**­Title: Book suggestion algorithm using Surprise**

**Introduction:**

The open-source Python module Surprise makes it simple for programmers to create recommender systems with explicit rating information. This library is written in Python which makes it easy to use in Python projects.

**Installation**

Surprise can easily be installed using the command below:

pip install scikit-surprise

or

conda install -c conda-forge scikit-surprise if you are on and anaconda environment

**Import libs**

I simply imported some fundamental libraries for data manipulation and visualization to get started as below.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

%matplotlib inline

**Read the datasets**

I utilized two CSV files from the dataset. The first one includes evaluation information for 10,000 books that over 53,000 users have reviewed. The metadata (title, author, ISBN, etc.) for each of the 10,000 books are included in the second file as below.

rate\_data = pd.read\_csv(‘contents/Books data/rating.csv’)

meta\_data = pd.read\_csv(‘content/Books data/books.csv’)

**Surprise Dataset creation**

We need to create a Dataset object before we can use Surprise to train recommender systems. A dataset that has the following fields in the following sequence is a surprise dataset object:

1. User Ids
2. Item Ids i.e., Id for each book
3. Corresponding rating on a scale from 1 – 5
4. from surprise import Dataset,Reader
5. reader = Reader(rating\_scale=(1,5)) ## Setting the rating scale between 1 and 5
6. data\_set = Dataset.load\_from\_df(rate\_data[['user\_id','book\_id','rating']],reader)

**Cross-Validation and Training of SVD (Singular Value Decomposition) model.**

In just a few lines of code, we can train and cross-validate an SVD (singular value decomposition) model to create a recommendation system. A well-liked matrix factorization algorithm for recommender systems is SVD.

A matrix of ratings is typically factored into a product of matrices reflecting latent characteristics for the products (in this case, books) and the users in matrix-based recommender systems.

|  |  |  |
| --- | --- | --- |
| Q11 | Q12 | Q13 |
| Q21 | Q22 | Q23 |
| Q31 | Q32 | Q33 |
| Q41 | Q42 | Q43 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 |  | 3 |  | 4 |
| 4 |  | 1 |  | 2 |
| 5 |  | 4 | 3 |  |
|  | 1 |  |  | 3 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P11 | P12 | P13 | P14 | P15 |
| P21 | P22 | P23 | P24 | P25 |
| P31 | P32 | P33 | P34 | P35 |



Observe how the rating matrix, R, in the following graphic has some missing values. When predicting existing ratings using the matrix factors, the matrix factorization algorithm use a technique like gradient descent to reduce inaccuracy. In order to "fill in the gaps" in the rating matrix, an algorithm like SVD enables us to anticipate the ratings that each user would give to each item in the dataset.

As seen in the equation below, SVD actually factors the original matrix into three matrices starting with an input matrix A.

**A=U∑VT**

These new matrices can be mapped as follows to the rating matrix R and the item and user factors Q and P:

**A=R ,Q=U,PT=∑VT**

For our book recommendation system, the SVD algorithm will combine matrices that reflect the book factor and the user factor to create the rating matrix.

The code below show cross validation of svd model using three-fold cross validation

from surprise import SVD

from surprise.model\_selection import cross\_validate

svd = SVD(verbose=True,n\_epochs=10)

cross\_validate(svd,data\_set,measures=['RMSE','MAE'],cv=3,verbose=True)

Running it produces:

RMSE (testset) 0.8550 0.8576 0.8563 0.8563 0.0011

MAE (testset) 0.6749 0.6768 0.6747 0.6755 0.0009

Fit time 25.15 34.33 28.71 29.40 3.78

Test time 4.76 6.90 4.69 5.45 1.02

After transforming the dataset for cross-validation into a Surprise Trainset object using the build full trainset method, we can also train the model on the entire dataset using the fit method, but this is resource intensive.

## The model can also be trained on the entire data set but this consumes alot of computing power and takes long

train\_set = data\_set.build\_full\_trainset()

svd.fit(trainset=train\_set) ## only run this if you have enough computing power and time to spare

**Generation of Rating prediction**

Given an ID for the user (UID) and an ID for the item/book, we can use our trained SVD model to estimate the rating a user would give a book (IID). The predict method is used to accomplish this, as shown in the code below.

svd.predict(uid=10,iid=100)

The predict method returns the Prediction displayed below. This Prediction includes a field called est that represents the estimated book rating for this particular user.

Prediction(uid=10, iid=100, r\_ui=None, est=3.9205292039443203, details={'was\_impossible': False})

According to the results above, we can see that the model projected that this particular user would give the book an IID of 100, or about a four-star rating. Although the model cannot explicitly suggest books, we may use this rating prediction utility to determine which books a user is most likely to appreciate, which enables us to defend promoting them to a user.

**Generation of book recommendation**

The utility functions listed below were created for the purpose of generating book suggestions using this rating prediction tool generated above.

## Now lets get to the fun part, Lets implement our utility

import difflib

import random

def fetch\_book\_id(book\_title,meta\_data):

## We will fetchthe book id based on the closest match with regards to the metadata we are parsed

present\_titles = list(meta\_data['title'].values)

close\_titles = difflib.get\_close\_matches(book\_title,present\_titles)

book\_id = meta\_data[meta\_data['title']==close\_titles[0]]['id'].values[0]

return book\_id

def fetch\_book\_info(book\_id,meta\_data):

## We will return the basic info about a book given the book id and the metadata

book\_info = meta\_data[meta\_data['id']==book\_id][['id','isbn','authors','title','original\_title']]

return book\_info.to\_dict(orient='records')

def predict\_review(user\_id,book\_title,model,meta\_data):

## We will predict the review on a scale of 1-5 that users have assigned a specific book

book\_id = fetch\_book\_id(book\_title,meta\_data)

review\_prediction = model.predict(uid=user\_id,iid=book\_id)

return review\_prediction.est

def gen\_recommendation(user\_id,model,meta\_data,thresh=4):

## Generate a book recommendation for a user based on a rating threshhold. Only books with

## the specified recommendations will be recommended

book\_titles = list(meta\_data['title'].values)

random.shuffle(book\_titles)

for book\_title in book\_titles:

rating = predict\_review(user\_id,book\_title,model,meta\_data)

if rating >= thresh:

book\_id = fetch\_book\_id(book\_title,meta\_data)

return fetch\_book\_info(book\_id,meta\_data)

The gen\_recommendation function creates a book recommendation for a user by repeatedly going through the shuffled list of book titles and forecasting user ratings for each title until it locates a book with a rating at or above the specified threshold that qualifies it for being recommended to a user. The book recommendation is made more random by rearranging the book titles at the beginning.

Test results are below:

[{'id': 740, 'isbn': '60529962', 'authors': 'Laura Ingalls Wilder, Garth Williams', 'title': 'The Little House Collection (Little House, #1-9)', 'original\_title': 'The Little House Collection'}]

We can see from the output above that the function produces a dictionary with information about the suggested book. Multiple calls to this function will result in multiple book recommendations. We can retrain the model once a user evaluates a book to create an even better recommender system by combining the user-review data with the rating data.

**Visualizing the book factors using t-SNE**

By using the book factor matrix, denoted as Q in the earlier graphic used to describe matrix factorization methods, we may advance this research and actually display the similarities across books.

We cannot easily perceive the 100 dimensions of each book's vector in this 10,000 by 100 matrix; but, we can apply a dimensionality reduction approach to represent each book as a two-dimensional point in space. Each book is represented as a two-dimensional point in the code below using a method known as t-SNE (t-Distributed Stochastic Neighbors Embedding), and the results are saved in a data frame.

## Now let us visualize our data so as to see similarities between the books

from sklearn.manifold import TSNE

tsne = TSNE(n\_components=2,n\_iter=500,verbose=3,random\_state=1)

books\_embd = tsne.fit\_transform(svd.qi)

proj = pd.DataFrame(columns=['x-axis','y-axis'],data=books\_embd)

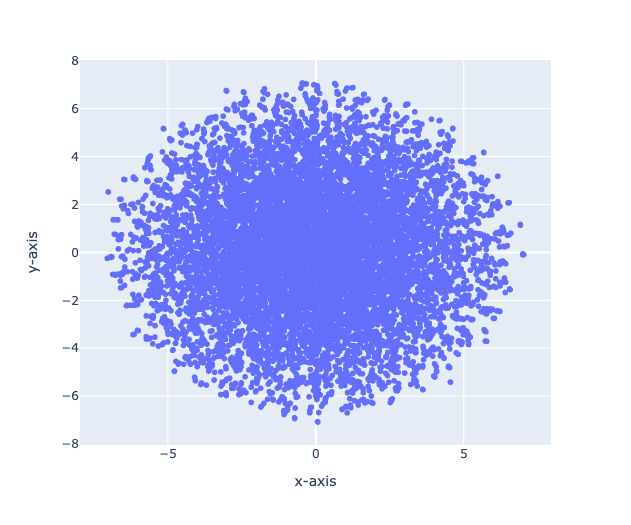
proj['title'] = meta\_data['original\_title']

I used Plotly to generate a visualization with each point representing a book in the original dataset after building this data frame with two-dimensional points for each book.

import plotly.express as px

figure = px.scatter(proj,x='x-axis',y='y-axis')

figure.show()



We can see that the points representing the 10,000 books appear to follow a two-dimensional normal distribution in the plot that the Plotly algorithm generated above. The following hypotheses regarding the books in the dataset can be used to explain this distribution:

* Some novels might be broadly well-liked by a large spectrum of readers, and as a result, they correspond to the scatterplot's central points.
* Other books might fit into highly specialized genres that appeal to particular readers, such as romance, mystery, and vampire fiction. These novels may represent locations other than the plot's center.