Algorithmic Digital Marketing Assignment 3

Summary	In this codelab, we have designed a prototype and deployed a recommendation system for snack shipping company.
URL	https://github.com/Pratlim29/ADMRecommendation
Category	Data Analysis, Recommender Systems and APIs
Author	Prathamesh Limaye and Saurabh Satra

About Dataset

The Dataset consists of data for the different kinds of users and ratings given by them for several snack items. The data has following columns:

- **userID**: Id's for different users
- **itemID:** id's for several snack items
- **Rating:** ratings given by different users for different snack items
- **SnackNames:** Several snack item names, for instance: Cookies, Pancakes etc.
- **SnackType:** Snack types according to meal timings, for instance: Breakfast, Lunch etc.

Restricted Boltzmann Machine (RBM)

A Restricted Boltzmann Machine (RBM) is a generative neural network model typically used to perform unsupervised learning. The main task of an RBM is to learn the joint probability distribution P(v, h), where v are the visible units and h the hidden ones. The hidden units represent latent variables while the visible units are clamped on the input data. Once the joint distribution is learnt, new examples are generated by sampling from it.
Here, we have analyzed an example of how to utilize the RBM to perform user/item recommendations.

We have loaded the dataset of snack ratings for several users with different kinds of snacks. As a second step we generate the user/item affinity matrix and then split the data into train and test sets. The splitter returns: Xtr: a matrix containing the train set ratings and Xtst: a matrix containing the test elements.

In [137]:
```
data = pd.read_csv('snackratings.csv', usecols=["userID", "itemID", "rating"])
data.head()
```

Out[137]:

	userID	itemID	rating
0	1	3342	5
1	1	7522	1
2	1	25363	2
3	1	38997	5
4	1	43685	1

In [138]:
```
data.shape
```

Out[138]:
```
(5000, 3)
```

In [139]:
```
# Convert to 32-bit in order to reduce memory consumption
data.loc[:, 'rating'] = data['rating'].astype(np.int32)
```

In [140]:
```
#to use standard names across the analysis
header = {
        "col_user": "userID",
        "col_item": "itemID",
        "col_rating": "rating",
    }
#instantiate the sparse matrix generation
am = AffinityMatrix(DF = data, **header)

#obtain the sparse matrix
X = am.gen_affinity_matrix()
```

The algorithm operates in three different steps:

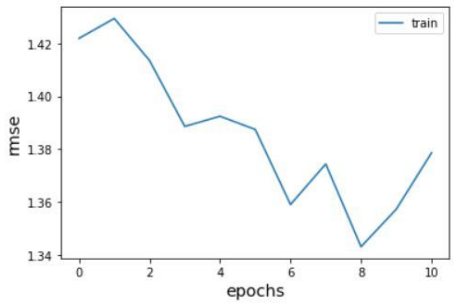
- Model initialization: The main parameters to specify are the number of hidden units, the number of training epochs and the minibatch size.
- Model fit: Through this, we train the model on the data. The method takes two arguments: the training and test set matrices.
- Model prediction: This is where we generate ratings for the unseen items. Once the model has been trained and we are satisfied with its overall accuracy, we sample new ratings from the learned distribution. In particular, we extract the top_k (e.g. 10) most relevant recommendations according to some predefined score.

During training, we have evaluated the root mean squared error to have an idea of how the learning is proceeding. We would generally like to see this quantity decreasing as a function of the learning epochs. To visualise this choose with_metrics = True in the RBM() model function.

In [143]:
```
#First we initialize the model class
model = RBM(hidden_units= 600, training_epoch = 10, minibatch_size= 60, keep_prob=0.9,with_metrics =True)
```

In [144]:
```
#Model Fit
train_time= model.fit(Xtr, Xtst)
```

```
Device mapping:
```



In [145]:
```
#number of top score elements to be recommended
K = 10

#Model prediction on the test set Xtst.
top_k, test_time =  model.recommend_k_items(Xtst)
```

In [146]:
```
top_k_df = am.map_back_sparse(top_k, kind = 'prediction')
test_df = am.map_back_sparse(Xtst, kind = 'ratings')
```

Here we evaluate the performance of the algorithm using the metrics provided in the PythonRankingEvaluation class.

```
def ranking_metrics(
    data_size,
    data_true,
    data_pred,
    time_train,
    time_test,
    K
):

    eval_map = map_at_k(data_true, data_pred, col_user="userID", col_item="itemID",
                    col_rating="rating", col_prediction="prediction",
                    relevancy_method="top_k", k= K)

    eval_ndcg = ndcg_at_k(data_true, data_pred, col_user="userID", col_item="itemID",
                      col_rating="rating", col_prediction="prediction",
                      relevancy_method="top_k", k= K)

    eval_precision = precision_at_k(data_true, data_pred, col_user="userID", col_item="itemID",
                                col_rating="rating", col_prediction="prediction",
                                relevancy_method="top_k", k= K)

    eval_recall = recall_at_k(data_true, data_pred, col_user="userID", col_item="itemID",
                          col_rating="rating", col_prediction="prediction",
                          relevancy_method="top_k", k= K)
```

```
eval= ranking_metrics(
    data_size = "snacks",
    data_true =test_df,
    data_pred =top_k_df,
    time_train=train_time,
    time_test =test_time,
    K =10)

eval
```

	Dataset	K	MAP	nDCG@k	Precision@k	Recall@k	Train time (s)	Test time (s)
0	snacks	10	0.457834	0.534757	0.116	0.58	74.640899	25.768311

Advantages of RBM:

The model generates ratings for a user/movie pair using a collaborative filtering based approach. While matrix factorization methods learn how to reproduce an instance of the user/item affinity matrix, the RBM learns the underlying probability distribution. This has several advantages:

- Generalizability : the model generalizes well to new examples.
- Stability in time: if the recommendation task is time-stationary, the model does not need to be trained often to accommodate new ratings/users.
- The tensorflow implementation presented here allows fast training on GPU

Riemannian Low-rank Matrix Completion Algorithm (RLRMC)

Riemannian Low-rank Matrix Completion (RLRMC) is a matrix factorization based (vanilla) matrix completion algorithm that solves the optimization problem using Riemannian conjugate gradients algorithm. The ratings matrix of snack items and users is modeled as a low-rank matrix. Let the number of snack items be d and the number of users be T. The RLRMC algorithm assumes that the ratings matrix M (of size d * T) is partially known. The entry at M(i, j) represents the rating given by the j-th user to the i-th snack item. RLRMC learns matrix M as M=L * R(Transpose), where L is a d * r matrix and R is a T * r matrix. Here, r is the rank hyper-parameter which needs to be provided to the RLRMC algorithm. Typically, it is assumed that r << d, T.

We have loaded the dataset of snack ratings for several users with different kinds of snacks. The second step is to define all the parameters such as rank of the matrix, regularization parameter,

iteration count etc. Then we will split the data into training and testing in the ratio of 80-20 respectively..

In [6]:
```
# Model parameters

# rank of the model, a positive integer (usually small), required parameter
rank_parameter = 10
# regularization parameter multiplied to loss function, a positive number (usually small), required parameter
regularization_parameter = 0.001
# initialization option for the model, 'svd' employs singular value decomposition, optional parameter
initialization_flag = 'svd' #default is 'random'
# maximum number of iterations for the solver, a positive integer, optional parameter
maximum_iteration = 100 #optional, default is 100
# maximum time in seconds for the solver, a positive integer, optional parameter
maximum_time = 300#optional, default is 1000

# Verbosity of the intermediate results
verbosity=0 #optional parameter, valid values are 0,1,2, default is 0
# Whether to compute per iteration train RMSE (and test RMSE, if test data is given)
compute_iter_rmse=True #optional parameter, boolean value, default is False
```

In [7]:
```
## Logging utilities. Please import 'logging' in order to use the following command.
# logging.basicConfig(level=logging.INFO)
```

In [8]:
```
## If both validation and test sets are required
# train, validation, test = python_random_split(df,[0.6, 0.2, 0.2])

## If validation set is not required
train, test = python_random_split(df,[0.8, 0.2])

## If test set is not required
# train, validation = python_random_split(df,[0.8, 0.2])

## If both validation and test sets are not required (i.e., the complete dataset is for training the model)
# train = df
```

Then we will train the data with the RLRMC algorithm with given parameters such as rank of the matrix, regularization parameter, iteration count etc. Then, to calculate the time taken by model to train, we will use the time() function in python.

```
model = RLRMCalgorithm(rank = rank_parameter,
                       C = regularization_parameter,
                       model_param = data.model_param,
                       initialize_flag = initialization_flag,
                       maxiter=maximum_iteration,
                       max_time=maximum_time)
```

```
start_time = time.time()

model.fit(data,verbosity=verbosity)

# fit_and_evaluate will compute RMSE on the validation set (if given) at every iteration
# model.fit_and_evaluate(data,verbosity=verbosity)

train_time = time.time() - start_time # train_time includes both model initialization and model training time.

print("Took {} seconds for training.".format(train_time))
```

```
WARNING:tensorflow:From C:\Users\prath\AppData\Local\Continuum\anaconda3\lib\site-packages\pymanopt\tools\autodiff\_tensorflow.
py:20: The name tf.Session is deprecated. Please use tf.compat.v1.Session instead.

Took 5.9141857624053955 seconds for training.
```

Then we predict the ratings for testing data. We have stored the rating results in the predictions dataframe. Then we have computed the RMSE and MAE values through python functions.

```
In [12]:
# Obtain prediction on the full test set
predictions_ndarr = model.predict(test['userID'].values,test['itemID'].values)
```

```
In [13]:
predictions_df = pd.DataFrame(data={"userID": test['userID'].values, "itemID":test['itemID'].values, "prediction":predictions_nd

## Compute test RMSE
eval_rmse = rmse(test, predictions_df)
## Compute test MAE
eval_mae = mae(test, predictions_df)

print("RMSE:\t%f" % eval_rmse,
      "MAE:\t%f" % eval_mae, sep='\n')
```

```
RMSE:   1.620657
MAE:    1.418156
```

These are the prediction results for some ratings given by different users. We have also stored the model into disk using the joblib library.

```
In [16]:
import joblib
# save the model to disk
filename = 'RLRMC.sav'
joblib.dump(model, filename)
```

```
Out[16]: ['RLRMC.sav']
```

```
In [17]:
thislist1 = [1]
thislist2 = [3342]
model.predict(thislist1, thislist2)
```

```
Out[17]: array([4.85302769])
```

```
In [18]:
thislist1 = [1]
thislist2 = [7522]
model.predict(thislist1, thislist2)
```

```
Out[18]: array([0.98325887])
```

```
In [19]:
thislist1 = [1]
thislist2 = [25363]
model.predict(thislist1, thislist2)
```

```
Out[19]: array([2.12459821])
```

FASTAPI

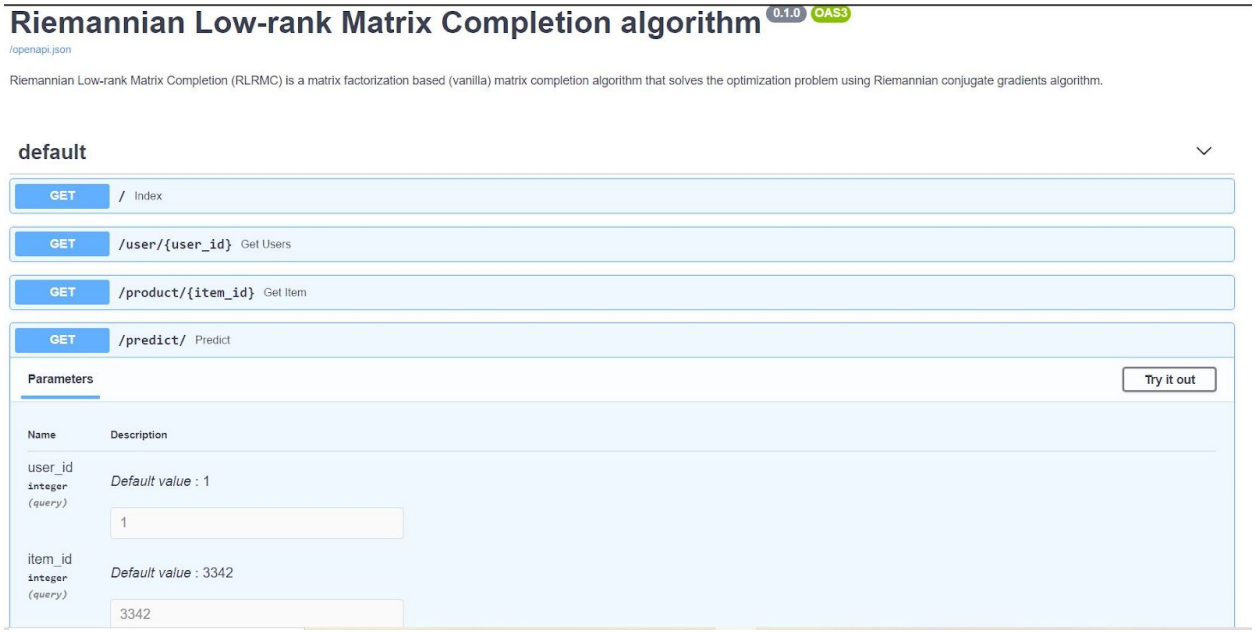
FastAPI is a high performance simple framework for building APIs(Application Programming Interface). It combines the best of all the well known frameworks in Python as well as Javascript. With FastAPI,we can have two different documentations with just a simple code. Adding either docs or redoc to the endpoint /url will result in two different and additional documentations .

We can then navigate to the localhost with **http://127.0.0.1/** address.

The basic workflow includes

- Import Packages
- Create Our Route (/predict)
- Load our Models and Vectorizer
- Receive Input From Endpoint
- Make our predictions

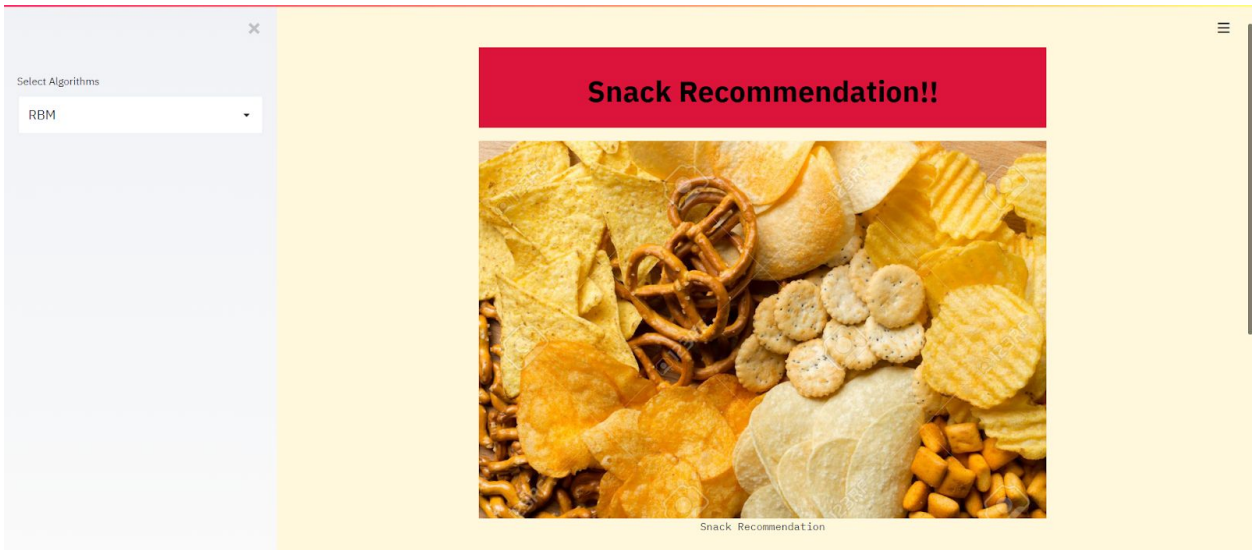
```
import uvicorn
from fastapi import FastAPI,Query
import joblib

# Models
rlrmc = open("RLRMC.sav","rb")
rlrmc_model = joblib.load(rlrmc)

# init app
app = FastAPI(title="Riemannian Low-rank Matrix Completion algorithm",
              description='''Riemannian Low-rank Matrix Completion (RLRMC) is a matrix factorization
              version="0.1.0",)

# Routes
@app.get('/')
async def index():
        return {"text":"Hello"}

@app.get('/user/{user_id}')
async def get_users(user_id:int):
        return {"user_id":user_id}

@app.get('/product/{item_id}')
async def get_item(item_id:int):
        return {"item_id":item_id}

# ML Aspect
@app.get('/predict/')
async def predict(user_id:int = 1, item_id:int = 3342):
        user_list = []
        item_list = []
        user_list.append(user_id)
        item_list.append(item_id)
        prediction = rlrmc_model.predict(user_list, item_list)
        if prediction[0] > 0:
                result = prediction[0]
        else:
                result = "Not Found"

        return {"prediction":result}

if __name__ == '__main__':
        uvicorn.run(app,host="127.0.0.1",port=8000)
```

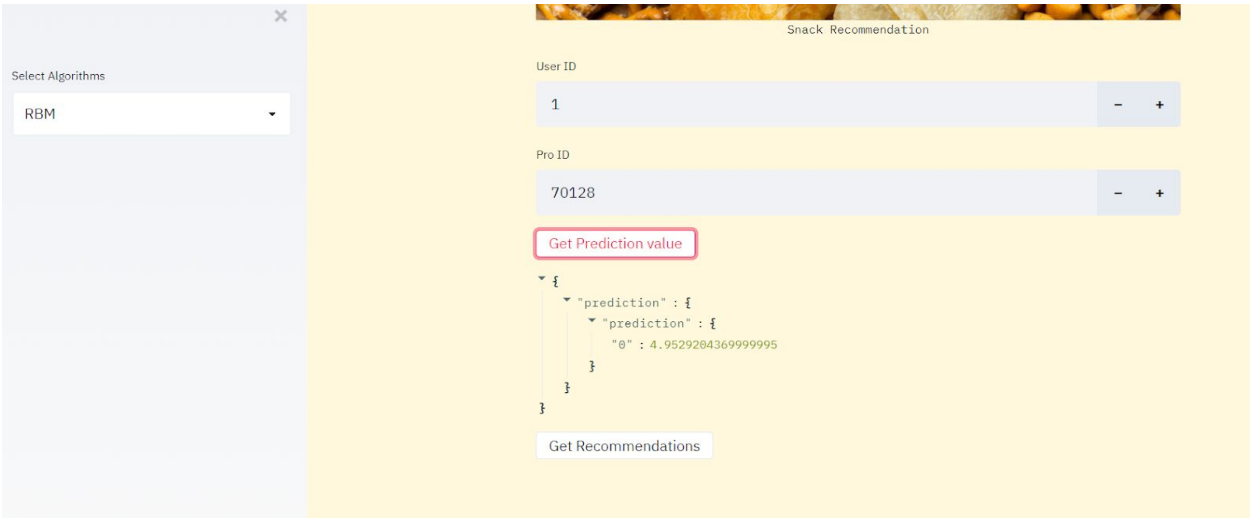


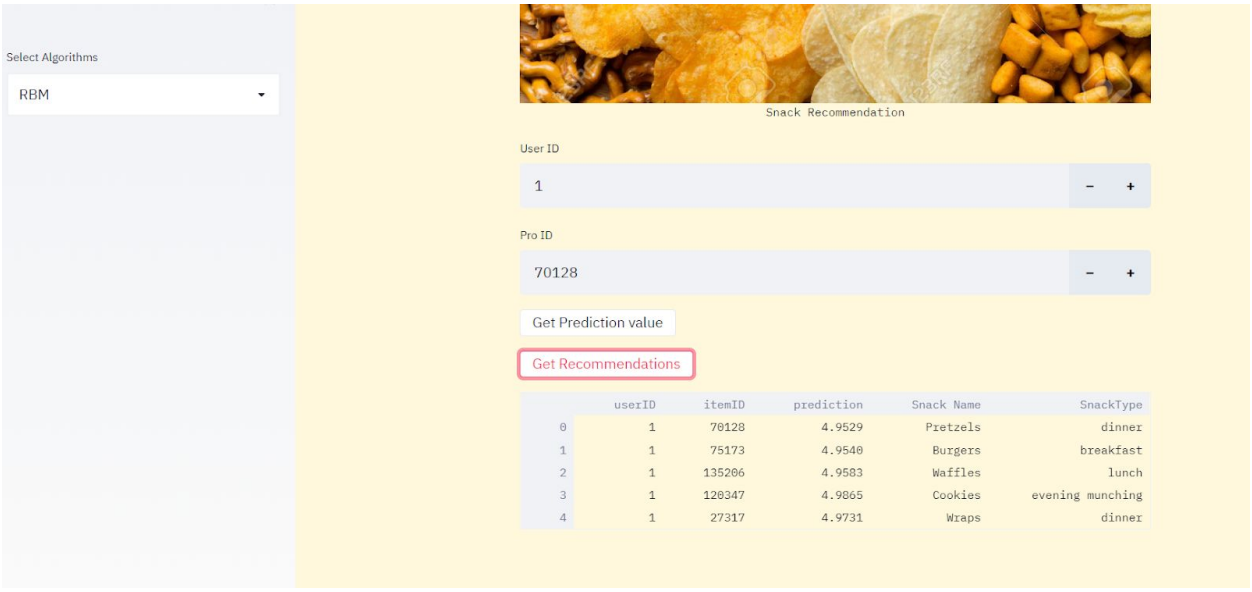
STREAMLIT

Streamlit is a tool that allows engineers to quickly build highly interactive web applications around their data, machine learning models, and pretty much anything.

Below are the snippets from the streamlit app:

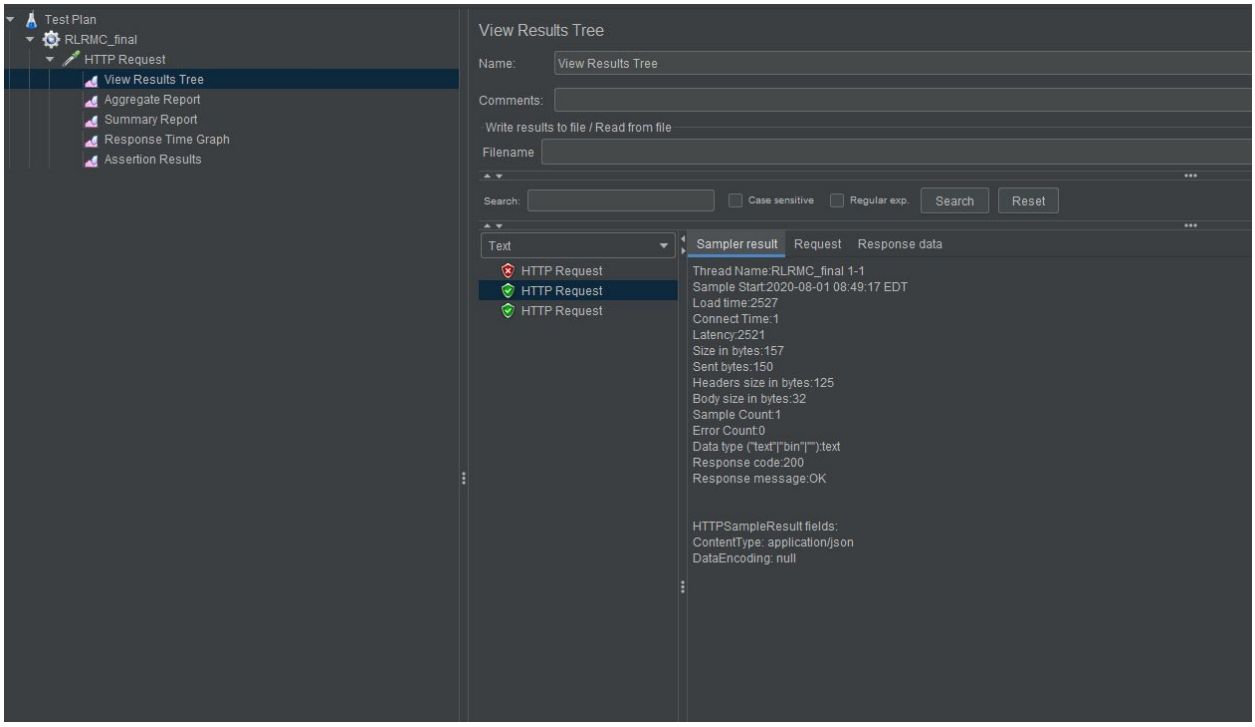


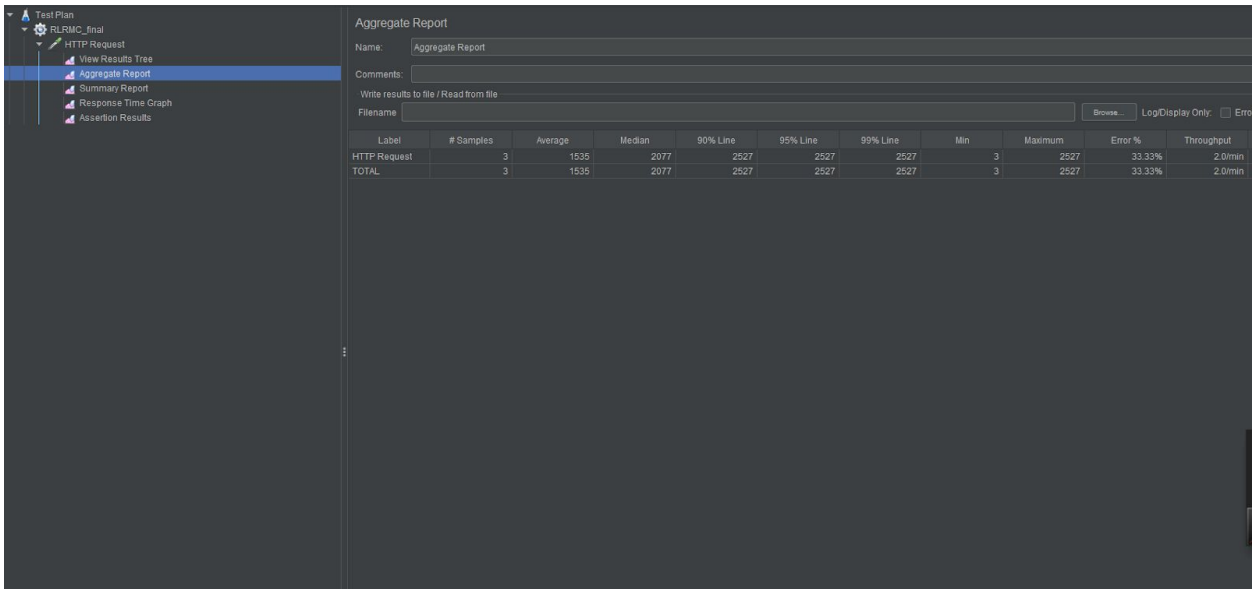


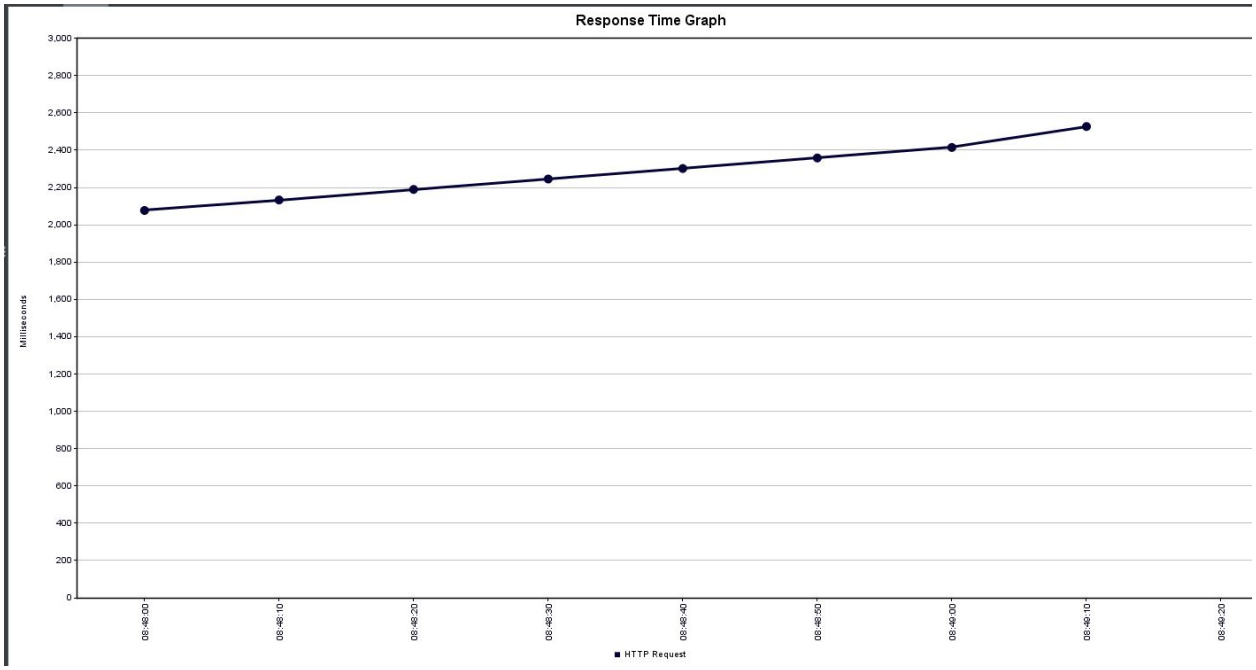


JMETER

Apache JMeter may be used to test performance both on static and dynamic resources, Web dynamic applications. It can be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze overall performance under different load types. Apache JMeter features Ability to load and performance test many different applications/server/protocol types such as Web: HTTP/HTTPS, SOAP/REST API, Databases etc.


Test Plan
RLRMC_final
HTTP Request
View Results Tree
Aggregate Report
Summary Report
Response Time Graph
Assertion Results
View Results Tree
Name:
View Results Tree
Comments:
Write results to file / Read from file
Filename
Search:
Case sensitive
Regular exp.
Search
Reset
Text
Sampler result
Request
Response data
HTTP Request
HTTP Request
HTTP Request
Thread Name:RLRMC_final 1-1
Sample Start:2020-08-01 08:49:17 EDT
Load time:2527
Connect Time:1
Latency:2521
Size in bytes:157
Sent bytes:150
Headers size in bytes:125
Body size in bytes:32
Sample Count:1
Error Count:0
Data type ("text"|"bin"|""):text
Response code:200
Response message:OK
HTTPSampleResult fields:
ContentType: application/json
DataEncoding: null


Test Plan
RLRMC_final
HTTP Request
View Results Tree
Aggregate Report
Summary Report
Response Time Graph
Assertion Results
Aggregate Report
Name:
Aggregate Report
Comments:
Write results to file / Read from file
Filename
Browse...
Log/Display Only:
Label
Samples
Average
Median
90% Line
95% Line
99% Line
Min
Maximum
Error %
Throughput
HTTP Request 3 1535 2077 2527 2527 2527 3 2527 33.33% 2.0/min
TOTAL 3 1535 2077 2527 2527 2527 3 2527 33.33% 2.0/min



CONCLUSION

We have built recommendation systems using algorithms such as Riemannian Low-rank Matrix Completion (RLRMC) and Restricted Boltzmann Machine (RBM) and compared the models on the basis of results obtained. We have evaluated the Accuracy, RMSE (Root Mean Square Error) and MAE (Mean Absolute Error) values for both models, and we noticed that RBM outperformed the RLRMC Algorithm. Even during the loading test through JMETER, RBM was faster than RLRMC.