the user has liked in the past. Items are considered similar based on item's features such as author, publisher, genre etc

Collaborative based filtering

Recommendations are based solely on user's past likes/ dislikes & how other users have rated other items. The system does not take into consideration an item's features like author, publisher, genre etc nor a user's features like age, gender, location etc. These take either a memory based approach or a model based approach

(1)Memory based approach: Utilizes entire user-item rating information to calculate similarity scores between items or users for making recommendations. These are further either of 2 types:

- **User based**: Two users are considered similar, if they rate items in a similar manner. An item is recommended to a user, if another user i.e., similar to the user in question has liked the item

- **Item based**: Two items are considered similar, if users rate them in a similar manner. An item is recommended to a user, that is similar to the items the user has rated in the past

(2)Model based approach: Utilizes user-item rating information to build a model & the model (not the entire dataset) is thereafter used for making recommendations. This approach is preferred in instances where time & scalability are a concern

This project aims to build a recommendation system based on collaborative filtering & will tackle an example of both memory based & model based algorithm

Datasource:

This project will use the 'Book-Crossing dataset' collected by Cai-Nicolas Ziegler (http://www2.informatik.uni-freiburg.de/~cziegler/BX/)

The dataset consists of 3 different tables:

- 'BX-Users': 278,858 records
- 'BX-Books': 271,379 records
- 'BX-Book-Ratings' : 1,149,780 records

In [1]:
```
# Libraries for data preparation & visualization
import numpy as np
import pandas as pd
import plotly.offline as py
import plotly.graph_objs as go
import plotly.io as pio
pio.renderers.default = "png"

# Ignore printing warnings for general readability
import warnings
warnings.filterwarnings("ignore")

# pip install scikit-surprise
# Importing libraries for model building & evaluation
from sklearn.model_selection import train_test_split
from surprise import Reader, Dataset
from surprise.model_selection import train_test_split, cross_validate, GridSe
from surprise import KNNBasic, KNNWithMeans, KNNWithZScore, KNNBaseline, SVD
from surprise import accuracy
```

In [2]:
```
# Loading the dataset
def loaddata(filename):
    df = pd.read_csv(f'{filename}.csv',sep=';',error_bad_lines=False,warn_bad
    return df

book   = loaddata("BX-Books")
user   = loaddata("BX-Users")
rating = loaddata("BX-Book-Ratings")
```

Exploring 'Book-Rating'

In [3]:
```
rating.shape
```

Out[3]:
```
(1149780, 3)
```

- There are 1,149,780 rows & 3 columns

In [4]:
```
rating.head(3)
```

Out[4]:

	User-ID	ISBN	Book-Rating
0	276725	034545104X	0
1	276726	0155061224	5
2	276727	0446520802	0

In [5]:
```
# Check datatypes & missing values
rating.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1149780 entries, 0 to 1149779
Data columns (total 3 columns):
 #   Column       Non-Null Count    Dtype
---  ------       --------------    -----
 0   User-ID      1149780 non-null  int64
 1   ISBN         1149780 non-null  object
 2   Book-Rating  1149780 non-null  int64
dtypes: int64(2), object(1)
memory usage: 26.3+ MB
```

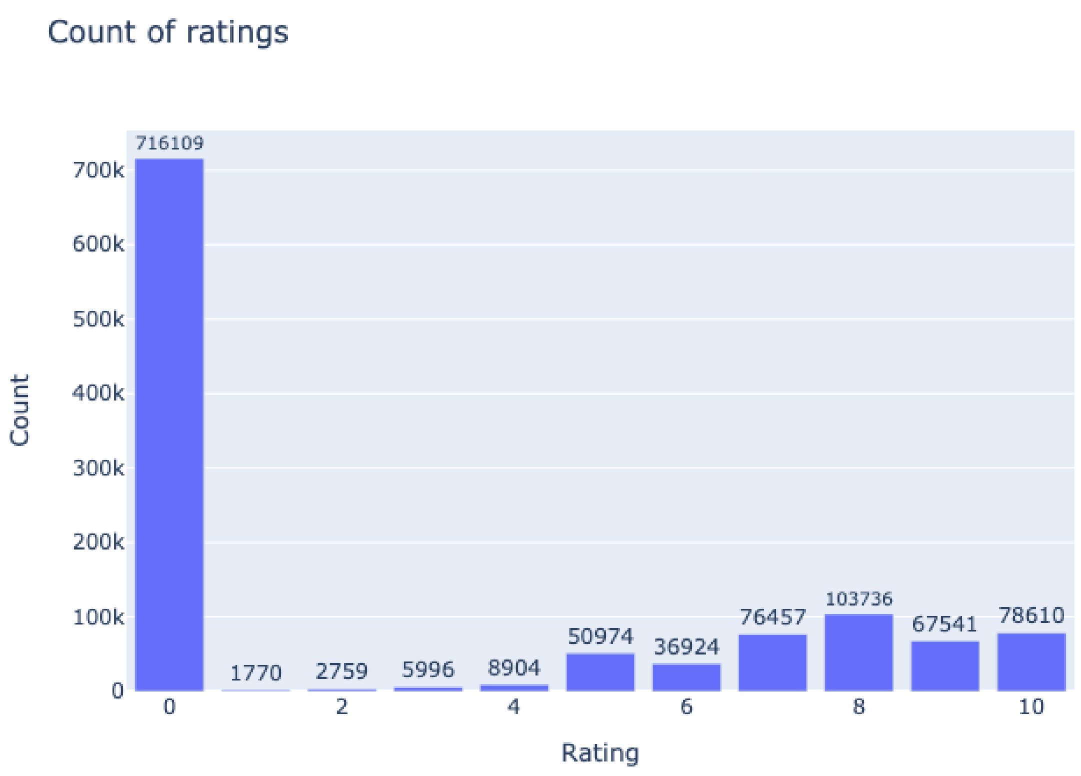
- There are no missing values

In [6]:
```
# Check for duplicate values
print(f'Duplicate entries: {rating.duplicated().sum()}')
```

```
Duplicate entries: 0
```

In [7]:
```
# Let's visualize the 'Book-Rating' column

rating_types = rating['Book-Rating'].value_counts()
x,y   = rating_types.index, rating_types.values
data  = go.Bar(x=x,y=y, text=y, textposition="outside")
Layout=go.Layout(title="Count of ratings",xaxis={'title':'Rating'},yaxis={'ti
go.Figure(data,Layout)
```



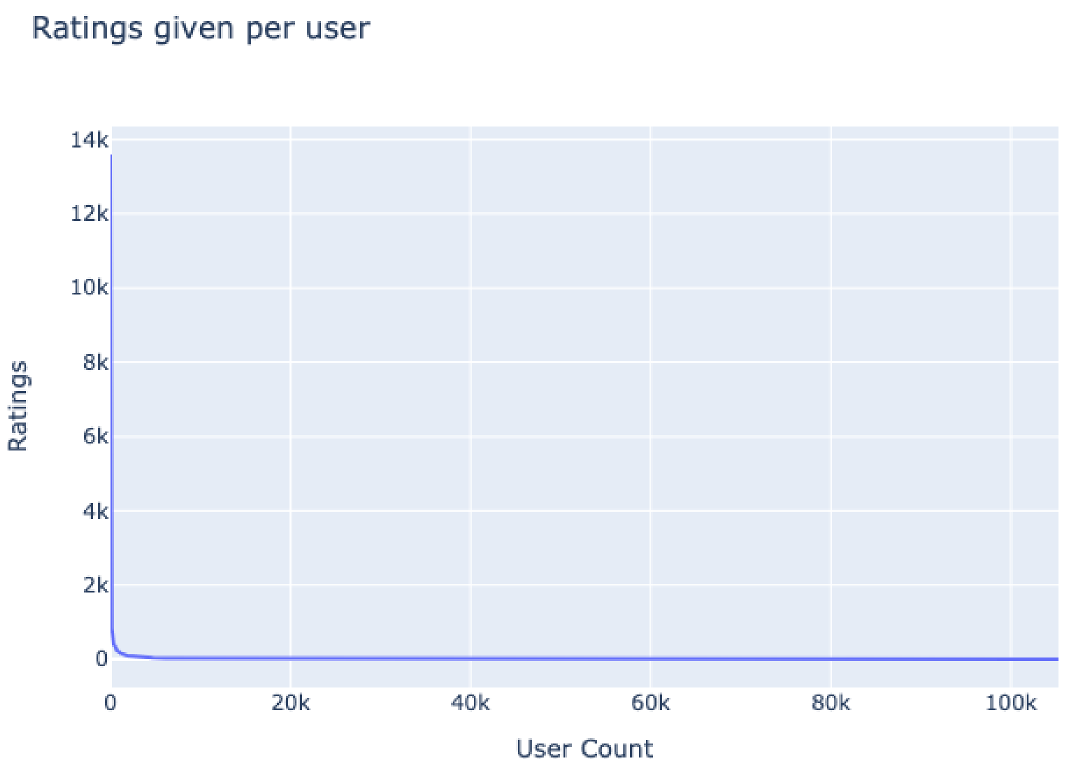
Ratings are of two types, an implicit rating & explicit rating. An implicit rating is based on tracking user interaction with an item such as a user clicking on an item and in this case is recorded as rating '0'. An explicit rating is when a user explicitly rates an item, in this case on a numeric scale of '1-10'

- Majority of ratings are implicit i.e., rating '0'
- Rating of '8' has the highest rating count among explicit ratings '1-10'

In [8]:
```
# Let's visualize ratings given by users

rating_users = rating['User-ID'].value_counts().reset_index().\
               rename({'Index':'User-ID','User-ID':'Rating'}, axis=1)
```

```
data  = go.Scatter(x = rating_users.index, y= rating_users['Rating'])
Layout= go.Layout(title="Ratings given per user",xaxis={'title':'User Count'}
go.Figure(data, Layout)
```



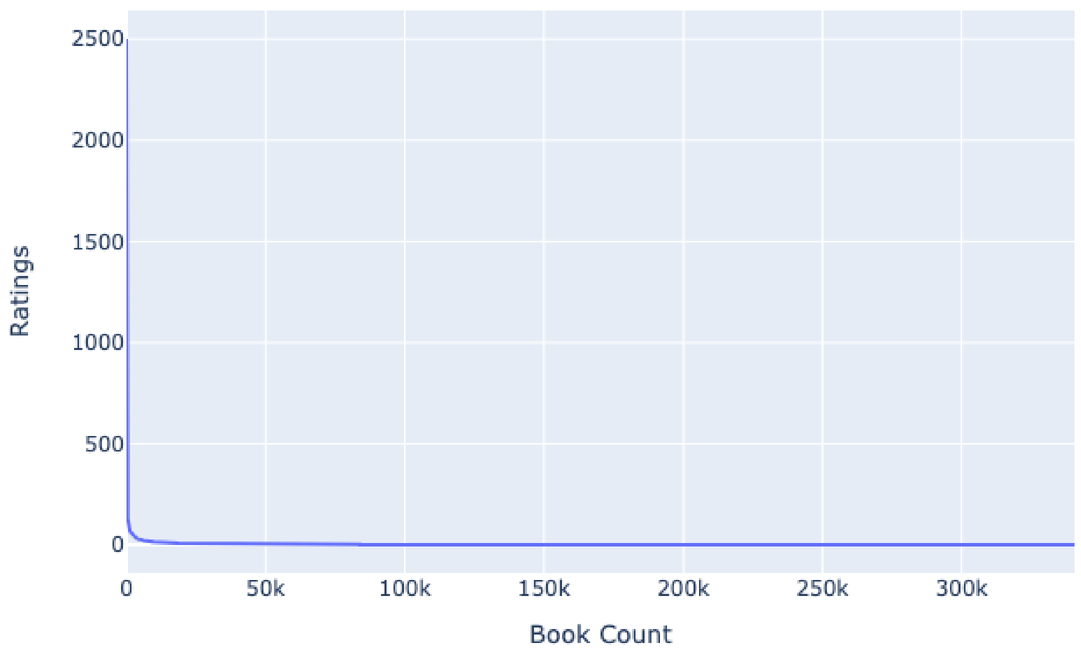
There is inherent bias in the dataset. There are few users who rate a lot & several users that provide very few ratings. One user has provided 13K+ ratings, only ~700 users (out of 100K+ users) have provided over 250 ratings

In [9]:
```
# Let's visualize ratings received by books

rating_books = rating['ISBN'].value_counts().reset_index().\
               rename({'Index':'ISBN','ISBN':'Rating'}, axis=1)

data  = go.Scatter(x = rating_books.index, y= rating_books['Rating'])
Layout= go.Layout(title="Ratings received per book",xaxis={'title':'Book Coun
go.Figure(data, Layout)
```

Ratings received per book



A similar bias is observed. There are a few books that have received many ratings and several books that have received very few ratings. One book has received over 2500 ratings, only ~2100 books (out of 300K+ books) have received more than 50 ratings

In [10]:
```
# In order to avoid rating bias & for making good recommendations, limit the 
# users that have made at least 250 ratings & books that have received at lea

rating = rating[rating['User-ID'].isin(rating_users[rating_users['Rating']>25
rating = rating[rating['ISBN'].isin(rating_books[rating_books['Rating']> 50][

rating
```

Out[10]:

	User-ID	ISBN	Book-Rating
1456	277427	002542730X	10
1468	277427	006092988X	0
1469	277427	0060930535	0
1470	277427	0060932139	0
1471	277427	0060934417	0
...	...	...	...
1147440	275970	1400031354	0
1147441	275970	1400031362	0
1147470	275970	1558744606	0
1147517	275970	1573229725	0
1147584	275970	1853260010	0

79847 rows × 3 columns

In [11]:

```python
# For the recommendation system, it is prefered to have the book titles rathe

rating = rating.merge(book, on="ISBN")[['User-ID','Book-Title','Book-Rating']
rating
```

Out[11]:

	User-ID	Book-Title	Book-Rating
0	277427	Politically Correct Bedtime Stories: Modern Ta...	10
1	3363	Politically Correct Bedtime Stories: Modern Ta...	0
2	11676	Politically Correct Bedtime Stories: Modern Ta...	6
3	12538	Politically Correct Bedtime Stories: Modern Ta...	10
4	13552	Politically Correct Bedtime Stories: Modern Ta...	0
...	...	...	...
79308	234828	Ringworld	8
79309	236283	Ringworld	0
79310	249628	Ringworld	0
79311	261829	Ringworld	0
79312	264321	Ringworld	8

79313 rows × 3 columns

- This has removed 534 records, possibly because there were some ISBN entries in the rating table that were not present in the book table. Finally, there are 79313 records

In [12]:

```python
# Check for duplicate values
print(f'Duplicate entries: {rating.duplicated().sum()}')
```

```
Duplicate entries: 531
```

Merging rating & book has introduced 531 duplicate records which will be dropped

In [13]:

```python
rating.drop_duplicates(inplace=True)
rating
```

Out[13]:

	User-ID	Book-Title	Book-Rating
0	277427	Politically Correct Bedtime Stories: Modern Ta...	10
1	3363	Politically Correct Bedtime Stories: Modern Ta...	0
2	11676	Politically Correct Bedtime Stories: Modern Ta...	6
3	12538	Politically Correct Bedtime Stories: Modern Ta...	10
4	13552	Politically Correct Bedtime Stories: Modern Ta...	0
...	...	...	...
79308	234828	Ringworld	8
79309	236283	Ringworld	0
79310	249628	Ringworld	0
79311	261829	Ringworld	0

	User-ID	Book-Title	Book-Rating
79312	264321	Ringworld	8

78782 rows × 3 columns

- Finally, there are 78782 records

In [14]:
```
list_of_distinct_users = list(rating['User-ID'].unique())
```

Machine Learning - Model Selection

In [15]:
```
# creating a surprise object

reader = Reader(rating_scale=(0, 10))
data   = Dataset.load_from_df(rating[['User-ID','Book-Title','Book-Rating']],


# Split the data into training & testing sets. Python's surprise documentatio
# https://surprise.readthedocs.io/en/stable/FAQ.html

raw_ratings = data.raw_ratings
import random
random.shuffle(raw_ratings)                    # shuffle dataset

threshold   = int(len(raw_ratings)*0.8)

train_raw_ratings = raw_ratings[:threshold] # 80% of data is trainset
test_raw_ratings  = raw_ratings[threshold:] # 20% of data is testset

data.raw_ratings = train_raw_ratings         # data is now the trainset
trainset         = data.build_full_trainset()
testset          = data.construct_testset(test_raw_ratings)
```

Python's surprise library has several built-in algorithms for building rating based recommendation systems

KNN (K Nearest Neighbours), memory based approach

This algorithm takes into consideration up-to 'K' nearest users (in user based collaborative filtering) or 'K' nearest items (in item based collaborative filtering) for making recommendations. By default, the algorithm is 'user-based', and k is 40 (kmin is 1). This means ratings of 40 nearest users are considered while recommending an an item to a user. Some variants of this algorithm include WithMeans, WithZscore & Baseline wherein the average rating of users, or the normalized ZScores of ratings or the baseline rating are also considered as the system generates recommendations

SVD (Singular Value Decomposition), model based approach

This algorithm takes a matrix factorization approach. The user-item rating matrix is factorized into smaller dimension user & item matrices consisting of latent factors (hidden characteristics). By default, number of latent factors is 100. These latent factors are able to capture the known user-item rating preference & in the process are able to predict an estimated rating for all user-item pair where user has not yet rated an item

In [16]:
```
# Trying KNN (K-Nearest Neighbors) & SVD (Singluar Value decomposition) algor

models=[KNNBasic(),KNNWithMeans(),KNNWithZScore(),KNNBaseline(),SVD()]
results = {}

for model in models:
    # perform 5 fold cross validation
    # evaluation metrics: mean absolute error & root mean square error
    CV_scores = cross_validate(model, data, measures=["MAE","RMSE"], cv=5, n_

    # storing the average score across the 5 fold cross validation for each m
    result = pd.DataFrame.from_dict(CV_scores).mean(axis=0).\
             rename({'test_mae':'MAE', 'test_rmse': 'RMSE'})
    results[str(model).split("algorithms.")[1].split("object ")[0]] = result
```

In [17]:
```
performance_df = pd.DataFrame.from_dict(results)
print("Model Performance: \n")
performance_df.T.sort_values(by='RMSE')
```

```
Model Performance:
```

Out[17]:

	MAE	RMSE	fit_time	test_time
knns.KNNWithMeans	2.351416	3.289229	0.131446	1.228990
knns.KNNBaseline	2.356247	3.299194	0.148220	1.486081
matrix_factorization.SVD	2.392815	3.313362	5.206008	0.194471
knns.KNNWithZScore	2.336986	3.327676	0.247611	2.502699
knns.KNNBasic	2.445218	3.501194	0.201198	1.567211

- KNNWithMeans has the least RMSE (root mean square error) among KNN algorithms
- The model fit_time is the maximum for SVD but the model test_time is the least

Machine Learning – Hyperparameter tuning with GridSearchCV

KNNWithMeans

- user_based: By default, this model parameter is 'True'. The other option 'False', corresponds to an item based approach
- min_support: This refers to number of items to consider (in user based approach) or number of users to consider (in item based approach) to calculate similarity before setting it to 0
- name: This refers to the distance measure that KNNWithMeans utilizes for calculating similarity. By default, the value is set as 'MSD' i.e., Mean Squared Distance. One other popular distance measure for rating based data is 'cosine' or angular distance. The cosine distance enables calculation of similarity among items & users accounting for inherent rating bias amongst users. E.g., users who like item2 twice as much as item1 may rate items as '8' & '4' if they are generous with their ratings but rate it only '4' & '2' if they are more stringent raters. MSD measures similarity based on the absolute ratings and will not be able to capture this inherent rating bias described above, however, cosine distance measure will be able to capture the same

In [18]:
```
# Hyperparameter tuning - KNNWithMeans

param_grid = { 'sim_options' : {'name': ['msd','cosine'], \
                                 'min_support': [3,5], \
                                 'user_based': [False, True]}
             }

gridsearchKNNWithMeans = GridSearchCV(KNNWithMeans, param_grid, measures=['ma
                                      cv=5, n_jobs=-1)

gridsearchKNNWithMeans.fit(data)

print(f'MAE Best Parameters:  {gridsearchKNNWithMeans.best_params["mae"]}')
print(f'MAE Best Score:       {gridsearchKNNWithMeans.best_score["mae"]}\n')

print(f'RMSE Best Parameters: {gridsearchKNNWithMeans.best_params["rmse"]}')
print(f'RMSE Best Score:      {gridsearchKNNWithMeans.best_score["rmse"]}\n')
```

```
MAE Best Parameters:  {'sim_options': {'name': 'cosine', 'min_support': 3, 'us
er_based': False}}
MAE Best Score:       2.356102894996334

RMSE Best Parameters: {'sim_options': {'name': 'msd', 'min_support': 5, 'user_
based': True}}
RMSE Best Score:      3.295189586290075
```

- Post Hyperparameter Tuning with GridSearchCV, the best parameters are found to be different for MAE & RMSE metrics
- 'Cosine' distance measure, min_support of 3 & user_based : False i.e., item based approach have been chosen for building recommendations

The logic/code below can be modified to make recommendations using 'MSD' distance & user based method if needed

In [19]:
```
# Model fit & prediction - KNNWithMeans

sim_options = {'name':'cosine','min_support':3,'user_based':False}
final_model = KNNWithMeans(sim_options=sim_options)

# Fitting the model on trainset & predicting on testset, printing test accura
pred = final_model.fit(trainset).test(testset)

print(f'\nUnbiased Testing Performance:')
print(f'MAE: {accuracy.mae(pred)}, RMSE: {accuracy.rmse(pred)}')
```

```
Computing the cosine similarity matrix...
Done computing similarity matrix.

Unbiased Testing Performance:
MAE:  2.3052
RMSE: 3.2597
MAE: 2.3052403701952713, RMSE: 3.2596506157022644
```

- The MAE & RMSE metrics for testset are comparable with what was obtained using cross validation & hyperparameter tuning stages with trainset. Chosen model hence, generalizes well

SVD

- n_factors: This refers to number of latent factors (hidden characteristics) for matrix factorization/ dimensionality reduction. By default, the value is 100
- n_epochs: This refers to number of iterations of stochiastic gradient descent procedure, utilized by SVD for learning the parameters and minimizing error
- lr_all & reg_all: i.e., learning rate and regularization rate. Learning rate is the step size of the said (above) SGD algorithm whereas regularization rate prevents overlearning, so the model may generalize well on data it has not yet seen. By default these values are set as 0.005 & 0.02

In [20]:
```
# Hyperparameter tuning - SVD

param_grid = {"n_factors": range(10,100,20),
              "n_epochs" : [5, 10, 20],
              "lr_all"   : [0.002, 0.005],
              "reg_all"  : [0.2, 0.5]}

gridsearchSVD = GridSearchCV(SVD, param_grid, measures=['mae', 'rmse'], cv=5,

gridsearchSVD.fit(data)

print(f'MAE Best Parameters:  {gridsearchSVD.best_params["mae"]}')
print(f'MAE Best Score:       {gridsearchSVD.best_score["mae"]}\n')

print(f'RMSE Best Parameters: {gridsearchSVD.best_params["rmse"]}')
print(f'RMSE Best Score:      {gridsearchSVD.best_score["rmse"]}\n')
```

```
MAE Best Parameters:  {'n_factors': 10, 'n_epochs': 20, 'lr_all': 0.005, 'reg_
all': 0.2}
MAE Best Score:       2.3774480013382537

RMSE Best Parameters: {'n_factors': 50, 'n_epochs': 10, 'lr_all': 0.005, 'reg_
all': 0.2}
RMSE Best Score:      3.1554222523739055
```

- Post Hyperparameter Tuning with GridSearchCV, the best parameters are found to be different for MAE & RMSE metrics
- 'n_factors':50, 'n_epochs':10, 'lr_all':0.005 & 'reg_all': 0.2 have been chosen for building recommendations

In [21]:
```
# Model fit & prediction - SVD

final_model = SVD(n_factors=50, n_epochs=10, lr_all=0.005, reg_all= 0.2)

# Fitting the model on trainset & predicting on testset, printing test accura
pred = final_model.fit(trainset).test(testset)

print(f'\nUnbiased Testing Performance')
print(f'MAE: {accuracy.mae(pred)}, RMSE: {accuracy.rmse(pred)}')
```

```
Unbiased Testing Performance
MAE:  2.3800
RMSE: 3.1341
MAE: 2.380013685426838, RMSE: 3.1341131981036168
```

- The MAE & RMSE metrics for testset are comparable with what was obtained using cross validation & hyperparameter tuning stages with trainset. Chosen model hence again, generalizes well

Building recommendations

In [22]:
```
# Entire dataset will be used for building recommendations

reader = Reader(rating_scale=(0, 10))
data = Dataset.load_from_df(rating[['User-ID','Book-Title','Book-Rating']], r
trainset = data.build_full_trainset()

# A list of useful trainset methods are explained here:
# https://surprise.readthedocs.io/en/stable/trainset.html
```

Two different functions are written to generate recommendations with the final chosen models KNNWithMeans & SVD

In [23]:
```
# KNNWithMeans

def generate_recommendationsKNN(userID=13552, like_recommend=5, get_recommend

    ''' This function generates "get_recommend" number of book recommendation
        KNNWithMeans & item based filtering. The function needs as input thre
        different parameters:
        (1) userID i.e., userID for which recommendations need to be generate
        (2) like_recommend i.e., number of top recommendations for the userID
        considered for making recommendations
        (3) get_recommend i.e., number of recommendations to generate for the
        Default values are: userID=13552, like_recommend=5, get_recommend=10
    '''

    # Compute item based similarity matrix
    sim_options       = {'name':'cosine','min_support':3,'user_based':False}
    similarity_matrix = KNNWithMeans(sim_options=sim_options).fit(trainset).\
                        compute_similarities()

    userID      = trainset.to_inner_uid(userID)    # converts the raw userID
    userRatings = trainset.ur[userID]              # method .ur takes user in
                                                   # returns back user rating


    # userRatings is a list of tuples [(,),(,),(,)..]. Each tuple contains it
    # given by the user for that item. Next, the tuples will be sorted within
    # in decreasing order of rating. Then top 'like_recommend' items & rating

    temp_df = pd.DataFrame(userRatings).sort_values(by=1, ascending=False).\
              head(like_recommend)
    userRatings = temp_df.to_records(index=False)

    # for each (item,rating) in top like_recommend user items, multiply the u
    # the item with the similarity score (later is obtained from item similar
    # all items. This helps calculate the weighted rating for all items. The 
    # are added & divided by sum of weights to estimate rating the user would

    recommendations   = {}

    for user_top_item, user_top_item_rating  in userRatings:

        all_item_indices          =    list(pd.DataFrame(similarity_matrix)[us
        all_item_weighted_rating  =    list(pd.DataFrame(similarity_matrix)[us
                                           user_top_item_rating)
        all_item_weights          =   list(pd.DataFrame(similarity_matrix)[us


        # All items & final estimated ratings are added to a dictionary calle

        for index in range(len(all_item_indices)):
            if index in recommendations:
                # sum of weighted ratings
                recommendations[index] += all_item_weighted_rating[index]
            else:
                recommendations[index]  = all_item_weighted_rating[index]


    for index in range(len(all_item_indices)):
            if all_item_weights[index]  !=0:
                # final ratings (sum of weighted ratings/sum of weights)
                recommendations[index]   =recommendations[index]/\
                                          (all_item_weights[index]*like_recom


    # convert dictionary recommendations to a be a list of tuples [(,),(,),(,
    # with each tuple being an item & estimated rating user would give that i
    # sort the tuples within the list to be in decreasing order of estimated 

    temp_df = pd.Series(recommendations).reset_index().sort_values(by=0, asce
    recommendations = list(temp_df.to_records(index=False))

    # return get_recommend number of recommedations (only return items the us
    # has not previously rated)

    final_recommendations = []
    count = 0

    for item, score in recommendations:
        flag = True
        for userItem, userRating in trainset.ur[userID]:
            if item == userItem:
                flag = False       # If item in recommendations has not been 
                break              # add to final_recommendations
        if flag == True:
            final_recommendations.append(trainset.to_raw_iid(item))
            count +=1              # trainset has the items stored as inner i
                                   # convert to raw id & append

        if count > get_recommend:  # Only get 'get_recommend' number of recom
            break

    return(final_recommendations)
```

In [24]:

```
recommendationsKNN = generate_recommendationsKNN(userID=13552, like_recommend
recommendationsKNN
```

```
Computing the cosine similarity matrix...
Done computing similarity matrix.
Computing the cosine similarity matrix...
Done computing similarity matrix.
```

Out[24]:

```
['The Lake House',
 'Harry Potter and the Chamber of Secrets (Book 2)',
 'Why Girls Are Weird : A Novel',
 'SKINNY LEGS AND ALL',
 '2nd Chance',
 'Round Ireland With a Fridge',
 "Harry Potter and the Sorcerer's Stone (Book 1)",
 'Summer Pleasures',
 'And Then There Were None : A Novel',
 'This Present Darkness',
 'Vittorio the Vampire: New Tales of the Vampires']
```

The above output shows the top 10 recommendations for userID 13552 based on the top 5 rated books of the user

In [25]:

```
# SVD

def generate_recommendationsSVD(userID=13552, get_recommend =10):

    ''' This function generates "get_recommend" number of book recommendation
        using Singular value decomposition. The function needs as input two
        different parameters:
        (1) userID i.e., userID for which recommendations need to be generate
        (2) get_recommend i.e., number of recommendations to generate for the
        Default values are: userID=13552, get_recommend=10
    '''

    model = SVD(n_factors=50, n_epochs=10, lr_all=0.005, reg_all= 0.2)
    model.fit(trainset)

    # predict rating for all pairs of users & items that are not in the train

    testset = trainset.build_anti_testset()
    predictions = model.test(testset)
    predictions_df = pd.DataFrame(predictions)

    # get the top get_recommend predictions for userID

    predictions_userID = predictions_df[predictions_df['uid'] == userID].\
                         sort_values(by="est", ascending = False).head(get_re

    recommendations = []
    recommendations.append(list(predictions_userID['iid']))
    recommendations = recommendations[0]

    return(recommendations)
```

In [26]:

```
recommendationsSVD = generate_recommendationsSVD(userID=13552, get_recommend 
recommendationsSVD
```

Out[26]:

```
["Sabine's Notebook: In Which the Extraordinary Correspondence of Griffin &am
p; Sabine Continues",
 'Griffin &amp; Sabine: An Extraordinary Correspondence',
 "Harry Potter and the Sorcerer's Stone (Book 1)",
 'The Darwin Awards: Evolution in Action',
 'The Lion, the Witch, and the Wardrobe (The Chronicles of Narnia, Book 2)',
 'Harry Potter and the Prisoner of Azkaban (Book 3)',
 'To Kill a Mockingbird',
 '84 Charing Cross Road',
 "Big Cherry Holler: A Big Stone Gap Novel (Ballantine Reader's Circle)",
 'The Book of Questions']
```

The above output shows the top 10 recommendations for the same user 13552

Comparing recommendations

- While the list of recommendations generated using KNNWithMeans & SVD are different

(expected as they are different algorithms), there are some similarities in the generated lists too
- Both algorithms recommended instances of Harry Potter novels for user 13552. Additionally, the recommended books seem to be a similar genre lending confidence in interpretability of recommendations

Conlusions & next steps

- We have successfully implemented a memory based as well as method based collaborative filtering approach to make recommendations in this project
- In instances with a new user or new item where little is known of the rating preference, collaborative filtering may not be the method of choice for generating recommendations. Content based filtering methods may be more appropriate. Often, a hybrid approach is taken for building real time recommendations using multiple different approaches in industry! The project can be extended to build hybrid recommendation systems in the future