**This document contains a summary of the capstone project submission by Glenn Dean, mentor: Dr. Bikash Agrawal.**

1. **PROBLEM STATEMENT**
2. **SOLUTION**
3. **Dataset**
4. **TECHNICAL**
5. **DEPLOYMENT TO AWS**
6. **GITHUB**

**PROBLEM STATEMENT**:

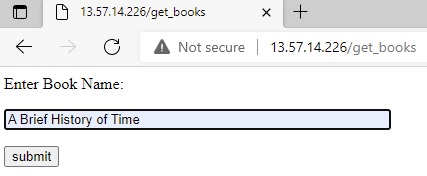
Given any item, for example a movie or a book, we would like to understand what other movies/books are similar to it. So, for example, if a person likes the movie ‘Star Wars’, we would like to find similar movies to it in the hope that the person will also like the similar movies – so you might recommend ‘The Empire Strikes Back’ or ‘Avengers’ say.

Another task that is very similar to the above, is the task of predicting how a user would rate a particular item, say a movie or a book. So if a person rated ‘Harry Potter and the Deathly Hallows’ a ‘5 Stars’, then you’d like to understand how this person would rate other books and then recommend the books that we predict the person would rate say a ‘4 Stars’ or higher. So, for example, you might recommend the book ‘Harry Potter and the Half-Blood Prince’ or ‘Fantastic Beasts and Where to Find Them’ (provided you predict the person would rate these books very high).

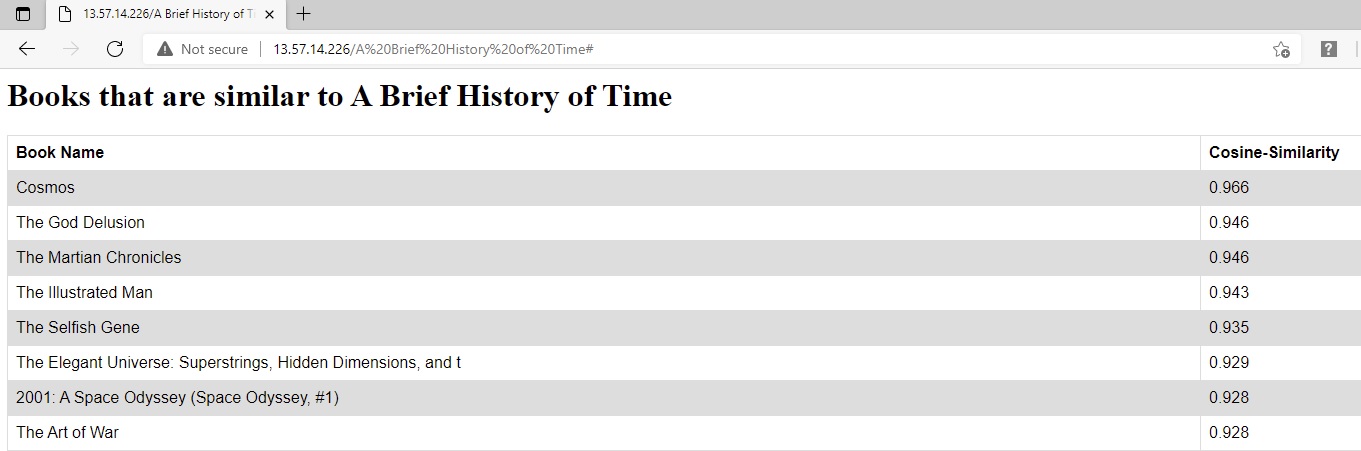
**SOLUTION:**

This capstone project creates three recommender systems to address the problems stated above.

In the **first recommender system**, we use ‘Truncated Singular Value Decomposition’ to compute ‘cosine-similarity’ to compute how similar two books are. With this recommender system, a webpage is provided where you can enter a book name, and the recommender system will provide a list of books that are similar to it. A picture of the webpage is given below (note that the webpage is located by typing in ‘get\_books’ after the main webpage (which is 13.57.14.226 is this particular case):



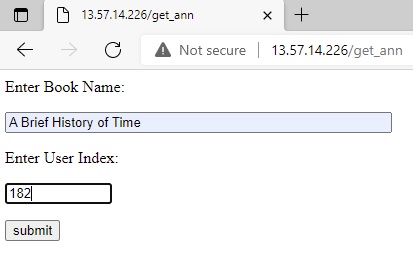
Once you click the ‘submit’ button, the results will be displayed, as seen in the pic below:



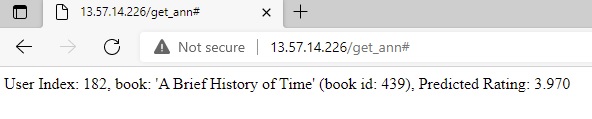
More details on this TruncatedSVD recommender system will be given in the ‘Technical’ section, where we will discuss the decomposition of matrices, and the number of ‘features’ or ‘concepts’ we reduced to. But for now, note that the higher the ‘cosine-similarity’, the more similar it is to the book ‘A Brief History of Time’.

In the **second recommender system**, we use ‘deep neural network’ (DNN) to make predictions for how a particular user will rate a book. This recommender system is very different from using TruncatedSVD, which provides a “strength of similarity” between books, but does NOT attempt to predict the ratings of books. The second recommender system DNN predicts a rating for a particular book (between 1.0 Stars to 5.0 Stars).

A picture of the webpage is given below (note that the webpage is located by typing in ‘get\_ann’ after the main webpage (which is 13.57.14.226 is this particular case):



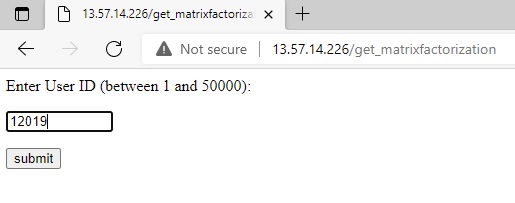
After you click the ‘submit’ button, the predicted rating for this book by this user will be displayed:

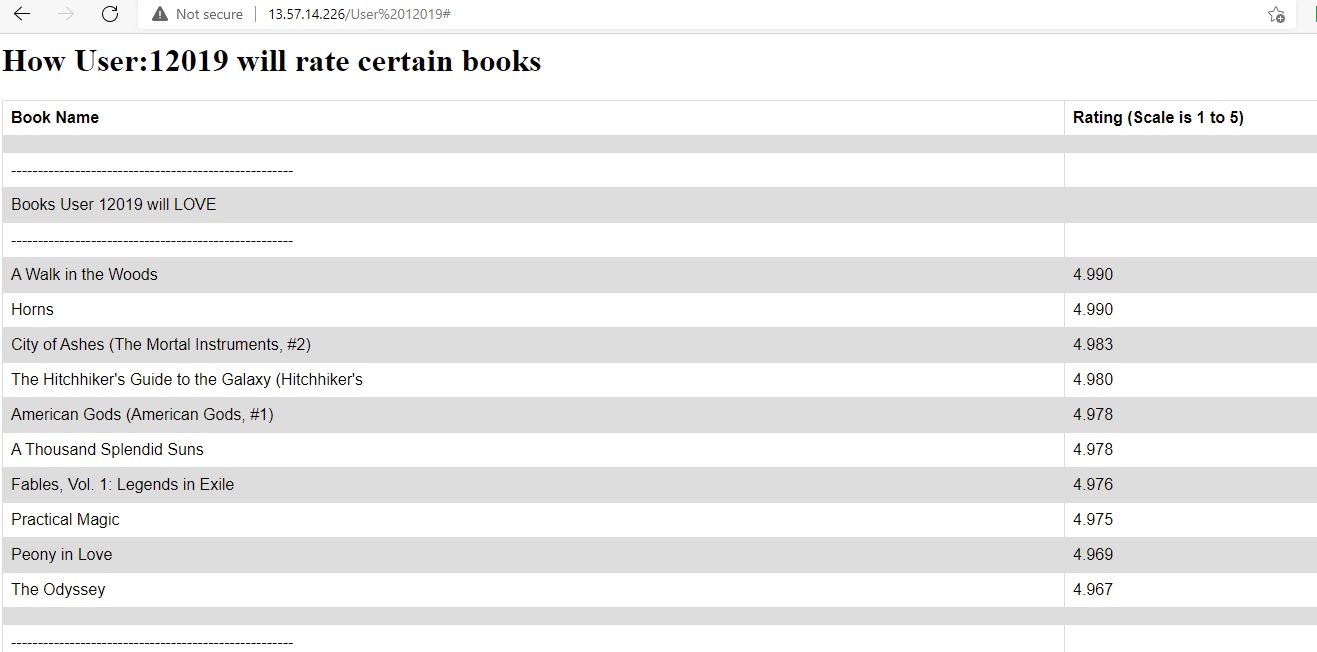


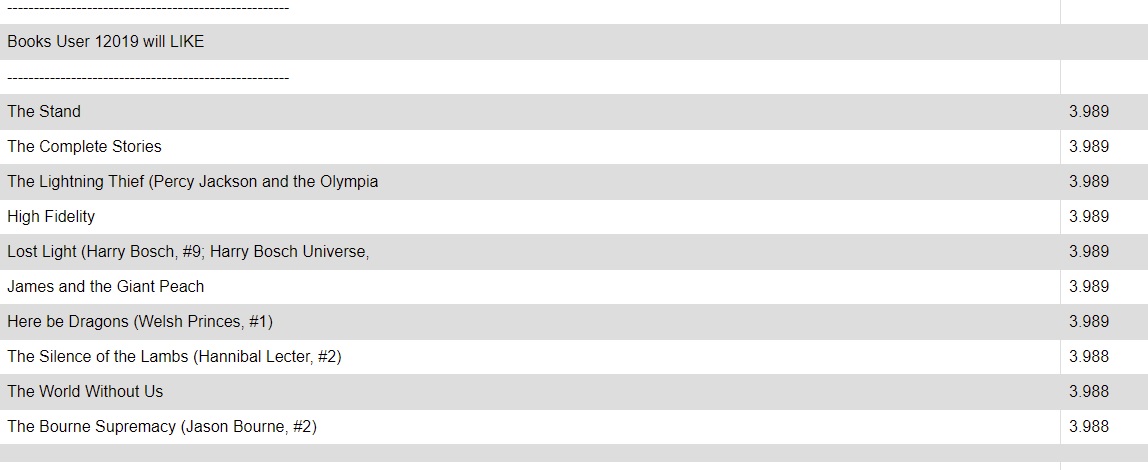
In this particular case, our recommender system indicates ‘User 182’ will rate the book ‘A Brief History of Time’ as 3.970.

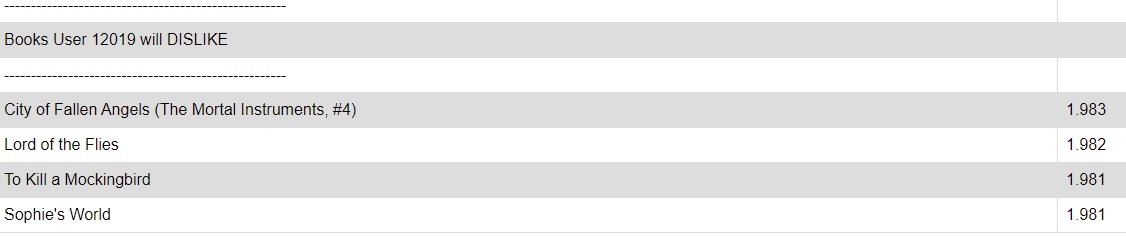
More details on the DNN recommender system will be given in the ‘Technical’ section. In particular, we treated this problem as a ‘classification’ problem and took the weighted average of the probabilities the DNN gave for each of the 5 values to get the rating (such as 3.970).

And finally our **third recommender system** uses ‘matrix multiplication’ via the Embedding() layer to make ratings for books by a user. Thru these ratings, the recommender system displays books by categories – namely it displays books this user will ‘LOVE’, display books this user will ‘LIKE’, and three other categories. Here is the webpage:









More details on matrix factorization will be given in the ‘Technical’ section.

**DATASET:**

The dataset used in this capstone project can be obtained via github:

<https://github.com/greenheritagellc/recommendation.git>

Remark on Scaling: The above dataset is fairly small – only about 10000 books and 55000 users. If you did a “pivot\_table” you would get 550,000,000 entries in the table, and even on my system with 8 processors and 16GB RAM it overloaded my system. When you start to get tables with say 1 billion entries or more, you need to migrate to a distributed environment like Spark.

**TECHNICAL:**

**First Recommender System: TruncatedSVD**

Given any real-valued matrix A of dimension m-by-n (written m x n), with rank r (this is the dimension of the column space), it can always be decomposed as a product of three matrices U,S,V such that A = U X S X V^t (V^t is the transpose of matrix V). Furthermore, U has dimensions m x m, S is upper diagonal with dimensions m x n, and V is of dimensions n x n. The matrix S will contain what are called the ‘singular’ values, hence the name ‘Singular Value Decomposition’. This singular values are simply the square root of the strictly positive eigenvalues for the matrix A^t X A (i.e. ‘A transpose’ X A, which is necessarily a square matrix).

The above has nothing to do with machine learning – it is simply a theorem of linear algebra and matrices.

BUT, we can apply this technique to recommender systems by reducing the number of movies to a much smaller number, called ‘concepts’. The idea is that let’s say we have 10,000 movies and 1,000,000 users, so the matrix of users ‘cross’ movies contains 10 billion entries! BUT the rank of this matrix may only be 50 say. Essentially what happens is we create 50 new concepts, and each movie can be viewed as a vector of length 50, each component being the ‘strength of this movie’ in regards to each of the 50 concepts. One can think of these ‘concepts’ as possibly ‘to what degree a movie is an ‘action’ movie, or ‘horror’ movie, or ‘cartoon’ movie, or ‘romance’ movie. This is how one thinks of the decomposition being done, but it is important to understand that the 50 ‘concepts’ are simply 50 new features where we represent the movie.

The most important thing to understand about TruncatedSVD is that you can specify how many concepts you want – in our particular case I chose 20. The second is that the multiplication of the matrices U X S is, after removing “blocks of zeroes”, is a matrix with dimensions mx20, which is significantly less than having 10 billion entries! This matrix can be stored easily. This matrix U X S can be thought of as “movies to concept”. Given any two movies m\_1 and m\_2, you have two vectors, of length 20, which represents m\_1 in terms of the 20 concepts, and similarly for m\_2. If you take the dot product of these two vectors divided by their lengths, this gives the “cosine of the angle between these two vectors. This angle represents how strongly they are similar to each other (for example if the angle is ‘close to 0’, hence the cosine(angle) is close to 1, these two vectors are aligned almost in the same direction, which means the two movies are very similar). This dot product between the two movie vectors is called the “cosine similarity” between movie m\_1 and m\_2. It necessarily has a value between -1.0 and 1.0.

So what our first recommender system does is, when you input the name of a book, it computes the cosine similarity between it and the other thousands of books in the database! It then ranks these cosine similarity from highest (which is 1.0) to the lowest (which is -1.0), and displays the highest cosine similarity (i.e. the books that are most similar to the book you entered, or to say it another way, the books that most closely match the 20 concepts to the book you entered).

**Second Recommender System: DNN**

The artificial neural network we create simply has two features, namely the user\_id and the movie\_id, and the output is treated as a 5-class classification (1 Star, thru, 5 Star). The results from this approach were quite poor compared to ‘matrix factorization’, but the RMSE score certainly did NOT bring out the vast differences between the two approaches (DNN had a RMSE score of .98, while the MatrixFactorization has .94, see Table below). This recommender system was included in the capstone project as an example of an ANN that does NOT perform well!

|  |  |
| --- | --- |
| Model | Mean Square Error |
| Deep Neural Network | 0.9889 |
| Matrix Factorization | 0.9433 |

**Third Recommender System: Matrix Factorization**

Matrix Factorization is a modified version of SVD – in fact the literature sometimes calls ‘matrix factorization’ as ‘compact-SVD’. The reason for this is in SVD the sigma matrix ‘S’ in the SVD decomposition into U X S X V^t, S is an upper-diagonal matrix, with the upper diagonal containing the singular values. BUT the matrix S may contain either rows or columns that are all zeroes! By way of an example, if you start with the matrix

[[3 2 2],

[2 3 -2]]

The singular values are 5 and 3 and the matrix S is

[[5 0 0],

[0 3 0]]

First notice the above matrix is NOT square, and also note the 3rd column contains all zeroes. BUT in matrix factorization, the sigma matrix is guaranteed to be square; in fact it is guaranteed to be a r x r matrix, where r is the rank of the matrix. For matrix factorization, the decomposition into U X S X V^t, each matrix now has dimensions m x r, r x r, and n x r. This decomposition has several nice properties, which are:

1. The matrix U, which is m x r, is significantly smaller than the matrix in SVD. In our example of m = 10000 and n = 1000000, with r=50, the matrix U only has 500,000 entries. The V matrix only has dimensions n x r, or 50,000,000 entries. So the original matrix A, which has 10,000,000,000 entries, can be decomposed into matrices that only needs to store 500,000 + 50 + 50,000,000, which is only 50,500,050 entries (almost 200 times less memory utilized).
2. The second nice feature is the decomposition can more naturally be thought of as “movies to concepts” (this is the U matrix), then “importance of each concept” (this is the S matrix, which contains the singular values), and then “users to concepts” (this is the V matrix).
3. The S matrix can be arranged so that the singular values are in decreasing order. This is valuable in that if you originally started with 50 concepts, and you analyze the singular values and find that say only 20 concepts are important due to the size of the singular value, you can re-run your matrix factorization neural network using 20 concepts instead of 50.

**DEPLOYMENT TO THE AWS:**

First, all models were tested and created using Jupyter Notebook – the models and data structures needed for deployment were either saved via TensorFlow.Keras or pickle. I need to say that a Jupyter Notebook project is not the same as being able to run your recommender system on a public website, where users from all over the world, can enter information and retrieve recommendations, without having to wait for models to be trained! This is why the trained models are stored for later retrieval.

The python library that we utilized to create webpages and to make recommendations from our saved-off models was Flask. I was stunned by how easy Flask made writing python code for webpages – Flask made it as if I was truly “in my Jupyter Notebook” projects!

Once testing with Flask on my local machine was complete, and making sure all the webpages worked, which were

<http://localhost::5000/get_books>

<http://localhost::5000/get_ann>

<http://localhost::5000/get_matrixfactorization>

then came the process of creating a Docker image, that could be loaded into an AWS container. After creating a ‘Dockerfile’, and a ‘requirements.txt’ file via ‘pip freeze > requirements.txt’, we were ready to build our Docker image via ‘docker build -t glennrecsys .’, and then to test whether it could run in a docker container we ran ‘docker run -p 5000:5000 glennrecsys’.

After everything checked out (again going to webpages <http://localhost::5000/get_books>), it was now time to upload this image to AWS – in particular we uploaded it to AWS Elastic Container Registry (ECR).

THEN it was time finally to deploy the image into a EC2 instance on AWS. After getting everything to work, AWS gives you a public IP address that anyone in the world can access – as of July 20th, 2021 that IP address is 13.57.14.226. So to access the three recommendation systems, you would go to

13.57.14.226/get\_books

13.57.14.226/get\_ann

13.57.14.226/get\_matrixfactorization

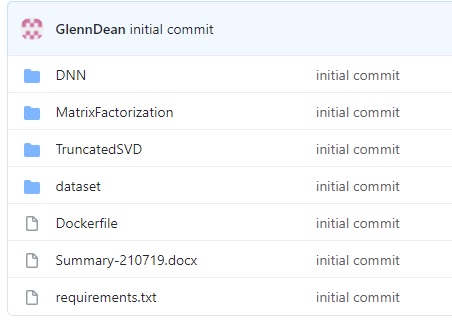
The diagram below may be helpful to understand how the overall development (and deployment) works: All code and testing initially is done in a Jupiter Notebook. THEN this code along with pickle files and migrated to a Flask application. After this, a Docker image is built from the code in your Flask application. And finally, this docker image is uploaded into AWS.

**GITHUB:**

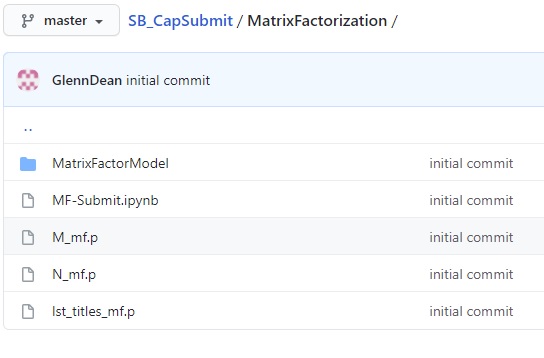
You can access all the project files at the github repository:

<https://github.com/GlennDean/SB_CapSubmit.git>

The organization of this repository is shown in the pic below:



If you open up say the MatrixFactorization folder, it looks like this:



As you can see, it has the Jupyter Notebook that was used to generate the matrix factorization recommender system. In addition to the notebook, it also has all the files that were created by this notebook that are “fed” into the Docker image that is containerized at AWS. These are the three pickle files (with extension ‘.p’) and the Keras saved model stored in folder MatrixFactorModel.