DEPP LEARNING BASED RECOMMENDATION SYSTEM

Final Report

Information Technology Capstone Project

COMP5703

CS26-4

Client: Caren Han

Group Members

1. Kaijia Chen (480218353)
2. Wentao Wu (4804168548)
3. Wenjing Deng (470349281)
4. Hefei Chen (470488188)
5. Zhou Sha (480322650)
6. Zaiwo Liu (490191684)

ABSTRACT

Recommend systems are an essential means of improving a user's web experience. With the increasing data, the recommended system model also requires the ability to process large amounts of data quickly. While deep learning can help the traditional model to process data quicker and train model better. Collaborative filtering is widely used in modern recommend systems. There are a large number of methods to improve the accuracy and ability of recommended items. In our project, we research three models of collaborative filtering that are all based on deep learning. First, Kernelized Synaptic Weight Matrices model, it is a novel neural network architecture, in which weight matrices are reparametrized in terms of low-dimensional vectors, interacting through kernel functions. Second, Graph Convolutional Matrix Completion, it is a graph auto-encoder framework based on differentiable message passing on the bipartite interaction graph. Third, Ranking-Critical Training, it is a new method for training collaborative filtering models based on actor-critic reinforcement learning, to directly optimize the non-differentiable quality metrics of interest. We deploy the code on Google Colab, break down the code to each block and make annotations to help read the code. We use MovieLens dataset to test these three models. They all achieve state-of-the-art performance in a collaborative filtering task. Compared with these three models, we chose the Kernelized Synaptic Weight Matrices model to build a REST API. Because it performs best in the ML-1M dataset.

TABLE OF CONTENTS

1. INTRODUCTION

With the rapid development of the Internet industry, there are more and more ways to obtain information. People have gradually changed from active access to passive access to information, and the amount of data is also growing exponentially. Therefore, in the explosive information, to obtain effectiveness, targeted information has become the needs of users. The recommendation system emerges as the times require. It is a meaningful way when it comes to improving the user's network experience.

The recommendation system is to mine the potentially interested objects of each user by establishing the binary relationship between users and information products, and then carry out a personalized recommendation by using the existing selection process or similarity relationship. Its essence is information filtering. Collaborative filtering, in which patterns across similar users and items to predict user preferences, is a widely used technology in recommendation systems. The underlying assumption of it is that if a user A has the same opinion as a user B on an issue, user A is more likely to have the same opinion with user B on a different issue than that of a randomly selected user.

We aim to build a collaborative filtering recommendation system based on deep learning. We have trained three deep learning models and evaluated their performances, then selected one model with the best performance to create an API for the website or platform to call. We used MovieLens to train our model, which can be applied to the recommendation system of movie websites. In addition, we can immediately retrain our models to meet the recommended requirements according to the needs of the websites or platforms.

2. RELATED LITERATURE

2.1 Collaborative Filtering

Collaborative filtering has significant advantages over traditional content-based filtering. Mainly because the machine will make mistakes during the scanning of keywords when processing content, and collaborative filtering is not based on this. In the essay Explaining Collaborative Filtering Recommendations, it mentions that CF

can filter a more comprehensive form of data, including text, music and fund trends. Moreover, there are many things that cannot be expressed in words, such as taste or quality, can be expressed intuitively through people's favourite level, which greatly enhances the superiority of collaborative filtering (Herlocker, 2001).

The CF algorithm can be divided into two steps (Paolo & Paolo, 2004). The first step is the similarity assessment, which involves comparing the ratings provided by pairs of users (rows in the matrix) to calculate their similarity. Usually, we treat the Pearson correlation coefficient as the most effective similarity assessment technique. The second step is the actual rating forecast, which includes predicting the rating of the active user's activity. The predicted rating is a weighted sum of ratings given by other users to the item, where the weight is the similarity rate of the active user to other users. In this way, ratings expressed by very similar users have a greater impact on the rating predicted by active users. The formula for the second step is as follows:

$$p_{a,i} = \overline{r}_a + \frac{\sum_{u=1}^{k} w_{a,u(r_{u,i} - \bar{r}_u)}}{\sum_{u=1}^{k} w_{a,u}}$$

From the formula, represents the predicted rating that active user a may provide for item i, is the average of the ratings provided by user u, w is the user similarity weight of a and u calculated in the first step and k is the number of users who consider the rating of item i in the weighted sum (called neighbours).

2.2 Information Filtering

Information filtering is a general term used to describe a series of processes that transfer information to users who need it. (Hanani, Shapira & Shovaal, 2001) Equivalent to the traditional database, the information filtering system is a non-structured or semi-structured information system. It mainly deals with text information and often has to deal with vast amounts of data. The purpose of the information filtering system is to select from a large number of dynamically generated information and show it to those users who meet the information needs. (Ciaccia & Patella, 2011) It is a behaviour that only keeps specific data in the input data stream according to the given information needs. It refers to the selection of information, satisfying the user's interest from the dynamic information flow, and the user's interest will not change in an extended period. In the field of content security, information filtering is to provide the sufficient flow of information, eliminate or reduce the harm caused by excessive

information, information confusion and information abuse. However, in the current research stage, it is still in the primary stage of human research. It is one of the main tasks of information filtering in the field of content security to remove inappropriate information for users.

At present, the problem is not to find the best way to filter information, but the idea that these systems need to learn the information needs of users independently. This is not only because they automate the filtering process, but also because of the construction and adaptation of filters. (Tauritz, 2002) Its branches, such as statistics, machine learning, pattern recognition and data mining, are the basis of developing experience-based information filters. In order to make the learning process possible, part of the information must be pre-filtered, which means that there are positive and negative examples. It is called training data, which can be generated by experts or through the feedback of ordinary users. (Wang, Meng & Hu, 2014)

2.3 Maximum Likelihood Estimation

Maximum likelihood estimation (MLE) is a statistical method, which is used to find the parameters of the correlation probability density function of a sample set. The maximum likelihood method uses the probability model explicitly, and its goal is to find the phylogenetic tree that can generate observation data with high probability. (Rossi & Richard, 2018) The maximum likelihood method is the representative of a kind of complete statistics-based system spanning-tree reconstruction method. (Hendry & Nielsen, 2012) This method considers the probability of each nucleotide substitution in each sequence alignment.

This is the principle of MLE. Given a probability distribution D, assuming that its probability density function (continuous distribution) or probability aggregate function (discrete distribution) is f_D, and a distribution parameter θ, we can extract a sample X_1, X_2,..., Xn with n values from this distribution. By using f_D, we can calculate its probability (Eliason, 2009):

$$P = (x_1, x_2, \ldots, x_n) = f_D(x_1, x_2, \ldots, x_n|\theta)$$

However, we may not know the value of θ, although we know that the sampling data is from distribution D. A natural idea is to extract a sample X_1, X_2,..., Xn with n values from this distribution, and then use these samples to estimate θ.

Once we get it, we can find an estimate of θ. Maximum likelihood estimation will look for the most possible value about θ. This method is just different from some other estimation methods, such as the unbiased estimation of θ. The unbiased estimation may not output the most possible value, but it will output a value that is neither overestimated nor underestimated.

To implement MLE mathematically, we first define the possibility (Eliason, 2009):

$$lik(\theta) = f_D(x_1, x_2, ..., x_n|\theta)$$

And on all values of θ, maximize this function. This value, which maximizes the probability, is called the maximum likelihood estimation of θ.

2.4 User-based CF and item-based CF

User-based CF is a simple algorithm that predicts what the current user will like by finding other users who behave like the behaviour of the current user and using their ratings on other items to do the recommendation. In the thesis Evaluating collaborative filtering recommender systems, the author gave such an example. Now judge whether to recommend product A to Mary. It is known that Mary does not rate for product A, but gives products B, C and D different rates. User-based CF searches for similar users based on Mary's rates for B, C, D and collects their rates for A products. These users' ratings for product A are weighted by their level of agreement with Mary's ratings in order to predict Mary's preference (Terveen, Herlocker, & Konstan, 2004).

But there are still problems in user-based CF. The major problem is the scalability of the collaborative filtering algorithms. These algorithms may have good performance when searching for thousands of potential neighbours in real-time, but when it comes to meet the demand for modern systems is to search for potential neighbours in the tens of millions of levels, it has poor performance (Sarwar, Karypis, Konstan, & Riedl, 2001). To solve those problems, we introduce the item-based CF.

Item-based CF uses the similarity between the rating patterns of the projects to make recommendations. If two projects tend to make the same users like and dislike them, then they are similar and users are expected to have similar preferences for similar projects. Thus, in its overall structure, the approach is similar to earlier content-

based recommendations and personalization methods, but project similarity is derived from user preference patterns rather than from project data. In the original form, project-based CF does not solve any problems: finding the most similar project to generate predictions and recommendations is still necessary. In systems with more users than projects, it allows neighbourhood discovery in the smaller of the two dimensions and it provides a major performance boost by well pre-computing the similarity matrix (Terveen, Herlocker, & Konstan, 2004).

2.5 Deep Collaborative Filtering

Recommender system ranges from content-based recommender system to collaborative filtering recommendations, while collaborative filtering extends from basic user-based collaborative filtering, item-based collaborative filtering, to model-based collaborative filtering. Perhaps deep learning is not as good as the image processing algorithm in the recommendation system, but deep learning does play a contributory effect. DCF is able to extract features directly from the content, and it is easy to process noise data. In addition, we can use RNN cyclic neural networks to model dynamic or sequence data, which to a high degree solves the cold-start problem caused by newly registered users (Jian, Jianhua, Kai, Yi, & Zuoyin, 2016).

The flow chart below shows the Deep Crossing model which is proposed by Microsoft. By adding the embedding layer, it is possible to transform the sparse features into low-dimensional dense features. Then use the Concat Layer to connect the segmented feature vectors. The next step is to complete the feature combination and transformation through the multi-layer neural network, and finally calculate the CTR rate through the scoring layer (Ying, T. Ryan, Jian, Haijing, & Yu, 2016). Unlike the classic DNN, the multilayer perceptron used by Deep crossing is composed of a residual network.

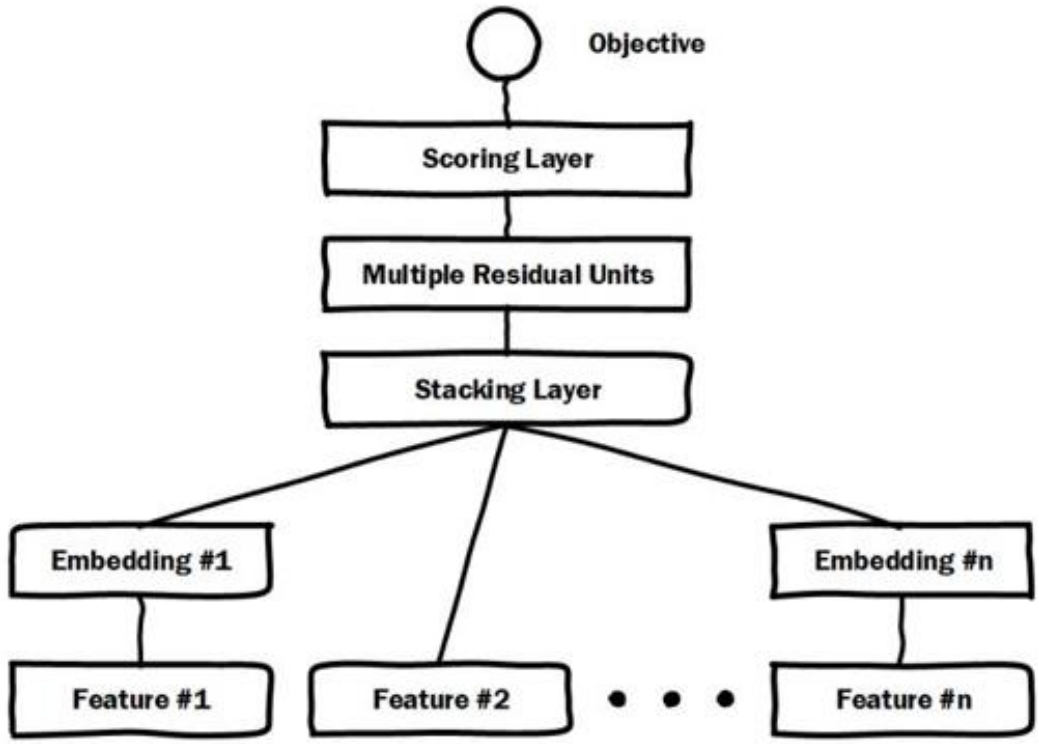


Figure 2.5-1: Embedding solution of DCF

FNN is another method to do the embedding initialization. The flow chart is shown below (Weinan, Tianming, & Jun 2016):

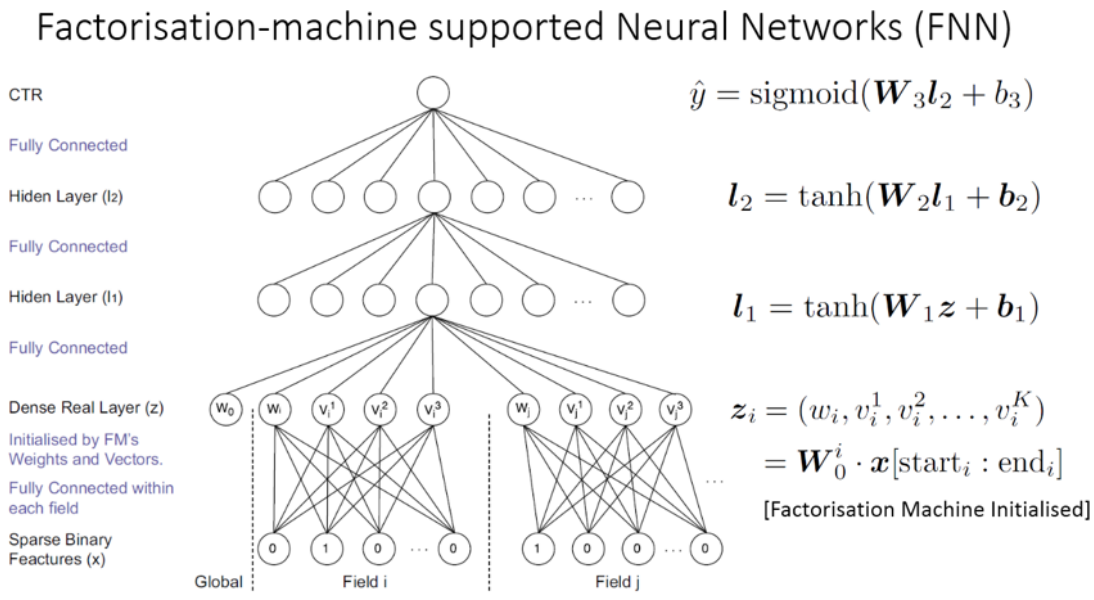


Figure 2.5-2: Basic algorithm and flow chart for FNN

Actually, we know that the features x of the dataset used in the recommender system is definitely discrete and sparse. In FNN we just divide x into N fields. For

each field, there's only one value rate 1 and the rest of those we rate it as 0. So the field i is calculated as shown: $x[start_i: end_i]$. W_0^i is the matrix for embedding and then put the result into the full-connected neural network.

2.6 Baseline Algorithm for Recommendation System

The kernel layer it creates can assign accurate and based on a scan of the original dataset. For the baseline algorithm, the thesis On the Difficulty of Evaluating Baselines: A Study on Recommender System (Rendle, 2019) gives a brief introduction. Different from the collaborative filtering, the idea of the baseline algorithm is to set up a baseline and put user bias and film bias into the equation. The formula for this algorithm is as follows:

$$b\text{ui} = \mu + b\text{u} + b\text{i}$$

In the above formula, the result of the equation is to give an estimate of user1 for item1 in the prepared baseline model. μ is the average of all users rating the movie; bu is the user bias (bu will be negative if the user asks for a higher and the score is relatively low; conversely, if the user frequently corrects many item scores, then bu is positive); bi It is the project deviation that reflects the popularity of the project(Berg, Kipf, & Welling, 2017). According to the paper, if user1 is unknown, the deviation bu is assumed to be zero. The same applies to item1. Although the Baseline model is simple, it actually contains personalized information about users and projects. In the face of large-scale data, simple algorithms can reduce a lot of computing time.

3. RESEARCH/PROJECT PROBLEMS

3.1 Research/Project Aims & Objectives

By analyzing the project description file and summarizing the results of multiple meetings with the client, the project's ultimate goal can be interpreted as designing and researching the recommended algorithm and providing the appropriate API for the website call that requires the recommendation system. The short-term goals of the project are divided into three phases. First, find the appropriate recommendation algorithms and test them to ensure their availability. Based on the determination of availability, evaluate the performance of each recommendation algorithm and select one of the best performing algorithms. Finally, build an API for the selected recommendation algorithm for the website to call.

3.2 Research/Project Questions

In order to achieve the goals and objectives of the project, the requirements of the project and the problems to be solved by the client are first defined and explained. After discussing with the client at the conference, there are several issues that need to be focused on.

First, at the beginning of the project, the number of recommended algorithms to be studied needs to be clearly defined. The excessive number of algorithms will seriously increase the workload of the project and the difficulty of screening the algorithm. The smaller number of algorithms will result in a lack of sufficient sample support for the screening algorithm so that the algorithm is not good enough. In this project, the customer needs a recommendation model based on collaborative screening and initially needs to select three different algorithms or models for research and analysis.

Second, identifying areas of project research is also one of the key issues. Although this project is to provide the corresponding algorithm for the website that needs the recommendation system, selecting the data set and the target website of the appropriate industry or field will help to narrow the scope and workload of the initial algorithm selection. In this project, the customer requests that the selected data set and the target website be related to the movie. In other words, the project will focus on algorithms related to recommending movies.

3.3 Research/Project Scope

3.3.1 Research/Project Charter

<table>
<tr><td colspan="2">Project title: Deep Learning based Recommendation System</td></tr>
<tr><td>Project Start Date: 2019. 8. 2</td><td>Project Finish Date: 2019. 11. 16</td></tr>
<tr><td colspan="2">Project Background:
Currently, people are living in the era of information explosion. When you are online, an effective recommendation system can help people find the relevant content they need more quickly.</td></tr>
<tr><td>Project Purpose (Objective):
This project is called a recommendation system based on deep learning. The goal of the project is to study the recommended algorithms and build the corresponding APIs for sites that need a recommendation system.</td><td>Main Project Success Criteria:
The recommended algorithm and related APIs must not only be completed in full but also meet the requirements of the project objectives. The recommendation system should also be tested and evaluated, and all support and related documentation should be provided and</td></tr>
</table>

	authenticated.
Project Deliverable: Project-related deliverables: The software system includes relevant code for the recommended algorithm, a link to the code in Google Colab, and a complete recommendation algorithm API. Project Management Deliverables: Support document, Scope description, Gantt chart, Project proposal report, Project progress report	
Cost Information: This project does not require additional funding support	

3.3.2 Research/Project Scope Statement

This project scope statement serves as a baseline document for defining the scope of Deep Learning based Recommendation System, ensuring a common understanding of the project scope among all the stakeholders. All the changes to the scope statement must be vetted through all the members in our team and make sure the changes are beneficial to the implementation of this project. This version's date of scope statement is September 5, 2019.

3.3.3 Project Justification

The scope of the project is to provide an API built by highly efficient recommendation algorithms for sites that require a recommendation system. The project consists primarily of analyzing and researching appropriate recommendation algorithms and building APIs for recommended models of optimal performance.

The research and analysis of the recommended algorithm is the main work at the beginning of the project. According to the customer's request, three models were selected as the recommended algorithm for research. They are key training for amortization of Collaborative Filtering (RaCT), Graph Convolution Matrix Completion (GC-MC) and Nucleation Synaptic Weight Matrix (KernelNet). The main work of the early days was to study the basic theory of the three models, test the usability of the model-related code, and evaluate the performance of the model.

After completing the initial work, the latter task is to compare the performance of the three models and select the best model. After that, we need to build the appropriate API for this model to simplify the demanding website invocation model. The recommendation system is an important goal of the project. Based on the related

collaborative filtering recommendation model, the user who browses the webpage is provided with items related to the browsed project, so that the user can find and query the required project information.

The data set used in the project is all data related to the movie, including Movie Lens 100K, Movie Lens 1M, Movie Lens 10M and so on.

4. METHODOLOGIES

4.1 Kernelized Synaptic Weight Matrices

4.1.1 Methods

Based on the essay Kernelized Synaptic Weight Matrices, we use advanced deep learning for model-based collaborative filtering. Unlike traditional deep learning collaborative filtering, this essay begins with a new term, KernelNet, and introduces optimization methods that reduce test errors and loss functions. Ultimately, the entire model consists of an RBF kernel, a supporting RBF kernel, and a fully connected neural network. The formula and flowchart are as follows:

$$x_j^{(l)} = f_j\left(\sum_i a_i^{(l)} K\left(\overrightarrow{u_i}^{(l)}, \overrightarrow{v_j}^{(l)}\right) x_i^{(l-1)}\right)$$

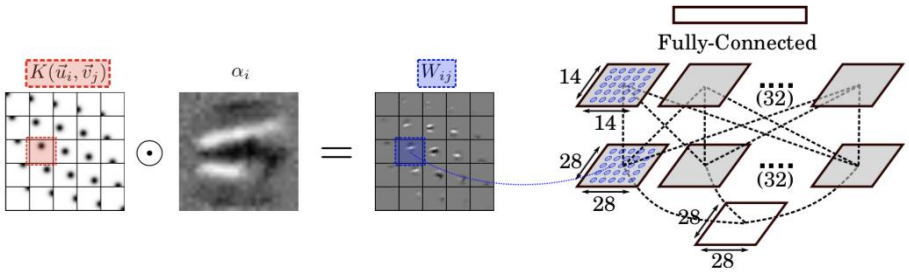


Figure 4.1.1-1: The whole flowchart for the Kernelized Synaptic Weight Matrices

In the above formula, $\vec{u}$ and $\vec{v}$ are initiated in Gaussian distribution and are used to build a finite support RBF kernel. $\vec{u}$ is the free parameter in data dot and $\vec{v}$ is the one in the model we set in initialling model part. The finite support RBF kernel aims at computing the tanh value of $\vec{u}$ and $\vec{v}$ and ensure the value is no less than zero. $a_i^{(l)}$ is the learning rate we set in hyper- parameter part. By producing $a_i^{(l)}$ with $K\left(\overrightarrow{u_i}^{(l)}, \overrightarrow{v_j}^{(l)}\right)$ we can get the weight W.

After doing this, we use a fully connected neural network to compute the output. Unlike the general neural network, the weight W here comes from the solution we mentioned before. Then use the output Z as the new weight to compute the fully connected neural network. The output in this section is a layer of output and a regularization term. The regularization term is used to calculate the loss function, which plays an important role in evaluating part. The formula and the flowchart are shown below.

$$
\begin{aligned}
x_j^{(l)} &= \sum_i \alpha_i K(\vec{u}_i, \vec{v}_j) x_i^{(l-1)} = \sum_i \alpha_i \langle \phi(\vec{u}_i), \phi(\vec{v}_j) \rangle x_i^{(l-1)} \\
&= \sum_i \alpha_i \vec{u}_i^{*T} \mathbf{M}_\phi \vec{v}_j^{*} x_i^{(l-1)} = \sum_i \alpha_i \left[\sum_{h,k} u_{ki}^* M_{kh} v_{jh}^* \right] x_i^{(l-1)} \\
&= \sum_{h,k} v_{jh}^* M_{kh} \underbrace{\left[\sum_i \alpha_i u_{ki}^* x_i^{(l-1)} \right]}_{z_k} = \sum_k z_k \left[\sum_h v_{jh}^* M_{kh} \right] \\
&= \sum_k z_k \Omega_{jk} \qquad (4)
\end{aligned}
$$

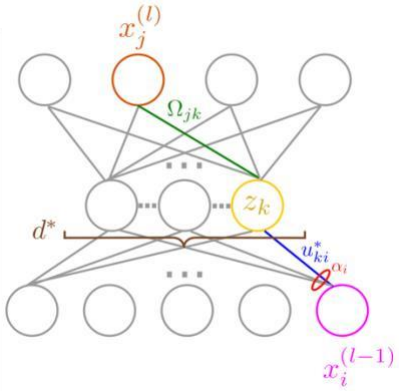


Figure 4.1.1-2: The formula and flowchart in building CNN part

After building the fully connected neural network, it directly imports the Gaussian kernel in tensorflow library to active the output. The whole process can be explained more easily according to the screenshot of the code part below:

Overview:

Once the algorithm trains the input with support RBF Kernels, it use full connected neural network to comupute the weght and then use the guaussion kernel to activate the output.

Class Name: kernel_layer

Constructor Input: batch,channels,activation,n_dim,n_hid

Returns: layer output,regularization term

Used in: training data

```
def kernel_layer(x, n_hid=500, n_dim=5, activation=tf.nn.sigmoid, lambda_s=lambda_s,
                 lambda_2=lambda_2, name=''):
    """
    a kernel sparsified layer

    :param x: input [batch, channels]
    :param n_hid: number of hidden units
    :param n_dim: number of dimensions to embed for kernelization
    :param activation: output activation
    :param name: layer name for scoping
    :return: layer output, regularization term
    """

    # define variables
    with tf.variable_scope(name):
        W = tf.get_variable('W', [x.shape[1], n_hid])
        n_in = x.get_shape().as_list()[1]
        u = tf.get_variable('u', initializer=tf.random.truncated_normal([n_in, 1, n_dim], 0., 1e-3))
        v = tf.get_variable('v', initializer=tf.random.truncated_normal([1, n_hid, n_dim], 0., 1e-3))
        b = tf.get_variable('b', [n_hid])

    # compute sparsifying kernel
    # as u and v move further from each other for some given pair of neurons, their connection
    # decreases in strength and eventually goes to zero.
    w_hat = kernel(u, v)

    # compute regularization terms
    #tensorflow can devide the l2 regulation loss value in order to make the result more accuracy.
    sparse_reg = tf.contrib.layers.l2_regularizer(lambda_s)
    sparse_reg_term = tf.contrib.layers.apply_regularization(sparse_reg, [w_hat])

    l2_reg = tf.contrib.layers.l2_regularizer(lambda_2)
    l2_reg_term = tf.contrib.layers.apply_regularization(l2_reg, [W])

    # compute output
    W_eff = W * w_hat
    y = tf.matmul(x, W_eff) + b
    y = activation(y)
    return y, sparse_reg_term + l2_reg_term
```

Figure 4.1.1-3: The core algorithm to build the kernel layer

In order to reduce the regularized squared- error, we adapt the L-BFGS Optimizer. The code part is shown below.

L-BFGS Optimizer

Overview: Use L-BFGS Optimizer to minimize a regularized square- error the regularization term added to the cost is $R=\lambda(W2+V2)$

Constructor Input: L-BFGS optimizer

Returns: optimizer

Used in: training data

```
[ ] # Instantiate L-BFGS Optimizer
    optimizer = tf.contrib.opt.ScipyOptimizerInterface(loss, options={'maxiter': output_every,'disp': verbose_bfgs,'maxcor': 10}
                                                       ,method='L-BFGS-B')
```

Figure 4.1.1-4: L-BFGS Optimizer function

To train the model, we create 20 epochs to get a lower RMSE value. In addition, we also trained the model with the training set and the testing set to achieve the highest accuracy through the loop test training. From the results of the last statistics, the root means square is 0.88827 at the first starting point and finally falls to 0.82813.

Training and validation loop

Overview:

Use rmse to define the efficiency of the model:

$Ermse = \sqrt{\sum}(pi - ri)^2 / N,$

where pi is the predicted rating, ri is the true rating. N is the number of validation samples.

Constructor Input: dataset

Returns: Ermse,validation error,train error

Used in: evaluate the efficiency of the model

```
[ ]  init = tf.global_variables_initializer()
     with tf.Session() as sess:
         sess.run(init)
         for i in range(int(n_epoch / output_every)):
             optimizer.minimize(sess, feed_dict={R: tr}) #do maxiter optimization steps
             pre = sess.run(prediction, feed_dict={R: tr}) #predict ratings

             error = (vm * (np.clip(pre, 1., 5.) - vr) ** 2).sum() / vm.sum() #compute validation error
             error_train = (tm * (np.clip(pre, 1., 5.) - tr) ** 2).sum() / tm.sum() #compute train error

             print('.-^-._' * 12)
             print('epoch:', i, 'validation rmse:', np.sqrt(error), 'train rmse:', np.sqrt(error_train))
             print('.-^-._' * 12)

         with open('summary_mllm.txt', 'a') as file:
             for a in sys.argv[1:]:
                 file.write(a + ' ')
             file.write(str(np.sqrt(error)) + ' ' + str(np.sqrt(error_train))
                        + ' ' + str(seed) + '\n')
             file.close()
```

Figure 4.1.1-5: Training and validation loop

4.1.2 Data Collection

The data we train the deep learning collaborate filtering method is MovieLens-1m. This dataset holds 1000,209 scoring records for approximately 3,900 movies from 6040 MovieLens users. The dataset can be downloaded by the following link:

http://www.grouplens.org/system/files/ml-1m.zip

The MovieLens dataset has three files. For the file “rating.dat”. it contains all the ratings and they are stored in “UserID::MovieID::Rating::Timestamp” format. This file is the core data we use in the deep learning collaborate filtering part.

Besides the file “rating.dat”, there still exist file “users.dat” and “movies.dat”, which are essential in creating API. The “users.dat” file introduces the user’s gender, age, occupation and zip-code based on his or her user ID. The “movies.dat” file introduces the movie's title and the genres it belongs to is based on its movie ID. By contracting those two data files, we can make the prediction from the model more reasonable and more meaningful.

4.1.3 Data Analysis

For the input part, the raw data set is transformed by loadData function and the final version is a sparse matrix, which means that most of the data in the matrix are

zero or NA value. loadData function also deals with the data partition part by dividing the data into training parts and testing parts.

Data partition

Overview: DataLoader for preparing the data, split it into training set (tr) and validation set (vr)

Class Name: data partition

Constructor Input: dataset

Returns: training set and validation set

Used in: class kernel

```
import numpy as np
from time import time
def loadData(path='./', valfrac=0.1, delimiter='::', seed=1234,
             transpose=False):
    np.random.seed(seed)

    tic = time()
    print('reading data...')
    data = np.loadtxt(path, skiprows=0, delimiter=delimiter).astype('int32')
    print('data read in', time() - tic, 'seconds')

    n_u = np.unique(data[:, 0]).shape[0]  # number of users
    n_m = np.unique(data[:, 1]).shape[0]  # number of movies
    n_r = data.shape[0]  # number of ratings
    # these dictionaries define a mapping from user/movie id to to user/movie number (continuous from zero)
    udict = {}
    for i, u in enumerate(np.unique(data[:, 0]).tolist()):
        udict[u] = i
    mdict = {}
    for i, m in enumerate(np.unique(data[:, 1]).tolist()):
        mdict[m] = i

    # shuffle indices
    idx = np.arange(n_r)
    np.random.shuffle(idx)
    trainRatings = np.zeros((n_u, n_m), dtype='float32')
    validRatings = np.zeros((n_u, n_m), dtype='float32')
    for i in range(n_r):
        u_id = data[idx[i], 0]
        m_id = data[idx[i], 1]
        r = data[idx[i], 2]
        # the first few ratings of the shuffled data array are validation data
        if i <= valfrac * n_r:
            validRatings[udict[u_id], mdict[m_id]] = int(r)
        # the rest are training data
        else:
            trainRatings[udict[u_id], mdict[m_id]] = int(r)

    if transpose:
        trainRatings = trainRatings.T
        validRatings = validRatings.T
```

Figure 4.1.3-1: Data Partition

We treat NA value as the missing value and in order to compute easily in the neural network part, we define them as zero.

```
In [3]: # load data
        tr, vr = loadData('./ml-1m/ratings.dat', delimiter='::',
                          seed=seed, transpose=True, valfrac=0.1)
        print(tr,vr)
        tm = np.greater(tr, 1e-12).astype('float32')  # masks indicating non-zero entries
        vm = np.greater(vr, 1e-12).astype('float32')

        n_m = tr.shape[0]  # number of movies
        n_u = tr.shape[1]  # number of users (may be switched depending on 'transpose' in loadData)

        reading data...
        data read in 5.096928834915161 seconds
        loaded dense data matrix
        [[5. 0. 0. ... 0. 0. 3.]
         [0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]
         ...
         [0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]] [[0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]
         ...
         [0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]]
```

Figure 4.1.3-2: The initial input for the kernel layer function

The purpose of the recommendation system is to derive a complete matrix suitable for user scoring logic based on existing incomplete sparse matrices. For the

output section, the model reaches its best performance after iterating 20 times and saves the minimum RMSE value and training time in the summary_ml1m.txt file.

Obviously, in order to create the API to benefit the real world, the input and output are not we want. For the recommendation system, the input should be the user or user ID. The output should be our recommended item. Therefore, we save the model and use the validation set to predict movie ratings.

```
In [10]: # Training and validation loop
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    for i in range(int(n_epoch / output_every)):
        optimizer.minimize(sess, feed_dict={R: tr}) #do maxiter optimization steps
        pre = sess.run(prediction, feed_dict={R: tr}) #predict ratings

        error = (vm * (np.clip(pre, 1., 5.) - vr) ** 2).sum() / vm.sum() #compute validation error
        error_train = (tm * (np.clip(pre, 1., 5.) - tr) ** 2).sum() / tm.sum() #compute train error

        print('.-^-._' * 12)
        print('epoch:', i, 'validation rmse:', np.sqrt(error), 'train rmse:', np.sqrt(error_train))
        print('.-^-._' * 12)
    saver.save(sess, './MyModel',global_step=20)
```

Figure 4.1.3-3: save the model to run the new code

After we reloaded the model and trained the validation set, we fetch the prediction. We save it in the pre.txt file and we can use the matrix in nd-array type to show the recommended items.

```
# Training and validation loop
init = tf.global_variables_initializer()
with tf.Session() as sess:
    sess.run(init)
    for i in range(int(n_epoch / output_every)):
        optimizer.minimize(sess, feed_dict={R: tr})
        pre = sess.run(prediction, feed_dict={R: tr})
```

```
np.savetxt("pre.txt",pre,delimiter=",")
```

```
m = np.loadtxt("pre.txt",delimiter=",")
```

Figure 4.1.3-4: The prediction for the validation set

And the final prediction for the validation set is shown below.

```
print(m)
```

```
[[4.37066984 4.03864384 3.43791032 ... 3.67340851 4.322824   3.33784342]
 [3.55417371 3.2077558  3.30317831 ... 3.3017323  3.13964152 1.47281551]
 [3.77192307 3.46730494 3.62082195 ... 3.0478971  2.74457526 1.02589095]
 ...
 [4.01643944 3.44752359 3.22458839 ... 3.39919162 3.76886725 3.73099494]
 [3.78178906 3.09745336 2.9608357  ... 3.36495495 3.55637765 3.92714143]
 [3.97113824 3.52288532 3.29823804 ... 3.91082644 3.78357959 4.20565224]]
```

Figure 4.1.3-5: a quick view for the prediction (range from 0 to 5)

Using this matrix, we can do the recommender system part. The more detail of the model can be seen in the Colab link:

https://colab.research.google.com/drive/1l5EP5dMe0XPEQ1H85iuIQE8cQus0O1qm

4.1.4 Testing

In order to get higher accuracy, we do the parameter tuning part. First, we change the activation function in tensorflow package to estimate the performance. We tried relu, softplus and tanh and compare the results with the sigmoid algorithm in the original code. The RMSE evaluator testing by validation set reduces at the first several epochs, however, it increases rapidly in afterwards epochs.

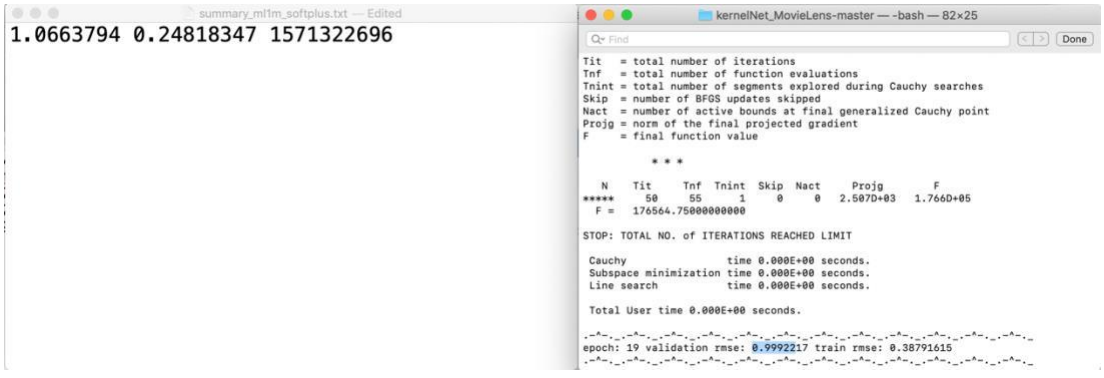


Figure 4.1.4-1: RMSE value and running time(activator: softplus)

Figure 4.1.4-2: RMSE value and running time(activator: tanh)

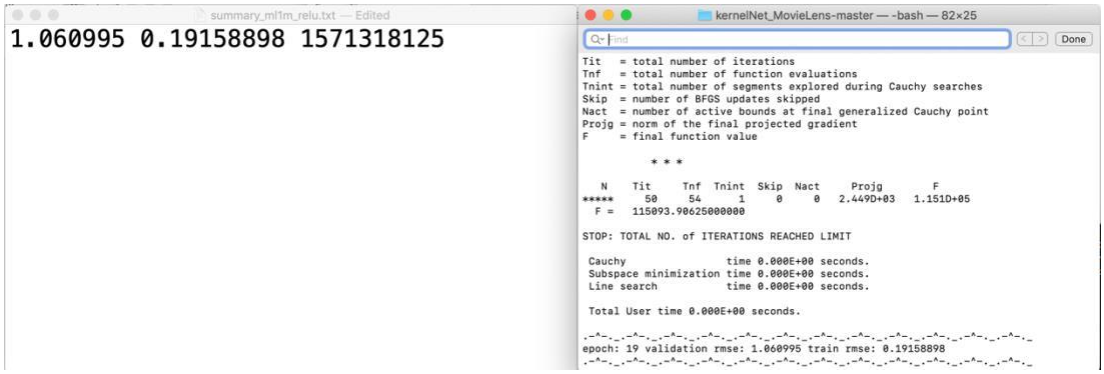


Figure 4.1.4-3: RMSE value and running time (activator: relu)

Due to the overfitting problem, we try different activators to figure out the lowest RMSE value. For tanh evaluator, when the model iterates in the 14 epochs, it already reaches the lowest RMSE value 0.8283386. For softplus evaluator, on epoch 2 it reaches its the lowest RMSE value 0.88014776. For relu evaluator, the lowest RMSE value is at the very beginning and equals to 0.8702067. Here are the detailed screenshots. For the initial code, it uses sigmoid activator and the final RMSE value is 0.8271394 by training 20 epochs. When considering the time we spent to run the model, the tanh activator is better than sigmoid activator.

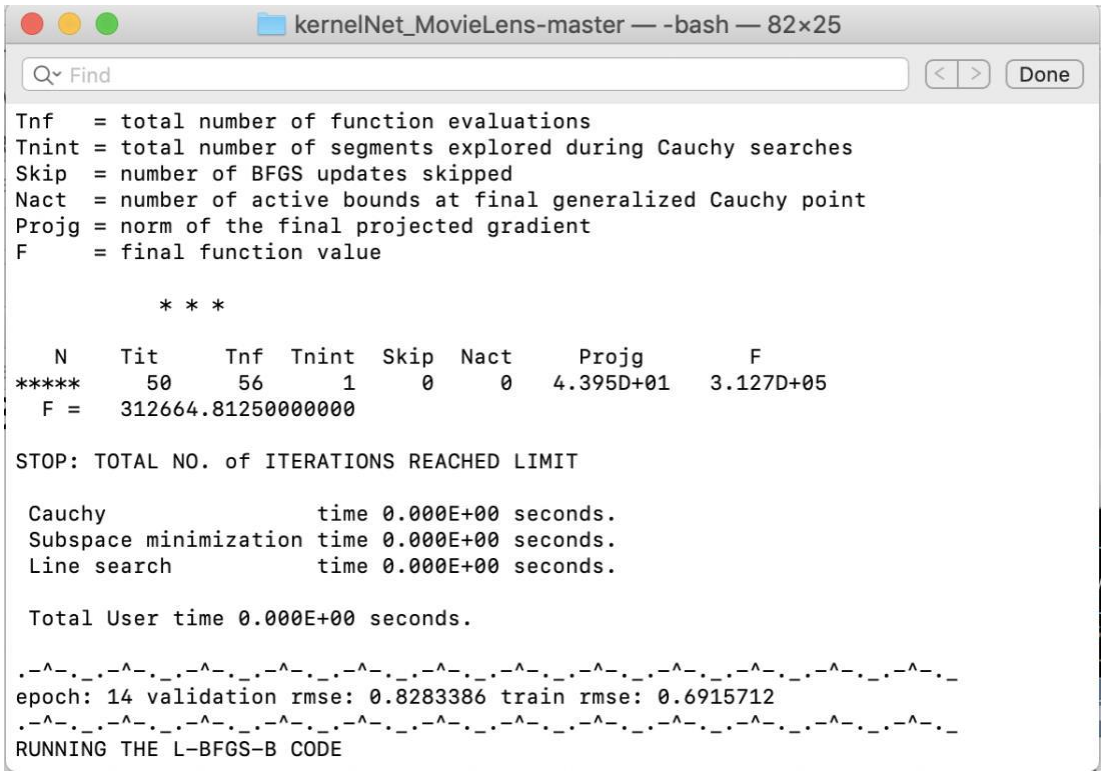


Figure 4.1.4-4: lowest RMSE value using tanh activator

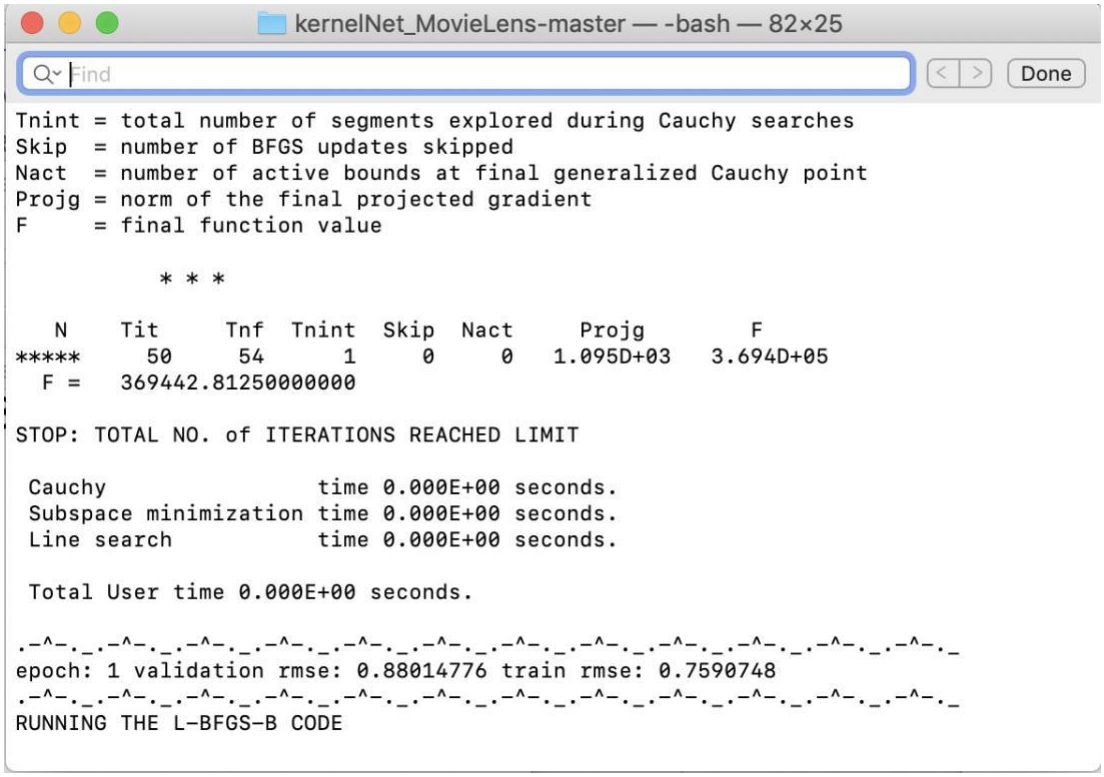


Figure 4.1.4-5: lowest RMSE value using softplus activator

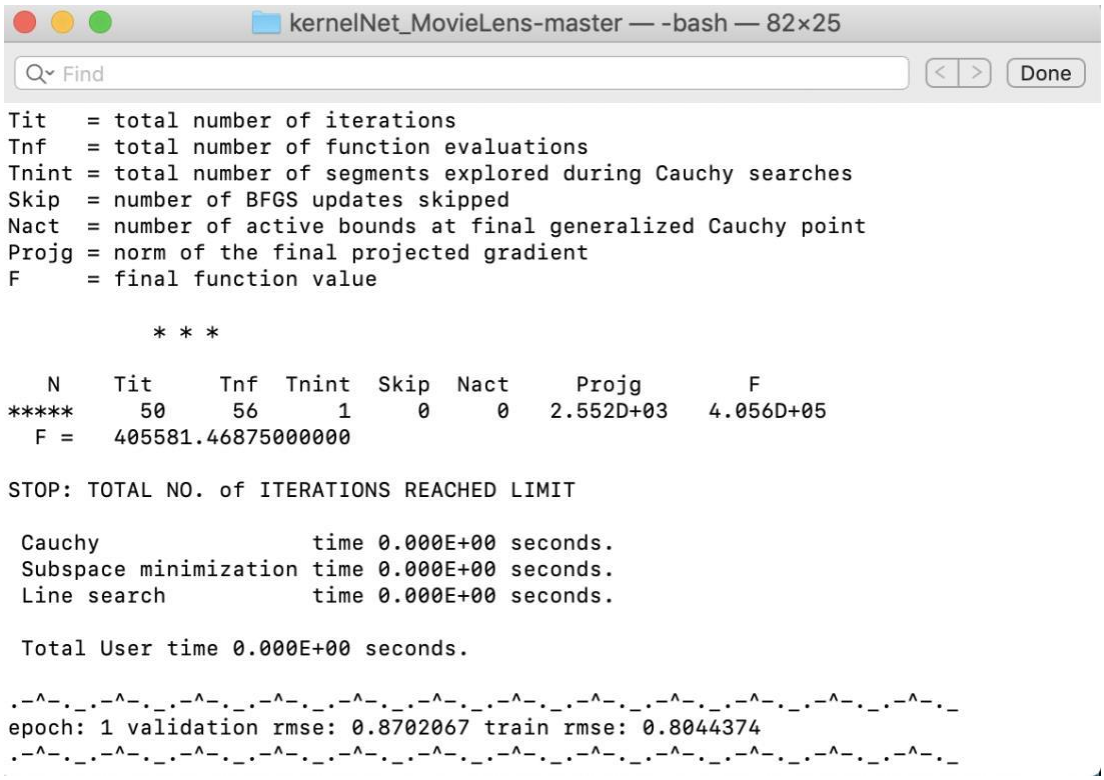


Figure 4.1.4-6: lowest RMSE value using relu activator

Then we changed the number of kernel layers but the efficiency seems to turn to very poor when changing the number for 2 to 3. Here are the screenshots for detailed information.

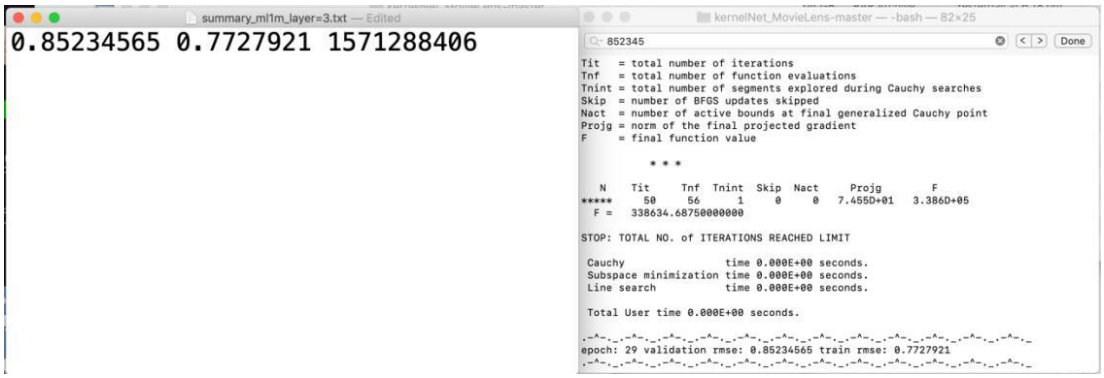


Figure 4.1.4-7: The final result when the number of kernel layer is equal to 3

Besides these two, we set the parameter *n_hid* in the range from 450 to 550 at the 50 steps. In the original code, the initial value equals to 500. We tried 450 and 550 to test its performance. Here are the screenshots. From the output, we can figure out that using n_hid = 550 has a much better performance than using n_hid = 500. Its

RMSE value equals to 0.8232917 which is lower than the RMSE value in the original code.

Figure 4.1.4-8: The final result when n_hid is equal to 450

Figure 4.1.4-9: The final result when n_hid is equal to 550

4.2 Graph Convolutional Matrix Completion

4.2.1 Methods

The Graph Convolutional Matrix Completion model implements collaborative filtering by transforming the problem of matrix completion to the link prediction on the graph based on the graph auto-encoder framework which models graph-structured data with end-to-end learning techniques (Berg, Kipf & Welling, 2017).

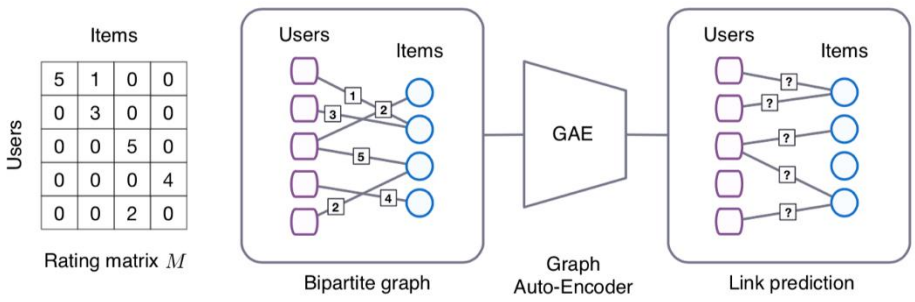


Figure 4.2.1-1: The process of GC-MC

As the figure above shown, the rating matrix M represents a shape of a number of users * the number of items. The entries in the matrix encode the user-item interaction where can be either observed rating(between 1-5) or unobserved(encode by 0). It is converted to the structure of bipartite graph and then passing message to the graph auto-encoder to model and predict links with bipartite structure which shows the interaction events between user and item. The table below shows the relationship between matrix completion and link prediction.

Matrix completion	**Link prediction on graph**
Interaction data	Bipartite graph (between user and item nodes)
Observed ratings/purchases	Links
Content information	Node features
Predict ratings	Predict labelled links

Table 4.2.1-1: The relationship between matrix completion and link prediction

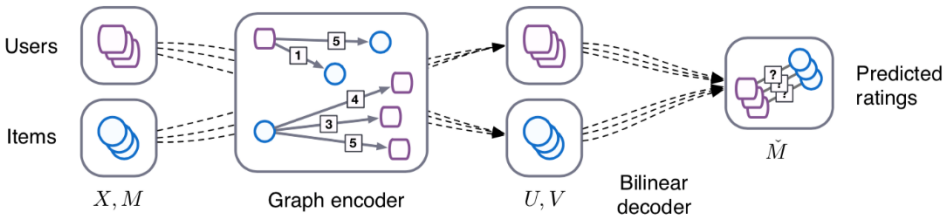


Figure 4.2.1-2: Schematic of the GC-MC model

The figure above shows the specific schematic of the graph auto-encoders which has two parts. One is the graph encoder which passes and transforms the message to produce latent features between user and item nodes based on the bipartite structure. The other is bilinear decoder which reconstructs the links by using the latent features to predict ratings.

The graph encoder refers to graph convolutional encoder which utilizes weight sharing for each location in the graph effectively and provides separate processing channels for each rating level (Berg, Kipf & Welling, 2017). It performs local operation across all location in the graph bypassing and transforming vector-valued messages of the following form:

$$\mu_{j\to i,r} = \frac{1}{c_{ij}} W_r x_j \,. \tag{1}$$

Then, it comes to the graph convolutional layer where the incoming messages are accumulated at every node into a single vector like:

$$h_i = \sigma\left[\text{accum}\left(\sum_{j\in\mathcal{N}_{i,1}} \mu_{j\to i,1}, \cdots, \sum_{j\in\mathcal{N}_{i,R}} \mu_{j\to i,R}\right)\right], \tag{2}$$

Where accum() is an accumulation operation. In this model, there are two different types of operation, which stack is a concatenation of vectors, the sum is the summation of all messages. It can be seen in the code:

For stack operation

```
[ ]  class StackGCN(Layer):
         """Graph convolution layer for bipartite graphs and sparse inputs."""

         def __init__(self, input_dim, output_dim, support, support_t, num_support, u_features_nonzero=None,
                      v_features_nonzero=None, sparse_inputs=False, dropout=0.,
                      act=tf.nn.relu, share_user_item_weights=True, **kwargs):
             super(StackGCN, self).__init__(**kwargs)

             assert output_dim % num_support == 0, 'output_dim must be multiple of num_support for stackGC layer'

             with tf.variable_scope(self.name + '_vars'):
                 self.vars['weights_u'] = weight_variable_random_uniform(input_dim, output_dim, name='weights_u')

                 if not share_user_item_weights:
                     self.vars['weights_v'] = weight_variable_random_uniform(input_dim, output_dim, name='weights_v')

                 else:
                     self.vars['weights_v'] = self.vars['weights_u']

             self.weights_u = tf.split(value=self.vars['weights_u'], axis=1, num_or_size_splits=num_support)
             self.weights_v = tf.split(value=self.vars['weights_v'], axis=1, num_or_size_splits=num_support)

             self.dropout = dropout

             self.sparse_inputs = sparse_inputs
             self.u_features_nonzero = u_features_nonzero
             self.v_features_nonzero = v_features_nonzero
             if sparse_inputs:
                 assert u_features_nonzero is not None and v_features_nonzero is not None, \
                     'u_features_nonzero and v_features_nonzero can not be None when sparse_inputs is True'

             self.support = tf.sparse_split(axis=1, num_split=num_support, sp_input=support)
             self.support_transpose = tf.sparse_split(axis=1, num_split=num_support, sp_input=support_t)

             self.act = act

             if self.logging:
```

Figure 4.2.1-3: Graph convolutional layer with stack operation

For sum operation

```
[ ]  class OrdinalMixtureGCN(Layer):

        """Graph convolution layer for bipartite graphs and sparse inputs."""

        def __init__(self, input_dim, output_dim, support, support_t, num_support, u_features_nonzero=None,
                     v_features_nonzero=None, sparse_inputs=False, dropout=0.,
                     act=tf.nn.relu, bias=False, share_user_item_weights=False, self_connections=False, **kwargs):
            super(OrdinalMixtureGCN, self).__init__(**kwargs)

            with tf.variable_scope(self.name + '_vars'):

                self.vars['weights_u'] = tf.stack([weight_variable_random_uniform(input_dim, output_dim,
                                                                                  name='weights_u_%d' % i)
                                                   for i in range(num_support)], axis=0)

                if bias:
                    self.vars['bias_u'] = bias_variable_const([output_dim], 0.01, name="bias_u")

                if not share_user_item_weights:
                    self.vars['weights_v'] = tf.stack([weight_variable_random_uniform(input_dim, output_dim,
                                                                                      name='weights_v_%d' % i)
                                                       for i in range(num_support)], axis=0)

                    if bias:
                        self.vars['bias_v'] = bias_variable_const([output_dim], 0.01, name="bias_v")

                else:
                    self.vars['weights_v'] = self.vars['weights_u']
                    if bias:
                        self.vars['bias_v'] = self.vars['bias_u']

            self.weights_u = self.vars['weights_u']
            self.weights_v = self.vars['weights_v']

            self.dropout = dropout

            self.sparse_inputs = sparse_inputs
```

Figure 4.2.1-4: Graph convolutional layer with sum operation

After that, a dense layer is followed which aims to get the final node embedding by transforming the output of last step like:

$$u_i = \sigma(W h_i)\,. \tag{3}$$

The corresponding code is like:

Dense layer

Class Name: Dense

Constructor Input:

dimention of user features (input_dim)

the number hidden units in 1st layer (output_dim)

dropout fraction (dropout)

activation function (act)

weight variable (share_user_item_weights)

Returns: matrix after activation function

Used in: def _build(self) in model.py

```
[ ]  class Dense(Layer):
        """Dense layer for two types of nodes in a bipartite graph. """

        def __init__(self, input_dim, output_dim, dropout=0., act=tf.nn.relu, share_user_item_weights=False,
                     bias=False, **kwargs):
            super(Dense, self).__init__(**kwargs)

            with tf.variable_scope(self.name + '_vars'):
                if not share_user_item_weights:

                    self.vars['weights_u'] = weight_variable_random_uniform(input_dim, output_dim, name="weights_u")
                    self.vars['weights_v'] = weight_variable_random_uniform(input_dim, output_dim, name="weights_v")

                    if bias:
                        self.vars['user_bias'] = bias_variable_truncated_normal([output_dim], name="bias_u")
                        self.vars['item_bias'] = bias_variable_truncated_normal([output_dim], name="bias_v")
```

Figure 4.2.1-5: Dense layer

For the bilinear decoder, it produces a probability distribution of possible rating levels by a bilinear decoder and uses a SoftMax function on the results like:

$$p(\check{M}_{ij} = r) = \frac{e^{u_i^T Q_r v_j}}{\sum_{s \in R} e^{u_i^T Q_s v_j}}, \tag{4}$$

Then the predicted rating is calculated as probability distribution dot product with rating levels:

$$\check{M}_{ij} = g(u_i, v_j) = \mathbb{E}_{p(\check{M}_{ij}=r)}[r] = \sum_{r \in R} r \; p(\check{M}_{ij} = r) \,. \tag{5}$$

This part is represented as a bilinear layer in the code as:

Bilinear decoder

Bilinear layer

```
[ ]  class BilinearMixture(Layer):
         """
         Decoder model layer for link-prediction with ratings
         To use in combination with bipartite layers.
         """

         def __init__(self, num_classes, u_indices, v_indices, input_dim, num_users, num_items, user_item_bias=False,
                      dropout=0., act=tf.nn.softmax, num_weights=3,
                      diagonal=True, **kwargs):
             super(BilinearMixture, self).__init__(**kwargs)
             with tf.variable_scope(self.name + '_vars'):

                 for i in range(num_weights):
                     if diagonal:
                         #  Diagonal weight matrices for each class stored as vectors
                         self.vars['weights_%d' % i] = weight_variable_random_uniform(1, input_dim, name='weights_%d' % i)

                     else:
                         self.vars['weights_%d' % i] = orthogonal([input_dim, input_dim], name='weights_%d' % i)

                 self.vars['weights_scalars'] = weight_variable_random_uniform(num_weights, num_classes,
                                                                               name='weights_u_scalars')

                 if user_item_bias:
                     self.vars['user_bias'] = bias_variable_zero([num_users, num_classes], name='user_bias')
                     self.vars['item_bias'] = bias_variable_zero([num_items, num_classes], name='item_bias')

             self.user_item_bias = user_item_bias

             if diagonal:
                 self._multiply_inputs_weights = tf.multiply
             else:
```

Figure 4.2.1-6: Bilinear layer

At this point, the model is built completely. There are two models can be used which class RecommenderSideInfoGAE() is for the dataset with the side information, and class RecommenderGAE() is for dataset without the side information. The difference between them is that there is one more dense layer of build() function in the class RecommenderSideInfoGAE(). It can be seen in the code as:

```
def _build(self):
    if self.accum == 'sum':
        self.layers.append(OrdinalMixtureGCN(input_dim=self.input_dim,
                                             output_dim=self.hidden[0],
                                             support=self.support,
                                             support_t=self.support_t,
                                             num_support=self.num_support,
                                             u_features_nonzero=self.u_features_nonzero,
                                             v_features_nonzero=self.v_features_nonzero,
                                             sparse_inputs=True,
                                             act=tf.nn.relu,
                                             bias=False,
                                             dropout=self.dropout,
                                             logging=self.logging,
                                             share_user_item_weights=True,
                                             self_connections=False))

    elif self.accum == 'stack':
        self.layers.append(StackGCN(input_dim=self.input_dim,
                                    output_dim=self.hidden[0],
                                    support=self.support,
                                    support_t=self.support_t,
                                    num_support=self.num_support,
                                    u_features_nonzero=self.u_features_nonzero,
                                    v_features_nonzero=self.v_features_nonzero,
                                    sparse_inputs=True,
                                    act=tf.nn.relu,
                                    dropout=self.dropout,
                                    logging=self.logging,
                                    share_user_item_weights=True))
    else:
        raise ValueError('accumulation function option invalid, can only be stack or sum.')

    self.layers.append(Dense(input_dim=self.hidden[0],
                             output_dim=self.hidden[1],
                             act=lambda x: x,
                             dropout=self.dropout,
                             logging=self.logging,
                             share_user_item_weights=True))

    self.layers.append(BilinearMixture(num_classes=self.num_classes,
                                       u_indices=self.u_indices,
                                       v_indices=self.v_indices,
                                       input_dim=self.hidden[1],
                                       num_users=self.num_users,
                                       num_items=self.num_items,
                                       user_item_bias=False,
                                       dropout=0.,
                                       act=lambda x: x,
                                       num_weights=self.num_basis_functions,
                                       logging=self.logging,
                                       diagonal=False))
```

Figure 4.2.1-7: build()in RecommenderGAE();

```
def _build(self):
    if self.accum == 'sum':
        self.layers.append(OrdinalMixtureGCN(input_dim=self.input_dim,
                                             output_dim=self.hidden[0],
                                             support=self.support,
                                             support_t=self.support_t,
                                             num_support=self.num_support,
                                             u_features_nonzero=self.u_features_nonzero,
                                             v_features_nonzero=self.v_features_nonzero,
                                             sparse_inputs=True,
                                             act=tf.nn.relu,
                                             bias=False,
                                             dropout=self.dropout,
                                             logging=self.logging,
                                             share_user_item_weights=True,
                                             self_connections=self.self_connections))

    elif self.accum == 'stack':
        self.layers.append(StackGCN(input_dim=self.input_dim,
                                    output_dim=self.hidden[0],
                                    support=self.support,
                                    support_t=self.support_t,
                                    num_support=self.num_support,
                                    u_features_nonzero=self.u_features_nonzero,
                                    v_features_nonzero=self.v_features_nonzero,
                                    sparse_inputs=True,
                                    act=tf.nn.relu,
                                    dropout=self.dropout,
                                    logging=self.logging,
                                    share_user_item_weights=True))

    else:
        raise ValueError('accumulation function option invalid, can only be stack or sum.')

    self.layers.append(Dense(input_dim=self.num_side_features,
                             output_dim=self.feat_hidden_dim,
                             act=tf.nn.relu,
                             dropout=0.,
                             logging=self.logging,
                             bias=True,
                             share_user_item_weights=False))

    self.layers.append(Dense(input_dim=self.hidden[0]+self.feat_hidden_dim,
                             output_dim=self.hidden[1],
                             act=lambda x: x,
                             dropout=self.dropout,
                             logging=self.logging,
                             share_user_item_weights=False))

    self.layers.append(BilinearMixture(num_classes=self.num_classes,
                                       u_indices=self.u_indices,
                                       v_indices=self.v_indices,
```

Figure 4.2.1-7: build() in RecommenderSideInfoGAE ()

The more detail of the model can be seen in the Colab link:

https://colab.research.google.com/drive/1WY6c6isb4ahwz11lnCniFDwmUL7KRafu

4.2.2 Data Collection

GC-MC model evaluates 6 datasets which are Flixster, Douban, YahooMusic, and MovieLens(100K, 1M, and 10M). They have user-item ratings with different sizes and additional users/items information.

The table below shows the detail information of each dataset.

Dataset	Users	Items	Features	Ratings	Density	Rating levels
Flixster	3,000	3,000	Users/Items	26,173	0.0029	0.5, 1, . . . , 5
Douban	3,000	3,000	Users	136,891	0.0152	1, 2, . . . , 5
YahooMusic	3,000	3,000	Items	5,335	0.0006	1, 2, . . . , 100
MovieLens 100K (ML-100K)	943	1,682	Users/Items	100,000	0.0630	1, 2, . . . , 5
MovieLens 1M (ML-1M)	6,040	3,706	—	1,000,209	0.0447	1, 2, . . . , 5
MovieLens 10M (ML-10M)	69,878	10,677	—	10,000,054	0.0134	0.5, 1, . . . , 5

Table 4.2.2-1: Datasets used in GC-MC model

For Flixser, Douban, and YahooMusic, they are all MatLab files named 'training_test_dataset.mat'. Each dataset contains 3000 users, 3000 items, their corresponding rating matrix, and optionally users/items side information in the form of features. In the code, the load_matlab_file() function is to load the dataset directly by the following:

▾ Load matlab file

Overview: load '.mat' files

Constructor Input:

path_file: string containing the file path

name_field: string containig the field name (default='shape')

Warning: '.mat' files should be saved in the '-v7.3' format

```
[ ] def load_matlab_file(path_file, name_field):

        db = h5py.File(path_file, 'r')
        ds = db[name_field]
        try:
            if 'ir' in ds.keys():
                data = np.asarray(ds['data'])
                ir = np.asarray(ds['ir'])
                jc = np.asarray(ds['jc'])
                out = sp.csc_matrix((data, ir, jc)).astype(np.float32)
        except AttributeError:
            # Transpose in case is a dense matrix because of the row- vs column- major ordering between python and matl
            out = np.asarray(ds).astype(np.float32).T

        db.close()

        return out
```

Figure 4.2.2-1: load_matlab_file() function

For MovieLens(100K, 1M, and 10M), they are zip compressed files. The dataset is downloaded firstly by the function of download_dataset() which is shown below:

▾ Download dataset

Overview: Downloads dataset if files are not present.

Used in: preprocessing.py

```
[ ] def download_dataset(dataset, files, data_dir):

        if not np.all([os.path.isfile(data_dir + f) for f in files]):
            url = "http://files.grouplens.org/datasets/movielens/" + dataset.replace('_', '-') + '.zip'
            request = urlopen(url)

            print('Downloading %s dataset' % dataset)
            if dataset in ['ml_100k', 'ml_1m']:
                target_dir = 'data/' + dataset.replace('_', '-')
            elif dataset == 'ml_10m':
                target_dir = 'data/' + 'ml-10M100K'
            else:
                raise ValueError('Invalid dataset option %s' % dataset)

            with ZipFile(BytesIO(request.read())) as zip_ref:
                zip_ref.extractall('data/')

            source = [target_dir + '/' + s for s in os.listdir(target_dir)]
            destination = data_dir+'/'
            for f in source:
                shutil.copy(f, destination)

            shutil.rmtree(target_dir)
```

Figure 4.2.2-2: download_dataset() function

Then applying load_data() function to load the dataset and create adjacency matrix and feature matrix. In addition, the shuffling of the dataset is done in this function.

▾ Load Data

Overview: Loads dataset and creates adjacency matrix and feature matrix

Constructor Input:
fname : str, dataset

seed: int, dataset shuffling seed

verbose: to print out statements or not

Returns:

num_users (int): Number of users and items respectively

num_items(int)

u_nodes (np.int32 arrays): User indices

v_nodes (np.int32 array): item (movie) indices

ratings (np.float32 array): User/item ratings s.t. ratings[k] is the rating given by user u_nodes[k] to item v_nodes[k]. Note that that the all pairs u_nodes[k]/v_nodes[k] are unique, but not necessarily all u_nodes[k] or all v_nodes[k] separately.

u_features (np.float32 array, or None): If present in dataset, contains the features of the users.

v_features (np.float32 array, or None): If present in dataset, contains the features of the users.

seed (int): For datashuffling seed with pythons own random.shuffle, as in CF-NADE.

Used in: preprocessing.py

```
[ ]  def load_data(fname, seed=1234, verbose=True):

         u_features = None
         v_features = None

         print('Loading dataset', fname)
```

Figure 4.2.2-3: load_data() function

4.2.3 Data Analysis

In this model, the dataset is split into train, validation, and test sets from the full bipartite adjacency matrix.

For Flixser, Douban, and YahooMusic, the process is completed in load_data_monti() function. It is worth mentioning that internally shuffle happens in training set and using 20% of the training set for validation.

▾ Load data monti

Overview: Loads data from Monti et al. paper

```
[ ] def load_data_monti(dataset, testing=False):
        """
        Loads data from Monti et al. paper.
        """

        path_dataset = 'data/' + dataset + '/training_test_dataset.mat'

        M = load_matlab_file(path_dataset, 'M')
        Otraining = load_matlab_file(path_dataset, 'Otraining')
        Otest = load_matlab_file(path_dataset, 'Otest')

        num_users = M.shape[0]
        num_items = M.shape[1]

        if dataset == 'flixster':
            Wrow = load_matlab_file(path_dataset, 'W_users')
            Wcol = load_matlab_file(path_dataset, 'W_movies')
            u_features = Wrow
            v_features = Wcol
            # print(num_items, v_features.shape)
            # v_features = np.eye(num_items)

        elif dataset == 'douban':
            Wrow = load_matlab_file(path_dataset, 'W_users')
            u_features = Wrow
            v_features = np.eye(num_items)
        elif dataset == 'yahoo_music':
            Wcol = load_matlab_file(path_dataset, 'W_tracks')
            u_features = np.eye(num_users)
            v_features = Wcol

        u_nodes_ratings = np.where(M)[0]
```

Figure 4.2.3-1: load_data_monti() function

For MovieLens datasets, load_official_trainvaltest_split() function is used to ML 100K, create_trainvaltest_split() function is used to ML 1M and ML 10M. Both of these functions compute 1-of-num_classes labels for each split. Also compute training adjacency matrix.

Load official trainvaltest split

Objective: Loads official train/test split and uses 10% of training samples for validaiton. For each split computes 1-of-num_classes labels. Also computes training adjacency matrix. Assumes flattening happens everywhere in row-major fashion.

```
[ ] def load_official_trainvaltest_split(dataset, testing=False):

        sep = '\t'

        # Check if files exist and download otherwise
        files = ['/u1.base', '/u1.test', '/u.item', '/u.user']
        fname = dataset
        data_dir = 'data/' + fname

        download_dataset(fname, files, data_dir)

        dtypes = {
            'u_nodes': np.int32, 'v_nodes': np.int32,
            'ratings': np.float32, 'timestamp': np.float64}

        filename_train = 'data/' + dataset + '/u1.base'
        filename_test = 'data/' + dataset + '/u1.test'

        data_train = pd.read_csv(
            filename_train, sep=sep, header=None,
            names=['u_nodes', 'v_nodes', 'ratings', 'timestamp'], dtype=dtypes)

        data_test = pd.read_csv(
            filename_test, sep=sep, header=None,
            names=['u_nodes', 'v_nodes', 'ratings', 'timestamp'], dtype=dtypes)

        data_array_train = data_train.as_matrix().tolist()
        data_array_train = np.array(data_array_train)
        data_array_test = data_test.as_matrix().tolist()
        data_array_test = np.array(data_array_test)
```

Figure 4.2.3-2: load_official_trainvaltest_split() function

Create trainvaltest split

Overview: Splits data set into train/val/test sets from full bipartite adjacency matrix. Shuffling of dataset is done in load_data function. For each split computes 1-of-num_classes labels. Also computes training adjacency matrix.

```
[ ] def create_trainvaltest_split(dataset, seed=1234, testing=False, datasplit_path=None, datasplit_from_file=False,
                                  verbose=True):

        if datasplit_from_file and os.path.isfile(datasplit_path):
            print('Reading dataset splits from file...')
            with open(datasplit_path, 'rb') as f:
                num_users, num_items, u_nodes, v_nodes, ratings, u_features, v_features = pkl.load(f)

            if verbose:
                print('Number of users = %d' % num_users)
                print('Number of items = %d' % num_items)
                print('Number of links = %d' % ratings.shape[0])
                print('Fraction of positive links = %.4f' % (float(ratings.shape[0]) / (num_users * num_items),))

        else:
            num_users, num_items, u_nodes, v_nodes, ratings, u_features, v_features = load_data(dataset, seed=seed,
                                                                                                verbose=verbose)

            with open(datasplit_path, 'wb') as f:
                pkl.dump([num_users, num_items, u_nodes, v_nodes, ratings, u_features, v_features], f)

        neutral_rating = -1

        rating_dict = {r: i for i, r in enumerate(np.sort(np.unique(ratings)).tolist())}

        labels = np.full((num_users, num_items), neutral_rating, dtype=np.int32)
        labels[u_nodes, v_nodes] = np.array([rating_dict[r] for r in ratings])
        labels = labels.reshape([-1])

        # number of test and validation edges
        num_test = int(np.ceil(ratings.shape[0] * 0.1))
```

Figure 4.2.3-3: create_trainvaltest_split() function

4.2.4 Deployment

Running on localhost

The primary method of running code on a local host is to run it in a terminal. In the current premise, there must be a Python environment and the necessary Python packages on the localhost.

Running code in a terminal is simple and convenient. First, go to the root directory of the GC-MC code and execute the python setup.py install command. The purpose of this step is Installation. After the installation, you need to execute cd gcmc to enter the gcmc subdirectory first, and then execute the relevant training or test commands. The specific running command is shown in the figure below.

```
#!/bin/bash

# Douban
python train.py -d douban --accum stack -do 0.7 -nleft -nb 2 -e 200 --features  --feat_hidden 64 --testing > douban_testing.txt  2>&1

# Flixster
python train.py -d flixster --accum stack -do 0.7 -nleft -nb 2 -e 200 --features --feat_hidden 64 --testing > flixster_testing.txt  2>&1

# Yahoo Music
python train.py -d yahoo_music --accum stack -do 0.7 -nleft -nb 2 -e 200 --features --feat_hidden 64 --testing > yahoo_music_testing.txt  2>&1

# Movielens 100K on official split with features
python train.py -d ml_100k --accum stack -do 0.7 -nleft -nb 2 -e 1000 --features --feat_hidden 10 --testing > ml_100k_feat_testing.txt  2>&1

# Movielens 100K on official split without features
python train.py -d ml_100k --accum stack -do 0.7 -nleft -nb 2 -e 1000 --testing  > ml_1m_testing.txt  > ml_100k_testing.txt  2>&1

# Movielens 1M
python train.py -d ml_1m --data_seed 1234 --accum sum -do 0.7 -nsym -nb 2 -e 3500 --testing > ml_1m_testing.txt 2>&1

# Movielens 10M
python train_mini_batch.py -d ml_10m --data_seed 1234 --accum stack -do 0.3 -nsym -nb 4 -e 20 --testing > ml_10m_testing.txt 2>&1
```

Figure 4.2.4-1: Running commands

Running on Google Colab

Except running on localhost successfully, we also want to run the code on Google Colab which is a jupyter notebook environment provided by Google in the cloud. It provides for others to view and run the code conveniently. The prerequisites to run codes on Colab is connected with Google Drive. It can be implemented by PyDrive which accesses Google Drive by the way of program. The user should have a Google Drive account and a verification code is needed to enter after running the code. After the connection is successful, download the code and set working directory. The code is following:

▾ Download the code

```
[ ] ! pip install PyDrive
```

```
[ ] from pydrive.auth import GoogleAuth
    from pydrive.drive import GoogleDrive
    from google.colab import auth
    from oauth2client.client import GoogleCredentials
    # Authenticate and create the PyDrive client.
    # This only needs to be done once per notebook.
    auth.authenticate_user()
    gauth = GoogleAuth()
    gauth.credentials = GoogleCredentials.get_application_default()
    drive = GoogleDrive(gauth)
```

```
[ ] # Downloading code from google drive
    file_id = '1pblN561PBvsts2UVF83yruVQYeK4s_qT' # URL id.
    downloaded = drive.CreateFile({'id': file_id})
    downloaded.GetContentFile('gc-mc-master.zip')
```

```
[ ] ! unzip '/content/gc-mc-master.zip'
```

▾ Setting working directory

```
[ ] import os
    os.chdir("/content/gc-mc-master")
```

Figure 4.2.4-2: Code of prerequisites

It also contains how to run the whole code by different data set in Prerequisites.

```
[ ] import os
    os.chdir("/content/gc-mc-master")
```

```
[ ] ! python setup.py install
```

```
[ ] os.chdir("/content/gc-mc-master/gcmc")
```

```
[ ] # Douban
    ! python train.py -d douban --accum stack -do 0.7 -nleft -nb 2 -e 200 --features --feat_hidden 64 --testing
```

```
[ ] # Flixster
    ! python train.py -d flixster --accum stack -do 0.7 -nleft -nb 2 -e 200 --features --feat_hidden 64 --testing
```

```
[ ] # Yahoo Music
    ! python train.py -d yahoo_music --accum stack -do 0.7 -nleft -nb 2 -e 200 --features --feat_hidden 64 --testing
```

```
[ ] # Movielens 100K on official split with features
    ! python train.py -d ml_100k --accum stack -do 0.7 -nleft -nb 2 -e 1000 --features --feat_hidden 10 --testing
```

```
[ ] # Movielens 100K on official split without features
    ! python train.py -d ml_100k --accum stack -do 0.7 -nleft -nb 2 -e 1000 --testing
```

```
[ ] # Movielens 1M
    ! python train.py -d ml_1m --data_seed 1234 --accum sum -do 0.7 -nsym -nb 2 -e 3500 --testing
```

```
[ ] # Movielens 10M
    ! python train_mini_batch.py -d ml_10m --data_seed 1234 --accum stack -do 0.3 -nsym -nb 4 -e 20 --testing
```

Figure 4.2.4-3: Running command

Update Python version

Because Python 2 is only maintained until 2020 and Python 3 is more and more popular, we want to update our code from Python 2 to Python 3 to avoid others cannot run successfully. There are some different usage of the package and function between different version, we modify some part of the code to make it suitable for the new version. The figure below shows the code where modifying.

Python 2	Python 3
import cPickle as pkl	import pickle as pkl
from urllib2 import urlopen	from urllib.request import urlopen
from StringIO import StringIO	import io
data = np.array(map(lambda x: id_dict[x], data))	data = np.array(list(map(lambda x: id_dict[x], data)))
with open(datasplit_path) as f:	with open(datasplit_path, 'rb') as f:
with open(datasplit_path, 'w') as f:	with open(datasplit_path, 'wb') as f:
train_u_indices_batch, train_v_indices_batch, train_labels_batch = data_iter.next()	train_u_indices_batch, train_v_indices_batch, train_labels_batch = data_iter.__next__()

Table 4.2.4-1: differences between Python 2 and Python 3

The complete code can be seen from the link below:

For Python 2,

https://drive.google.com/drive/folders/1qms-9V8plqPlPYToGAmuMoudWEaAUd68?usp=sharing

For Python 3,

https://drive.google.com/drive/folders/1CntdV8ATUsLmu1gkFWbWrIXlpSlTIwi2?usp=sharing

Restore the model

We also write a new Python file to restore the model. In this way, users can use this model to predict very quickly without training the model. In addition, it can be used in building an API in the future conveniently. Except for the stage of preprocessing the data, the main code which is used to restore the model is:

```
403  variables_to_restore = model.variable_averages.variables_to_restore()
404  saver = tf.train.Saver(variables_to_restore)
405  saver.restore(sess, "tmp/recommendersideinfogae.ckpt-200" )
406
407  outputs = sess.run(model.outputs, feed_dict=val_feed_dict)
408  print(outputs)
```

Figure 4.2.4-4: The code of restore the model

4.2.5 Testing

Unit Testing: After downloading the relevant code of GC-MC to the local, all the code of GC-MC consists of ten python files: __init__.py, data_utils.py, initializations.py, layers.py, metrics.py , model.py, preprocessing.py, train.py, train_mini_batch.py, utils.py. Each python file is one of the minimum components of the GC-MC.

Integration Testing: In order to ensure the integrity of the GC-MC code, after completing the unit test, some team members initially run the code on the localhost and correct the errors that occur during the operation, thus ensuring the availability of the GC-MC code.

System Test: After verifying the validity of the GC-MC code, the project team migrated it to Google Colab and integrated it into an ipynb file. The role of the system test is to ensure that the GC-MC can run smoothly in Colab after integration. In the process, team members can also verify the performance of the GC-MC by detecting the speed and accuracy of the code.

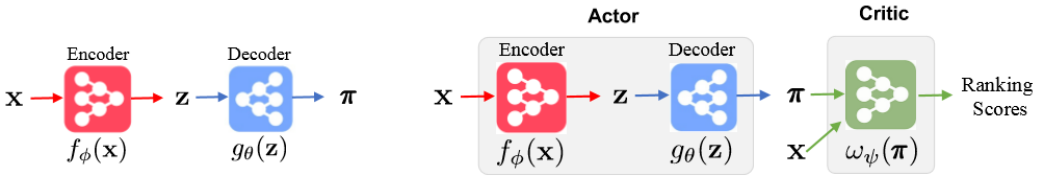
Performance Testing: Testing of the performance and performance of the GC-MC model. Here is mainly to test the performance of GC-MC under different data sets, and the impact of using different parameters and algorithms on performance, so as to ensure that the performance obtained is reasonable and valid.

Here we mainly test the performance of the model on six different and authoritative data sets. And adjust the parameters and method selection, including Accumulation function, Dropout, Normalization, Hidden units, etc. And RMSE is used as an indicator to evaluate the performance of the model.

	ML 100k	ML 1M	ML 10M	Flixster	Douban	Yahoo Music
Accumulation function	stack	sum	stack	Concatenation(as opposed to summation)		
dropout	0.7	0.7	0.3	0.7		
normalization	left	symmetric	symmetric	left		
hidden units	10	None	None	64		
RMSE	0.905	0.832	0.777	0.941/0.917	0.734	20.5

4.3 Towards Amortized Ranking-Critical Training for Collaborative Filtering

The recommendation system is an important way when it comes to improving the user's network experience. Collaborative filtering, which uses similar users and items patterns to predict user preferences, is a widely used technology in recommendation systems. Because it conforms to the learning paradigm of latent variable models (LVMs), its effectiveness and simplicity make it the dominant method of recommendation system. However, traditional LVMs use linear mapping with limited modelling capability, which may produce a suboptimal performance for large datasets. Therefore, the use of deep neural networks (DNN) is a better alternative. There are three deep learning models for collaborative filtering which are variational autoencoders (VAEs), collaborative denoising autoencoder (CDAE) and neural collaborative filtering (NCF). Among them, VAEs alleviates the model size grows linearly problem via amortized inference and integrates the flexible representation from deep neural network into the latent variable model, thus mitigating the limitations of traditional LVMs. Usually, VAEs are trained through maximizing the likelihood (MLE) of users interacting with ground-truth items, but this does not directly maximize recommended quality metrics that people care about, such as top-N projects' ranking. This paper presents a novel actor-critic algorithm for ranking-based training which calls Ranking-Critical Training (RaCT). Here is the schematic diagram:



(a) Traditional auto-encoder paradigm (b) Proposed actor-critic paradigm

Figure 4.3-1: Schematic of RaCT

Compared with the traditional learning-to-ranking method, which needs to rerun the optimization procedure for a new list, RaCT amortizes the scoring process through the neural network, and can directly provide the ranking score for the new lists.

Input : Interaction matrix $\mathbf{X}$; Actor parameters (encoder ϕ and decoder θ), Critic parameters ψ.

Initialize: Randomly initialize weights ϕ, θ and ψ
`/* Stage 1: Pretrain the actor via MLE`
while *not converged do* **do**
    Sample a batch of users $\mathcal{U}$;
    Update $\{\theta, \phi\}$ with gradient $\frac{\partial \mathcal{L}_\beta}{\partial \theta}$ and $\frac{\partial \mathcal{L}_\beta}{\partial \phi}$ in (4);
end
`/* Stage 2: Pretrain the critic via MSE`
while *not converged do* **do**
    Sample a batch of users $\mathcal{U}$;
    Construct features $\boldsymbol{h}$ in (5) and target y from the Oracle;
    Update ψ with gradient $\frac{\partial \mathcal{L}_C}{\partial \psi}$ in (6);
end
`/* Stage 3: Alternative training of actor and critic`
for $t = 1, 2, \ldots, T$ **do**
    Sample a batch of users $\mathcal{U}$;
    `/* Actor step`
    Update $\{\theta, \phi\}$ with gradient $\frac{\partial \mathcal{L}_A}{\partial \theta}$ and $\frac{\partial \mathcal{L}_A}{\partial \phi}$ in (7);
    `/* Critic step`
    Construct features $\boldsymbol{h}$ in (5) and target y from the Oracle;
    Update ψ with gradient $\frac{\partial \mathcal{L}_C}{\partial \psi}$ in (6);
end

Figure 4.3-2: Pseudo-code for RaCT

4.3.1 Methods

Based on paper *Towards Amortized Ranking-Critical Training for Collaborative Filtering*, authors put forward a new method to improve variational autoencoders (VAEs) which is Ranking Critical Training (RaCT). Since VAEs integrate the flexible representation from the deep neural network into the hidden variable model, the latest performance is obtained, which reduces the limitations of the traditional linear factor model. VAES is typically trained by maximizing the likelihood of users (MLE) interacting with real-world projects on the ground.

The vector is represented by the bold small letter X, the matrix by the bold capital letter X, and the scalar by the small non-bold letter X. We use ◦ as a function component, and ○ as a base for multiplication by element, | · | for cardinality of a set, and δ (·) is the indicator function.

We use n = {1,…, *N*}to index users, use M = {1,…, M} to index items The user project interaction matrix X = {0,1} $^{N \times M}$ collected from the user's implicit feedback is defined as:

$$x_{nm} \begin{cases} 1, & \text{if interaction of user } n \text{ with item } m \text{ is observed;} \\ 0, & \text{otherwise} \end{cases}$$

Note that $x_{nm} = 0$ does not necessarily mean that user n dislikes item m; it can be that the user is not aware of the item. Further, $x_{nm} /= 1$ is not equivalent to saying user n likes item m, but at least there is interest.

Given the user's interaction history $x = [x_1,\ldots, x_M]^T > \{0,1\}^M$, to predict the users' interaction behaviour with all remaining items. In order to simulate this process during training, a random binary mask $b = \{0,1\}^M$ is introduced. Set 1 as un-masked, and 0 as masked. Thus, $x_h = x \circ b$ is the user's partial interaction history. The goal becomes recovering the masked interactions: $x_p = x \circ (1 - b)$.

In LVMs, we assumed each user's binary interaction behaviour can be represented by a k-dimensional user-dependent latent representation $z = R^K$. When applying VAEs to collaborative filtering, the user's latent feature z is represented as a distribution $q(z|x)$, obtained from some partial history x_h of x. $q(z|x)$ follows a Gaussian form, the inference of z is performed as:

$$q_{\phi}(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2)), \quad \text{with} \quad \boldsymbol{\mu}, \boldsymbol{\sigma}^2 = f_{\phi}(\boldsymbol{x}_h), \quad \boldsymbol{x}_h = \boldsymbol{x} \odot \boldsymbol{b}, \quad \boldsymbol{b} \sim \text{Ber}(\alpha)$$

Where α is the hyper-parameter of a Bernoulli distribution, $f\varphi$ is a φ-parameterized neural network, which outputs the mean μ and variance σ^2 of the Gaussian distribution.

To train the VAE, its parameters $\{\boldsymbol{\varphi}, \boldsymbol{\theta}\}$ yields the following generalized objective:

$$\mathcal{L}_{\beta}(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{\phi}) = \mathcal{L}_E + \beta \mathcal{L}_R, \text{ with } \mathcal{L}_E = -\mathbb{E}_{q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})}\left[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}|\boldsymbol{z})\right] \text{ and } \mathcal{L}_R = \text{KL}(q_{\boldsymbol{\phi}}(\boldsymbol{z}|\boldsymbol{x})||p(\boldsymbol{z}))$$

Where L_E is the negative log-likelihood (NLL) term, L_R is the KL regularization term, and β is a weighting hyper-parameter.

While MLE-based training does not directly maximize the recommended quality metrics that people usually concern with, such as top-N ranking. Therefore, the authors add a critical process to measure the recommended items. Consider the computation procedure of the evaluation metric as a function decomposition $w = w_0 \circ w\varphi$, including two steps:

- $\omega_0 : \boldsymbol{\pi} \mapsto \boldsymbol{h}$, feature engineering of prediction $\boldsymbol{\pi}$ into the *sufficient statistics* $\boldsymbol{h}$;
- $\omega_{\psi} : \boldsymbol{h} \mapsto \hat{y}$, neural approximation of the mapping from the statistics $\boldsymbol{h}$ to the estimated ranking score $\hat{y}$, using a ψ-parameterized neural network;

Depends on the feature *h*, the critic process achieves big success. Feature *h* is defined as:

$$\boldsymbol{h} = [\mathcal{L}_E, |\mathcal{H}_0|, |\mathcal{H}_1|],$$

By adding the critic process, the new model, RaCT performs much better than VAEs.

4.3.2 Data Collection

In the experiment, we use 3 datasets, ML-20M, Netflix and MSD.

ML-20M is a dataset that contains 20 million ratings and 465,000 tag applications that applied by 27,000 movies by 138,000 users. It describes 5-star rating and free-text tagging activity from a movie recommendation service. Its format is described as below:

User Ids

MovieLens users are randomly selected. Their identity has been anonymous. User ids are consistent between `ratings.csv` and `tags.csv` (i.e., the same id refers to the same user across the two files).

Movie Ids

The data set only includes movies with at least one rating or tag. These movie Ids are consistent with those used on the MovieLens website.

ratings.csv contains all ratings of movies. Each line represents one rating of one movie by one user. tag.csv contains all tags that represent one tag applied to one movie by one user. movies.csv is a file that contains movie's information. link.csv is identifiers that can be used to link to other sources of movie data.

Netflix is a movie rating dataset that contains over 100 million ratings from 480 thousand randomly-chosen, anonymous Netflix customers over 17 thousand movie titles. The following is the training dataset format:

MovieIDs, it ranges from 1 to 17770. CustomerIDs, it ranges from 1 to 2649429. Ratings range from 1 to 5. Dates, the format is YYYY-MM-DD.

And this is movies file format:

MovieID. It does not correspond to actual Netflix movie ids or IMDB movie ids. YearOfRelease is the release year of the movie. It can range from 1890 to 2005 and may correspond to the release of corresponding DVD. Title is the Netflix movie title and may not correspond to titles used on other sites.

The last is MSD dataset. It is different from the other two datasets. It is a song dataset that contains millions of songs. It has 44, 745 unique artists, 7, 643 unique terms, 2, 321 unique MusicBrainz tags and 43, 943 artists with at least one term. So this large dataset is appropriate for the RaCT model.

4.3.3 Data Analysis

Downloaded the data from websites, first we unzip the data and extract the MovieID and UserID that ratings over 3.5 into a new file. And split the whole dataset into train dataset, test dataset and validation dataset. All these preprocesses are done by numpy and pandas package.

Second, to input the data to the model, we use sparse.csr_matrix function to make the user interactions history to be a sparse matrix. Get a k-dimensional user-dependent latent representation through the encoder. Then with the decoder, we generate the predictions for users.

Third, we use critic to measure the predictions, get the final top-N results. With this process, we can recommend items to users more accurately.

4.3.4 Deployment

To implement the RaCT model code. We deploy them on Google Colab. We break down useful code to each block and make comments on them.

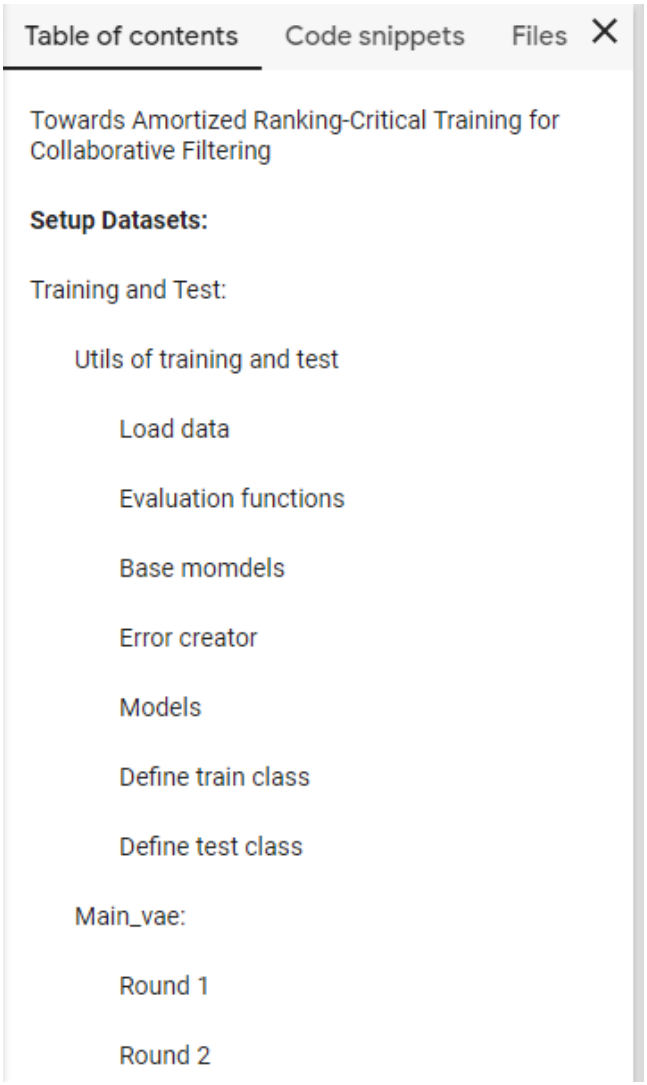


Figure 4.3.4-1 Contents of code

This is the contents of our code deployment. It mainly has two parts, setup datasets and training and testing. In the setup datasets part, we download three datasets from websites and preprocess them.

Setup Datasets:

It is used to download from websites, extract, and process the datasets.

It includes **ml-20m, netflix-prize and msd**. (We only download the ml-20m to reduce the datasets)

```
import urllib
import os
import shutil
import pandas as pd
import numpy as np
import sys
import zipfile
import distutils
import distutils.util


def save_zip_data(write_path, zip_url):
    # zip_url = "http://files.grouplens.org/datasets/movielens/ml-20m.zip"
    print('reading response')
    with urllib.request.urlopen(zip_url) as response:
        zip_file = response.read()
    print('writing zip file')
    with open(write_path, 'wb') as f:
        f.write(zip_file)

def maybe_download_and_extract_movie_data(data_dir, force_overwrite=False):
    write_path = os.path.join(data_dir, 'ml-20m.zip')
    zip_url = "http://files.grouplens.org/datasets/movielens/ml-20m.zip"
    if not os.path.isfile(write_path):
        os.makedirs(data_dir, exist_ok=True)
        print("Zip not downloaded. Downloading now...")
        save_zip_data(write_path, zip_url)
        print("Zip downloaded")
    else:
        print("Zip already downloaded")

    extract_destination = os.path.join(data_dir, "ml-20m")
    if os.path.isdir(extract_destination):
```

Figure 4.3.4-2 setup datasets(1)

In the screenshot, the code is for downloading the datasets and unzip them.

```python
def load_train_data(csv_file):
    tp = pd.read_csv(csv_file)
    n_users = tp['uid'].max() + 1

    rows, cols = tp['uid'], tp['sid']
    data = sparse.csr_matrix((np.ones_like(rows),
                              (rows, cols)), dtype='float64',
                              shape=(n_users, n_items))
    return data


def load_tr_te_data(csv_file_tr, csv_file_te):
    tp_tr = pd.read_csv(csv_file_tr)
    tp_te = pd.read_csv(csv_file_te)

    start_idx = min(tp_tr['uid'].min(), tp_te['uid'].min())
    end_idx = max(tp_tr['uid'].max(), tp_te['uid'].max())

    rows_tr, cols_tr = tp_tr['uid'] - start_idx, tp_tr['sid']
    rows_te, cols_te = tp_te['uid'] - start_idx, tp_te['sid']

    data_tr = sparse.csr_matrix((np.ones_like(rows_tr),
                              (rows_tr, cols_tr)), dtype='float64', shape=(end_idx - start_idx + 1, n_items))
    data_te = sparse.csr_matrix((np.ones_like(rows_te),
                              (rows_te, cols_te)), dtype='float64', shape=(end_idx - start_idx + 1, n_items))
    return data_tr, data_te


def process_unzipped_data(DATA_DIR,
                          force_overwrite=False,
                          n_heldout_users=10000,
                          discard_ratings_below=3.5,
                          min_users_per_item_to_include=0,
                          min_clicks_per_user_to_include=5):

    pro_dir = os.path.join(DATA_DIR, 'pro_sg')
    if os.path.isdir(pro_dir):
        if force_overwrite:
            print("Deleting processed-directory and recreating...")
            shutil.rmtree(pro_dir)
        else:
            print("pro_sg dir already exists. Exiting.")
            return
    raw_data = pd.read_csv(os.path.join(DATA_DIR, 'ratings.csv'), header=0)
```

Figure 4.3.4-3 setup datasets(2)

The code shows how to preprocess the datasets before training.

```
class MultiVAE(CriticModelMixin, StochasticActorModelMixin, BaseModel):
    def __init__(
            self,
            batch_of_users,
            heldout_batch,
            input_dim=None,
            anneal_cap=0.2,
            epochs_to_anneal_over=50,
            batch_size=500,
            evaluation_metric='NDCG',
            lr_actor=1e-3,
            lr_critic=1e-4,
            lr_ac=2e-6,
            ac_reg_loss_scaler=0.0,  # We'll increase it as need be.
            omit_num_seen_from_critic=False,
            omit_num_unseen_from_critic=False,
            **kwargs):
        local_variables = locals()
        local_variables.pop('kwargs')
        self._set_locals(local_variables)
        self.build_graph()
        self.saver = tf.train.Saver()
        if hasattr(self, 'superclass_stuff'):
            self.superclass_stuff()
    def construct_actor_error(self):
        # NOTE: Check the KL term later.
        # NOTE: This will give the wrong number for validation stuff, because batch_of_users isn't the right output!.
        self.actor_error_mask = tf.identity(self.batch_of_users)
        log_softmax = tf.nn.log_softmax(self.prediction)
        actor_error = -tf.reduce_sum(log_softmax * self.actor_error_mask, axis=-1)
        # This way, KL isn't factored into the critic at all. Which is probably what we want, although the AC should have it.
        mean_actor_error = tf.reduce_mean(actor_error) + (
            self.kl_loss_scaler * self.actor_regularization_loss)
        self.actor_error = actor_error
        self.mean_actor_error = mean_actor_error
    def create_logging_ops(self):
        tf.summary.scalar('mean_actor_error', self.mean_actor_error)  # Includes KL term...
        tf.summary.scalar('actor_reg', self.actor_regularization_loss)
        tf.summary.scalar('ndcg@100', tf.reduce_mean(self.true_ndcg))
        tf.summary.scalar('mean_critic_error', self.mean_critic_error)
```

Figure 4.3.4-4 code of RaCT model

In this section, we implement RaCT model by three basic models, actor, critic and the base model. It uses mean square error (MSE) as a metric to justify the actor's prediction and get the top-N ranking items as the final results.

Round 1

First round Train

Round1 is training actor(we reduced the epochs_pred_only to 20 to reduce the training time)

```
[ ]
import sys
import os
UTILS_DIR = os.path.join(os.path.abspath(os.path.dirname('__file__')), '..', 'utils')
sys.path.insert(1, UTILS_DIR)
if __name__ == '__main__':
    BREAK_EARLY = False
    BATCH_SIZE = 500
    for data_subdir in ['ml-20m']:#, 'netflix-prize', 'msd']:
        actor_path = "VAE_ACTOR_TRAIN_{}".format(data_subdir)
        train(
            model_class='multi_vae',
            data_subdir=data_subdir,
            n_epochs_pred_only=20,
            n_epochs_ac_only=0,
            n_epochs_pred_and_ac=0,
            max_kl=0.2,
            ac_reg_loss_scaler=0.0,
            actor_reg_loss_scaler=0.01,
            evaluation_metric="NDCG",
            logging_frequency=50,
            batch_size=BATCH_SIZE,
            break_early=BREAK_EARLY,
            verbose=False,
            version_tag="FULL_RUN_ON_OTHER_DATASETS",
            path_to_save_actor=actor_path,
            log_critic_training_error=False,
        )
        print("Now, hopefully on to testing...")
        test(
            model_class='multi_vae',
            data_subdir=data_subdir,
            n_epochs_pred_only=20,
            n_epochs_ac_only=0,
            n_epochs_pred_and_ac=0,
            max_kl=0.2,
            ac_reg_loss_scaler=0.0,
            actor_reg_loss_scaler=0.01,
            evaluation_metric="NDCG",
            batch_size=BATCH_SIZE,
            break_early=BREAK_EARLY,
            verbose=False,
            version_tag="FULL_RUN_ON_OTHER_DATASETS",
        )
```

Figure 4.3.4-5 Round 1 of training RaCT model

This section is the first step of training of RaCT model which is actor training. The variable n_epochs_pred_only means how many epochs the actor would be trained. Since it costs too much time if the number of epochs is too large. Therefore, we decided to use 20 epochs to train the model. And the time consumption is about 12 minutes.

Round 2

Round2 is training critic(we reduced the epochs_ac and epochs_pred_and_ac to 10 to reduce training time)

```
print("On to round 2! Now we'll do the critic.")
train(
            model_class='multi_vae',
            data_subdir=data_subdir,
            n_epochs_pred_only=0,
            n_epochs_ac_only=10,
            n_epochs_pred_and_ac=10,
            max_kl=0.2,
            ac_reg_loss_scaler=0.0,
            actor_reg_loss_scaler=0.01,
            evaluation_metric="NDCG",
            logging_frequency=50,
            batch_size=BATCH_SIZE,
            break_early=BREAK_EARLY,
            verbose=False,
            version_tag="FULL_RUN_ON_OTHER_DATASETS",
            restore_trained_actor_path=actor_path,
        )
print("Now, hopefully on to testing...")
test(
            model_class='multi_vae',
            data_subdir=data_subdir,
            n_epochs_pred_only=0,
            n_epochs_ac_only=10,
            n_epochs_pred_and_ac=10,
            max_kl=0.2,
            ac_reg_loss_scaler=0.0,
            actor_reg_loss_scaler=0.01,
            evaluation_metric="NDCG",
            batch_size=BATCH_SIZE,
            break_early=BREAK_EARLY,
            verbose=False,
            version_tag="FULL_RUN_ON_OTHER_DATASETS",
            restore_trained_actor_path=actor_path,
        )
print("Bye bye")
exit()
```

Figure 4.3.4-6 Round 2 of training RaCT model

This section is the second and third step of the algorithm which is critic training and alternative training of actor and critic. The variable n_epochs_ac_only controls the epoch of critic training. And n_epochs_pred_and_ac controls the third step.

The complete code can be seen from the link below:

https://colab.research.google.com/drive/1bxyrGw4HW-8Sxp68rYPfRtAECqsC6l5g

4.3.5 Testing

1. Unit testing.

Unit testing is the first stage of the testing. It is the most basic part of the testing, but also one of the most important parts to ensure that all components of our

recommendation system can work properly. Finding problems earlier means that it takes less time to fix them.

After getting the original code from GitHub, we got more than 10 Python files. We ran all the codes separately on our local computer to ensure there were no basic errors.

After running all the Python files, we met some errors.

```
usage: ipykernel_launcher.py [-h] [--force-overwrite FORCE_OVERWRITE]
                             [--dataset DATASET]
ipykernel_launcher.py: error: unrecognized arguments: -f /root/.local/share/jupyter/runtime/kernel-2666be6e-3ce9-4201-9835-c2c84106768b.json
An exception has occurred, use %tb to see the full traceback.

SystemExit: 2
```

Figure 4.3.5-1: Error for Setup_data.py

When we tested the code for data setup, we met an error called 'ipykernel_launcher.py: error: unrecognized arguments'. We found that the problem code was parser.parse_args(). Through multiple modifications and tests, we modified it to parser.parse_known_args()[0] to solve the problem.

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-49919444a4f8> in <module>
      1 import sys
      2 import os
----> 3 UTILS_DIR = os.path.join(os.path.abspath(os.path.dirname(__file__)), '..', 'utils')
      4 sys.path.insert(1, UTILS_DIR)
      5

NameError: name '__file__' is not defined
```

Figure 4.3.5-2: Error for main_vae.py

When we tested the code for training, it occurred an error which is name '__file__' is not defined. Through multiple modifications and tests, we added single quotation marks to __file__ to solve this problem.

So far, all unit tests have passed.

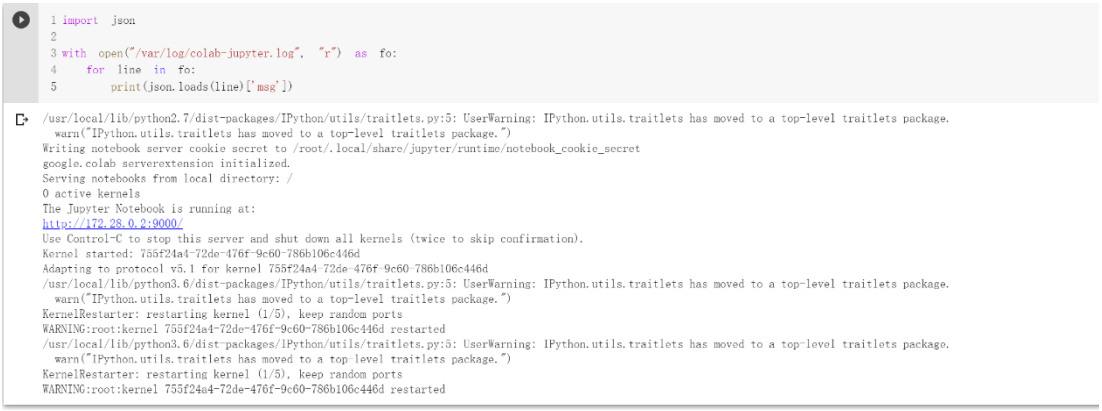
2. Integration testing.

After the unit testing, we conducted integration testing. The purpose of this test is to expose errors in code interaction. After we deployed all Python files to the local computer, we ran all the codes to make sure that there was no error in their interaction.

3. System testing.

System testing tested the code which deployed on Google Colab to verify that it can run smoothly on Colab. It combined with other parts of the system and tested the deployed Colab strictly in the actual operating environment to find potential problems.

After the system testing, we found that it may crash, which causing it to stop running.



```
1 import json
2
3 with open("/var/log/colab-jupyter.log", "r") as fo:
4     for line in fo:
5         print(json.loads(line)['msg'])
```

```
/usr/local/lib/python2.7/dist-packages/IPython/utils/traitlets.py:5: UserWarning: IPython.utils.traitlets has moved to a top-level traitlets package.
  warn("IPython.utils.traitlets has moved to a top-level traitlets package.")
Writing notebook server cookie secret to /root/.local/share/jupyter/runtime/notebook_cookie_secret
google.colab serverextension initialized.
Serving notebooks from local directory: /
0 active kernels
The Jupyter Notebook is running at:
http://172.28.0.2:9000/
Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
Kernel started: 755f24a4-72de-476f-9c60-786b106c446d
Adapting to protocol v5.1 for kernel 755f24a4-72de-476f-9c60-786b106c446d
/usr/local/lib/python3.6/dist-packages/IPython/utils/traitlets.py:5: UserWarning: IPython.utils.traitlets has moved to a top-level traitlets package.
  warn("IPython.utils.traitlets has moved to a top-level traitlets package.")
KernelRestarter: restarting kernel (1/5), keep random ports
WARNING:root:kernel 755f24a4-72de-476f-9c60-786b106c446d restarted
/usr/local/lib/python3.6/dist-packages/IPython/utils/traitlets.py:5: UserWarning: IPython.utils.traitlets has moved to a top-level traitlets package.
  warn("IPython.utils.traitlets has moved to a top-level traitlets package.")
KernelRestarter: restarting kernel (1/5), keep random ports
WARNING:root:kernel 755f24a4-72de-476f-9c60-786b106c446d restarted
```

colab-jupyter.log

Timestamp	Level	Message
Nov 14, 2019, 1:38:25 PM	WARNING	WARNING:root:kernel 755f24a4-72de-476f-9c60-786b106c446d restarted
Nov 14, 2019, 1:38:25 PM	INFO	KernelRestarter: restarting kernel (1/5), keep random ports
Nov 14, 2019, 1:38:22 PM	WARNING	warn("IPython.utils.traitlets has moved to a top-level traitlets package.")
Nov 14, 2019, 1:38:22 PM	WARNING	/usr/local/lib/python3.6/dist-packages/IPython/utils/traitlets.py:5: UserWarning: IPython.utils.traitlets has moved to a top-level traitlets package.
Nov 14, 2019, 1:36:54 PM	INFO	Adapting to protocol v5.1 for kernel 755f24a4-72de-476f-9c60-786b106c446d
Nov 14, 2019, 1:36:49 PM	INFO	Kernel started: 755f24a4-72de-476f-9c60-786b106c446d
Nov 14, 2019, 1:36:19 PM	INFO	Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
Nov 14, 2019, 1:36:19 PM	INFO	http://172.28.0.2:9000/
Nov 14, 2019, 1:36:19 PM	INFO	The Jupyter Notebook is running at:
Nov 14, 2019, 1:36:19 PM	INFO	0 active kernels

Figure 4.3.5-3: Running log of the Colab

We were able to get more insight into what is going on by printing logs contents. We checked the running log and tried to find out what was wrong. We considered that it may be caused by the large size of datasets and the Colab was out-of-memory. We figured it out by running the block one by one.

4. Performance testing.

Performance testing is a non-functional test used to determine the behaviour of our recommendation system under various conditions. We will carry out the load test, stress test, endurance test and etc. Ultimately, we need to keep our recommendation system in a reasonable state of operation.

Also, training speed is one of the performance indexes. We modified the training epoch and tried to find a balance between speed and accuracy. Finally, we chose 20 epochs to achieve speed and performance.

The test accuracy is also one of the performance indexes. We used NDCG to evaluate the model we have trained. NDCG is an evaluation index to measure the quality of ranking, which considers the correlation of all elements. Ndcg@100 refers to top 100 recommended users for effect evaluation. The higher the value, the better the quality of the ranking. Here are the NDCG results:

```
Testing done. That broke it out of the loop.
Test UNNORMALIZED DCG@100=2.10700 (0.40703)
Test NDCG@100=0.41338 (0.00208)
Test Recall@50=0.52753 (0.06401)
Test Recall@020=0.38543 (0.06027)
Test NDCG@0200=0.45101 (0.04576)
Test NDCG@5=0.29737 (0.06048)
Test NDCG@3=0.30568 (0.07006)
Test NDCG@1=0.33880 (0.10583)
```

```
Testing done. That broke it out of the loop.
Test UNNORMALIZED DCG@100=2.15157 (0.42115)
Test NDCG@100=0.41907 (0.00210)
Test Recall@50=0.53056 (0.06409)
Test Recall@020=0.39049 (0.06035)
Test NDCG@0200=0.45629 (0.04614)
Test NDCG@5=0.31080 (0.06220)
Test NDCG@3=0.32176 (0.07192)
Test NDCG@1=0.35240 (0.10682)
```

Figure 4.3.5-4: NDCG score

```
/usr/local/lib/python3.6/dist-packages/matplotlib/cbook/
Passing one of 'on', 'true', 'off', 'false' as a boolean
  warn_deprecated("2.2", "Passing one of 'on', 'true', '
```

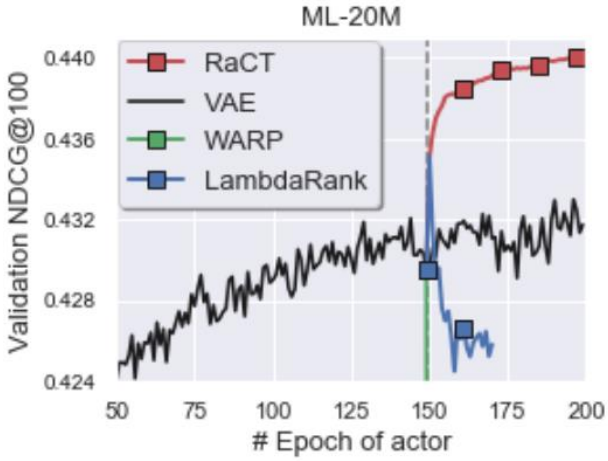


Figure 4.3.5-5: Plot of improvement of RaCT model with ML-20M dataset

We can see that RaCT has the best performance in these 4 methods.

5. Compatibility testing.

Compatibility testing measured how our recommendation system work in different environments. It checks whether the system is compatible with different operating systems, browsers or resolutions. It will ensure that our recommendation system functions are consistently performed in any environment that users use.

We have run it on Cloud Platform (Google Colab), Windows and Mac Os. All systems ran smoothly.

5. RESOURCES

5.1 KernelNet

5.1.1 Hardware & Software

Hardware for the local test:

MacBook8.1

Intel Core M 1.2 GHz 2 cores

L2 Cache (per Core): 256 KB

L3 Cache: 4 MB

Hyper-Threading Technology: Enabled

Memory: 8 GB

The model will run on Python on all major operating systems (Windows, Mac and Linux) on computers.

Softwares included:

Google Colab: For maintaining consistency of hardware.

Git: Source code acquiring + version control

Jupyter Notebook: Code sharing with explanations, data preparation

Google Chrome: API display

Spyder/Atom: The development environment for Python programming

Slack: Project management

Zoom: Online meeting

Trello & Elegantt: For Gantt chart generating and updating.

Python packages used:

For model training:

Pandas

Tensorflow

Numpy

For API building:

Flask

Flask-RESTPlus

Virtualenv

5.1.2 Materials

Research paper: *Kernelized Synaptic Weight Matrices(ICML 2018 Lorenz Muller, Julien Martel, Giacomo Indiveri).*

Github Code of KernelNet:

https://github.com/lorenzMuller/kernelNet_MovieLens

Sample Data (MovieLens-1M): http://www.grouplens.org/system/files/ml-1m.zip

Online tutorial of API building: *Building beautiful REST APIs using Flask, Swagger UI and Flask-RESTPlus*

*(*http://michal.karzynski.pl/blog/2016/06/19/building-beautiful-restful-apis-using-flask-swagger-ui-flask-restplus/)

Google account for accessing Google Colab

5.1.3 Roles & Responsibilities

Researcher: For this model, each member did the research role by doing literature reviews and paper readings.

Programmer: In this model, Zhou Sha and Zaiwo Liu implemented the code on Google Collaboratory, and implemented the final API of this model with the help of other members.

Project member: Each member of our team joined every team meetings, created slides for the presentation, and worked together to write the initial/progress/final report.

Project team leader: Each member took this role in turn weekly.

Minute taker: Each member took their own minutes and shared after meetings.

5.2 Graph Convolutional Matrix Completion

5.2.1 Hardware & Software

Hardware:

The project is developed and run on our personal laptop whose operating system is Windows or Mac.

The hardware includes:

Macbook Pro 8.1(CPU: Intel Core i5; Memory: 8G)

Dell G3 (CPU: Intel Core i7; Memory: 32G)

Software:

GitHub – Backup different versions of codes

Google Drive – Create and share documents and codes, connect with Google Colab

Google Colab – Mount Google Drive, run the codes

Slack – Communicate with group members and tutors

Zoom – Online meeting, share the outcomes every week

Trello – Manage tasks and procedures collaboratively

Python packages used:

Tensorflow

Pandas

Numpy

Scipy

Pickle

Random

5.2.2 Materials

Research paper: *Graph Convolutional Matrix Completion*

https://arxiv.org/abs/1706.02263

Github Code of GC-MC: https://github.com/riannevdberg/gc-mc

Google account for accessing Google Colab

5.2.3 Roles & Responsibilities

Researcher: For this model, each member did the research role by doing literature reviews and paper readings.

Programmer: In this model, Wenjing Deng and Hefei Chen implemented the code on Google Colab to make it hierarchical way and comment the code in the corresponding area, then evaluated the performance of the model.

Project member: Each member attended the weekly meeting on time to solve the problems and exchange their own experience. Also, created slides for the presentation, and worked together to write the initial/progress/final report.

Project team leader: Each member took this role in turn weekly.

Minutes taker: Each member took their own minutes and shared after meetings by Slack.

5.3 Ranking-Critical Training

5.3.1 Hardware & Software

Hardware for the local test:

XPS 15-9575

CPU: Intel Core i7-8705G

Clockspeed: 3.1 GHz

No of Cores: 4 Cores, 8 Logical cores

Turbo Speed: 4.1 GHz

L1 Cache: 256KB

L2 Cache: 1MB

L3 Cache: 8MB

Memory: 16GB, 2400Mhz

Graphics: Radeon RX Vega M GL

Video RAM: 4GB

Softwares:

Github: Source code acquisition and deployment instructions.

Google Colab: Environment consistency, collaborative development and GPU/TPU accelerated.

Jupyter Notebook: Model test, evaluation graphs generation

Slack: Collaboration hub for the project.

Trello: Task allocation, update and management with Kanban system.

Wechat: Timely communication within the group.

Zoom: Online meeting.

Python packages used:

Tensorflow

Pandas

Numpy

Bottleneck

Matplotlib

Scipy

5.3.2 Materials

Github code for Ranking-Critical Training: https://github.com/samlobel/RaCT_CF

Training dataset

Movielens-20M: http://files.grouplens.org/datasets/movielens/ml-20m.zip

Netflix: https://s3-us-west-2.amazonaws.com/cf-datasets/netflix-prize-data.zip

MSD: http://labrosa.ee.columbia.edu/millionsong/sites/default/files/challenge/train_triplets.txt.zip

Paper review: https://arxiv.org/abs/1906.04281

5.3.3 Roles & Responsibilities

Researcher: For this model, each member did the research role by doing literature reviews and paper readings.

Programmer: In this model, Kaijia Chen and Wentao Wu implemented the code on Google Colab to make it hierarchical way and comment the code in the corresponding area, then evaluated the performance of the model.

Project member: Each member attended the weekly meeting on time to solve the problems and exchange their own experience. Also, created slides for the presentation, and worked together to write the initial/progress/final report.

Project team leader: Each member took this role in turn weekly.

Minutes taker: Each member took their own minutes and shared after meetings by Slack.

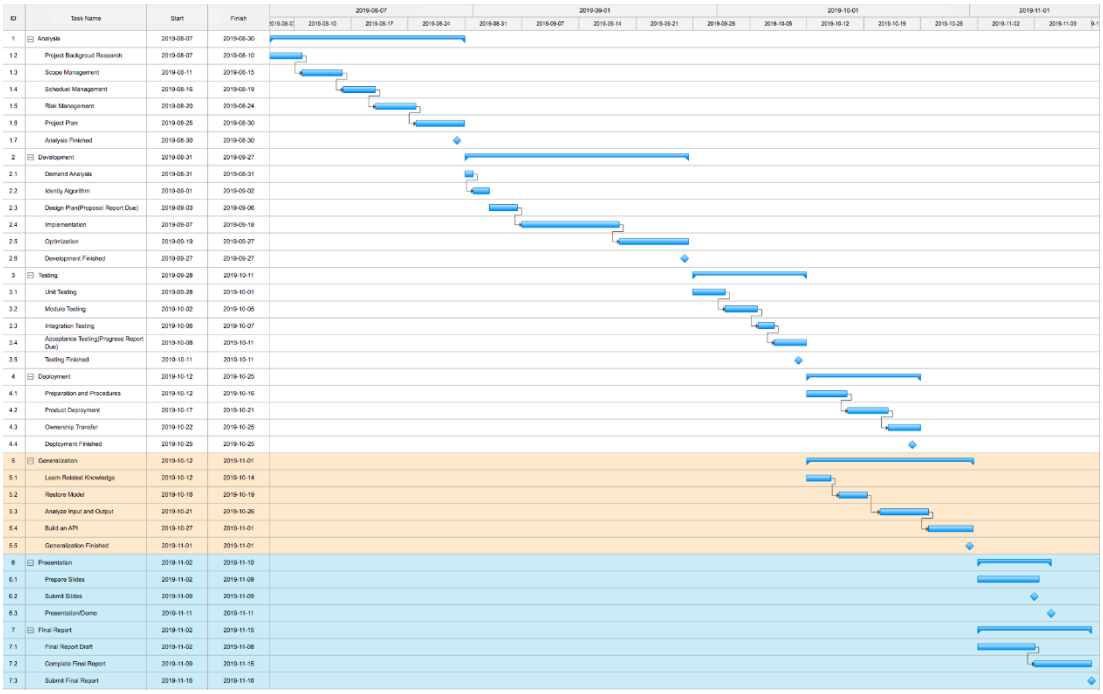
6. MILESTONES / SCHEDULE

The figure below is the original schedule of our project. We have successfully completed several stages of analysis, development and testing. During the deployment phase, we have a meeting with our tutors and clients to build an API which makes it easy for our model to be applied by the website platform. Therefore, we add a generalization stage and update our schedule and milestones table.

ID	Task Name	Start	Finish
1	Analysis	2019-08-07	2019-08-30
1.1	Project Backgroud Research	2019-08-07	2019-08-10
1.2	Scope Management	2019-08-11	2019-08-15
1.3	Schedual Management	2019-08-16	2019-08-19
1.4	Risk Management	2019-08-20	2019-08-24
1.5	Project Plan	2019-08-25	2019-08-30
1.6	Analysis Finished	2019-08-30	2019-08-30
2	Development	2019-08-31	2019-09-27
2.1	Demand Analysis	2019-08-31	2019-08-31
2.3	Identiy Algorithm	2019-09-01	2019-09-02
2.3	Design Plan(Proposol Report Due)	2019-09-03	2019-09-06
2.4	Implementation	2019-09-07	2019-09-18
2.5	Optimization	2019-09-19	2019-09-27
2.6	Development Finished	2019-09-27	2019-09-27
3	Testing	2019-09-28	2019-10-11
3.1	Unit Testing	2019-09-28	2019-10-01
3.2	Module Testing	2019-10-02	2019-10-05
3.3	Integration Testing	2019-10-06	2019-10-07
3.4	Acceptance Testing(Progress Report Due)	2019-10-08	2019-10-11
3.5	Testing Finished	2019-10-11	2019-10-11
4	Deployment	2019-10-12	2019-10-18
4.1	Preparation and Procedures	2019-10-12	2019-10-14
4.2	Product Deployment	2019-10-15	2019-10-16
4.3	Ownership Transfer	2019-10-17	2019-10-18
4.4	Deployment Finished	2019-10-18	2019-10-18
5	Training and Documentation	2019-10-19	2019-11-08
5.1	Training Support	2019-10-19	2019-10-19
5.2	Consultant Support	2019-10-20	2019-10-21
5.3	Ducument Support	2019-10-22	2019-10-23
5.4	Technical Support	2019-10-24	2019-10-25
5.5	Prepare Presentation	2019-10-26	2019-10-30
5.6	Complete Final Report	2019-10-31	2019-11-08
5.7	Training and Decumentation Finished	2019-11-08	2019-11-08

Figure 6-1: Original project schedule

The figure below is the updated schedule of our project. In the new schedule, we extend the deployment phase and carry out the generalization phase at the same time. In addition, we skip the training & documentation phase which is unnecessary. Because of the deadline of presentation and report are extant, we adjust the schedule of our project and update the arrangement to prepare and accomplish better results.



ID	Task Name	Start	Finish
1	Analysis	2019-08-07	2019-08-30
1.2	Project Backgroud Research	2019-08-07	2019-08-10
1.3	Scope Management	2019-08-11	2019-08-15
1.4	Schedual Management	2019-08-16	2019-08-19
1.5	Risk Management	2019-08-20	2019-08-24
1.6	Project Plan	2019-08-25	2019-08-30
1.7	Analysis Finished	2019-08-30	2019-08-30
2	Development	2019-08-31	2019-09-27
2.1	Demand Analysis	2019-08-31	2019-08-31
2.2	Identiy Algorithm	2019-09-01	2019-09-02
2.3	Design Plan(Proposol Report Due)	2019-09-03	2019-09-06
2.4	Implementation	2019-09-07	2019-09-18
2.5	Optimization	2019-09-19	2019-09-27
2.6	Development Finished	2019-09-27	2019-09-27
3	Testing	2019-09-28	2019-10-11
3.1	Unit Testing	2019-09-28	2019-10-01
3.2	Module Testing	2019-10-02	2019-10-05
3.3	Integration Testing	2019-10-06	2019-10-07
3.4	Acceptance Testing(Progress Report Due)	2019-10-08	2019-10-11
3.5	Testing Finished	2019-10-11	2019-10-11
4	Deployment	2019-10-12	2019-10-25
4.1	Preparation and Procedures	2019-10-12	2019-10-16
4.2	Product Deployment	2019-10-17	2019-10-21
4.3	Ownership Transfer	2019-10-22	2019-10-25
4.4	Deployment Finished	2019-10-25	2019-10-25
5	Generalization	2019-10-12	2019-11-01
5.1	Learn Related Knowledge	2019-10-12	2019-10-14
5.2	Restore Model	2019-10-16	2019-10-19
5.3	Analyze Input and Output	2019-10-21	2019-10-26
5.4	Build an API	2019-10-27	2019-11-01
5.5	Generalization Finished	2019-11-01	2019-11-01
6	Presentation	2019-11-02	2019-11-10
6.1	Prepare Slides	2019-11-02	2019-11-09
6.2	Submit Slides	2019-11-09	2019-11-09
6.3	Presentation/Demo	2019-11-11	2019-11-11
7	Final Report	2019-11-02	2019-11-15
7.1	Final Report Draft	2019-11-02	2019-11-08
7.2	Complete Final Report	2019-11-09	2019-11-15
7.3	Submit Final Report	2019-11-16	2019-11-16

Figure 6-2: The updated project schedule

We also update the milestones table to outline the lasted milestones of the project weekly.

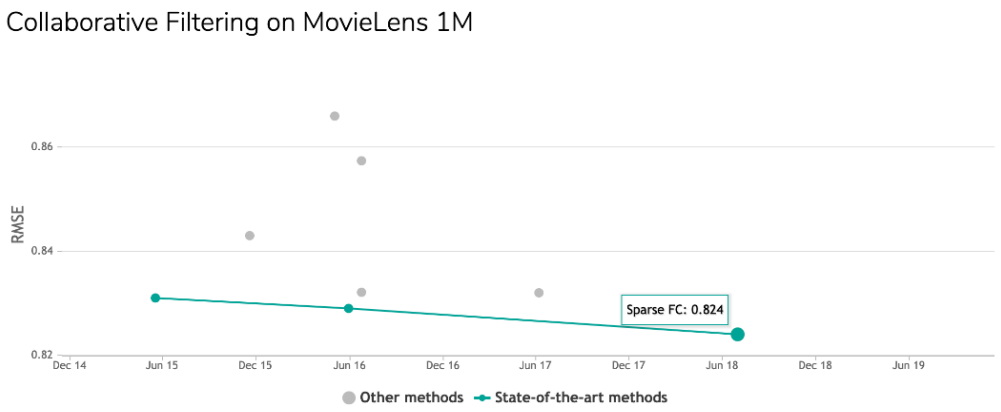
1 Milestone	Tasks	Reporting	Date
Week-1	Understand the project & research background knowledge	None	2019-08-09
Week-2	Meet with our client and define the project plan	Client meeting to review the project	2019-08-16
Week-3	Identify the requirements of the project clearly & Analyze two websites	Client meeting to identify the requirement about the project	2019-08-23
Week-4	Focus on the recommendation part & Do literature review	Client meeting to compare the difference between two websites	2019-08-30
Week-5	Proposal Report Due & Define recommendation system algorithm	Client meeting to understand the details of papers	2019-09-06
Week-6	Complete design plan	Client meeting to review the design plan	2019-09-13
Week-7	Implement recommendation system	Client meeting to review Implementation	2019-09-20
Week-8	Optimization	Client meeting to review final performance	2019-09-27
Week-9	Testing & Progress Report Due	Client meeting to review test outcome	2019-10-11
Week-10	~~Deployment~~ Deployment & Generalization I	Client meeting to deploy the system	2019-10-18
Week-11	~~Training & Documentation~~ Deployment & Generalization II	Client meeting to review the input and output of models	2019-10-25
Week-12	~~Final Presentation~~ Generalization III	Client meeting to review the implementation of API	2019-11-02
Week-13	~~Final Report (thesis)~~ Final Presentation slides	Client meeting to review presentation slides	2019-11-09
Week-14	Final Presentation & Final Report (thesis)	None	2019-11-16

7. RESULTS

7.1 Testing and Evaluation of three models

7.1.1 Kernelized Synaptic Matrices

The code for Kernelized matrices ranks first when training the data source Movielen-1m. It is not surprised to have the lowest RMSE value when dealing with the medium dataset because the changing weight depends on the dataset size to some degree.



RANK	METHOD	RMSE	PAPER TITLE	YEAR	PAPER	CODE
1	Sparse FC	0.824	Kernelized Synaptic Weight Matrices	2018		
2	CF-NADE	0.829	A Neural Autoregressive Approach to Collaborative Filtering	2016		
3	I-AutoRec	0.831	AutoRec: Autoencoders Meet Collaborative Filtering	2015		
4	GC-MC	0.832	Graph Convolutional Matrix Completion	2017		
5	I-CFN	0.8321	Hybrid Recommender System based on Autoencoders	2016		
6	NNMF	0.843	Neural Network Matrix Factorization	2015		
7	U-CFN	0.8574	Hybrid Recommender System based on Autoencoders	2016		
8	Factorization with dictionary learning	0.866	Dictionary Learning for Massive Matrix Factorization	2016		

Figure 7.1.1-1: RMSE value for Collaborative Filtering methods on Movielens 1M (source:https://paperswithcode.com/sota/collaborative-filtering-on-movielens-1m)

When training the large dataset, the free parameters will increase, and it will lead to the complex and cumbersome calculations. It directly makes the Kernelized matrices perform not that good when dealing with the Movielen-10m dataset.

7.1.2 GC-MC model

The figure below shows the RMSE scores of the different models in the ML-100K. The lower the score of the RMSE, the better the performance of the model. Compared to other models, GC-MC has an RMSE score of 0.910, which is the lowest score in the evaluation model. Therefore, in ML-100K, GC-MC performance is the best of all evaluation models.

Model	ML-100K + Feature (RMSE)
MC	0.973
IMC	1.653
GMC	0.996
sRGCNN	0.929
GC-MC	0.910

Table 7.1.2-1 RMSE Score for Movie Lens 100K

7.1.3 RaCT model

```
Test UNNORMALIZED DCG@100=2.10700 (0.40703)
Test NDCG@100=0.41338 (0.00208)
Test Recall@50=0.52753 (0.06401)
Test Recall@020=0.38543 (0.06027)
Test NDCG@0200=0.45101 (0.04576)
Test NDCG@5=0.29737 (0.06048)
Test NDCG@3=0.30568 (0.07006)
Test NDCG@1=0.33880 (0.10583)
Round 1 done
```

Figure 7.1.3-1 Testing result of Round 1

On the first Round, the NDCG evaluation value is close the value that in the training process.

```
Test UNNORMALIZED DCG@100=2.15157 (0.42115)
Test NDCG@100=0.41907 (0.00210)
Test Recall@50=0.53056 (0.06409)
Test Recall@020=0.39049 (0.06035)
Test NDCG@0200=0.45629 (0.04614)
Test NDCG@5=0.31080 (0.06220)
Test NDCG@3=0.32176 (0.07192)
Test NDCG@1=0.35240 (0.10682)
Bye bye
```

Figure 7.1.3-2 Testing result of Round 2

By the critic training, we can see the evaluation value is better than the first round one.

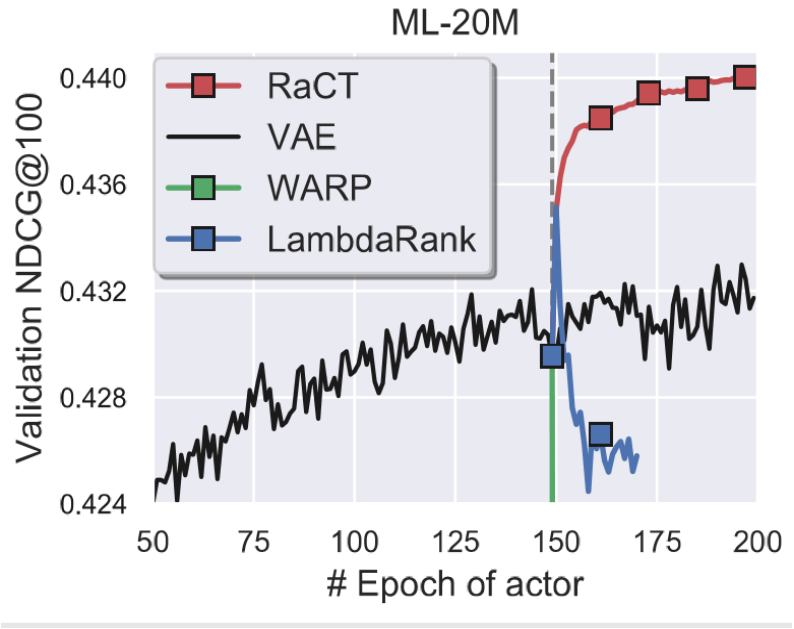


Figure 7.1.3-3 Plot of improvement of RaCT model with ML-20M dataset

This is a plot of improvement of RaCT model with ML-20M dataset. We can see there is a big difference after the epoch more than 150.

Dataset	ML-20M			Netflix			MSD		
Metric	R@20	R@50	NDCG@100	R@20	R@50	NDCG@100	R@20	R@50	NDCG@100
RaCT	**0.403**	**0.543**	**0.434**	**0.357**	**0.450**	**0.392**	**0.268**	**0.364**	**0.319**
VAE‡	0.396	0.536	0.426	0.350	0.443	0.385	0.260	0.356	0.310
WARP	0.314	0.466	0.341	0.270	0.365	0.306	0.206	0.302	0.249
LambdaNet	0.395	0.534	0.427	0.352	0.441	0.386	0.259	0.355	0.308
VAE	0.395	0.537	0.426	0.351	0.444	0.386	0.266	**0.364**	0.316
CDAE	0.391	0.523	0.418	0.343	0.428	0.376	0.188	0.283	0.237
WMF	0.360	0.498	0.386	0.316	0.404	0.351	0.211	0.312	0.257
SLIM	0.370	0.495	0.401	0.347	0.428	0.379	–	–	–

Figure 7.1.3-4 Comparison among RaCT model and other models with different datasets

We also test the model by using other two datasets, Netflix and MSD. We can see the RaCT model performs better than all other state-of-the-art models.

7.2 Design and Development of APIs

Our final results are the APIs we built based on our models. The final APIs are based on the KernelNet model, because we found the KernelNet works best among the three models on the MovieLens-1M, while GCMC works best on the MovieLens-100K and RaCT works best on MovieLens-10M. We assume that the normal size of data for recommendation systems should be around 1 megabyte, which is of similar size with the MovieLens-1M. Thus, we chose the KernelNet model to be our final model.

Before the API design, we trained and tested the model locally with the sample data. The hardware we used to train the model is the same as the former section, and the environment we built is simply Python with Tensorflow. However, the training can also be done on the cloud by Google Colab. Then, after training the model, we use the trained model to make predictions on the original sparse matrix. In the predictions, we had all the ratings from each user to each item, which we call it a user rating matrix. By extracting this matrix from the output and saving it locally, we can furtherly use it as the input of our APIs, or we could access it from our APIs to generate the recommendations. The last way is what we had accomplished.

In order to develop the REST APIs, we are using Flask, Flask-RESTPlus and Swagger UI with Python programming. In the APIs we had implemented several functions to request some information of the user from the database (Figure 7。2-1),

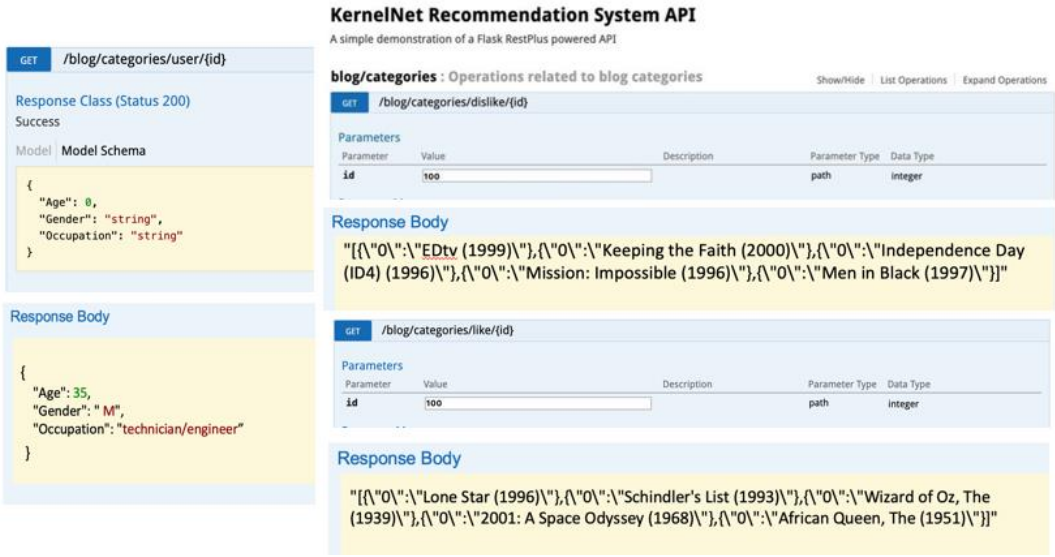


Figure 7.2-1

then using the information with the trained model to get the recommended items for the user (Figure 7.2-2). For example, we can get the top-5 or the lowest 5 rated items of one user, where we can gain some ideas of the type of items (movies) this user might like/dislike. Also, we can get this user's persona from the database, such as gender, occupation, age group, etc. After we familiarize with this user, we can use the whole dataset to predict the ratings he/she might give to all items, then use the prediction (rating matrix) to recommend the top-x items with highest ratings for this particular user. In conclusion, in the APIs you only need to have the user's id, then you can get the user's profile and the recommendations easily.

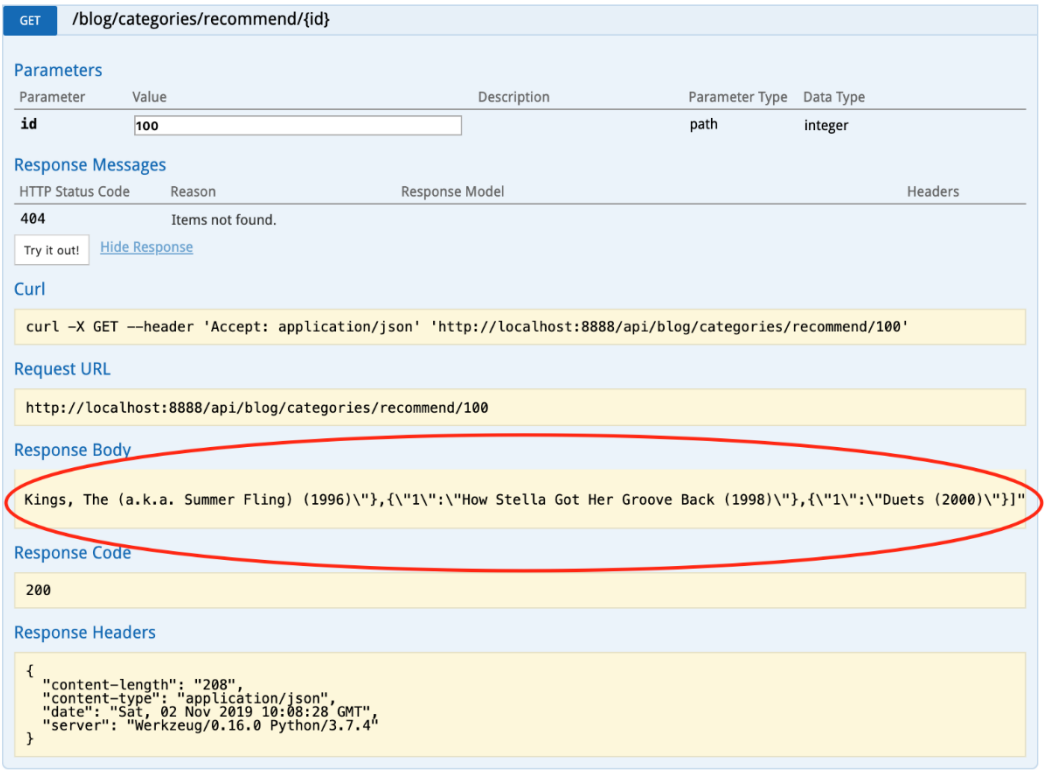


Figure 7.2-2

In order to view and test our APIs as above, we used Google Chrome as the explorer and used the Virtualenv under Python to create the virtual machine and run it on the localhost server (Figure 7.2-3). Then by typing HTTPS://localhost:8888/api, we can be able to see the above REST APIs.

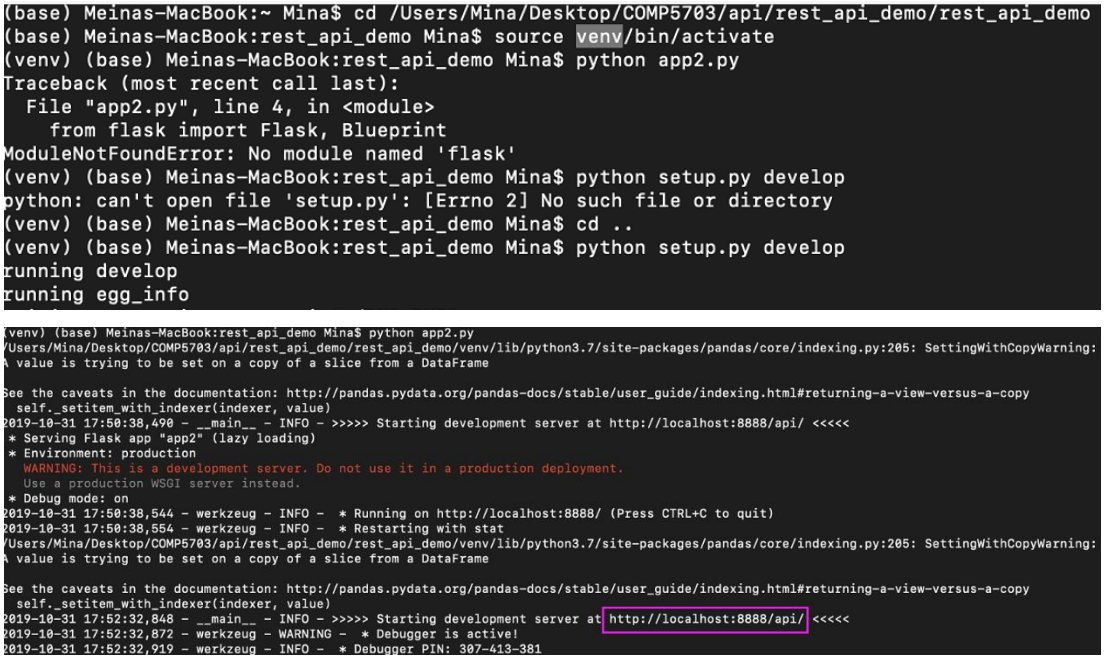


Figure 7.2-3

7.3 Challenges

There were many challenges occurred before we get our final results, these challenges can be classified into two parts. The first challenging part is about extracting the rating matrix from the code, while the second challenging part is about API design.

Initially, when we tried to extract the useable output from the original code, we have to determine where the matrix has been created in the code, which means we are required to not just run the code, but also understand the code written by machine learning algorithms line by line to locate that output. We finally found the matrix by reading the research paper of this model closely and used the paper to navigate the section to find the matrix. Then, after we found the matrix, we decided to store it into a text file for further use.

Next, we are facing obstacles when we design the APIs because we are never familiar with API design, and we are following one simple online tutorial from scratch to build the APIs, and integrate our own functions into the sample API page. During the designing phase, we did a whole lot of debugging and testings and communicated with each member frequently to solve these problems together with the help of our supervisor. Since many issues are networking related, we kept searching and learning from the Internet.

7.4 Deliverables

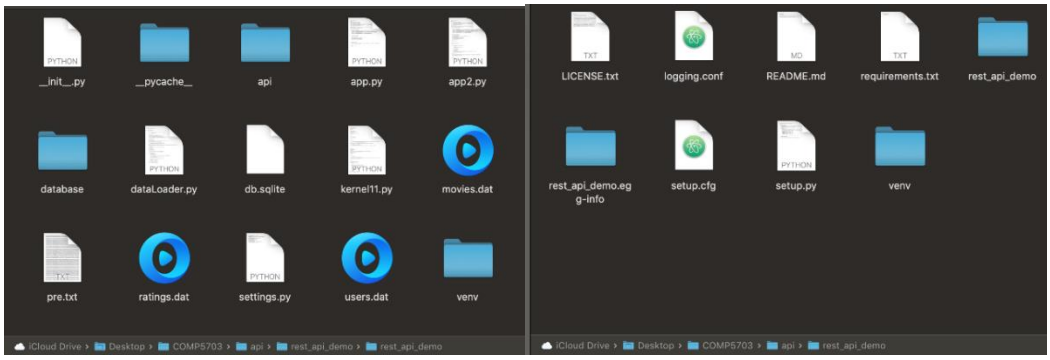


Figure 7.4-1

Consequently, we had our deliverables compressed into one file. In the zip file (Figure 7.4-1), we have the sample data which is MovieLens-1M; the KernelNet model's code for training and testing; the rating matrix generated by the trained model saved as a text file; the Python code of APIs we built; and the requirement of the environment to run the APIs as another text file.

8. DISCUSSION

With a semester work, we implement three improved models and build a REST (Representational State Transfer) API for the KernelNet model. Through our test, we found the performance of these three models are all better than state-of-the-art models, especially in the large datasets and with more epochs of training. In the results part, we can see each evaluation of model performs better compared with its related model. The KernelNet model gets the lowest score on RMSE evaluator among CF-NADE, I-AutoRec, GC-MC, I-CFN, NNMF and U-CFN models. Coincidentally, our second model is GC-MC. Its performance is the best among its related models, such as MC, IMC, GMC and sRGCNN. However, it is not good as the KernelNet one. Since they all use the similar dataset to do the test so that we can know the KernelNet model is better. These two models are both improved the matrix representation during the training process. While for the RaCT model, it improves one of the state-of-the-art methods, VAEs model, by adding one more step to measure the prediction results so that it can enhance the accuracy of prediction. Its evaluation method is also different from others. Its performance is better than its related models, such as VAE, WARP, LambdaNet. And we use the larger datasets to train and test the model because one of

its advantages is scalable for large datasets. We not only use movies datasets but also use songs dataset. It is stable and consistently superior to other models.

Even though most current used models show a good result in the recommended system, the improved model still has its advantages. Because of the increasing number of data and the growing capacity of data sets, the ability of model processing must be improved, and the speed must be accelerated to adapt to this change. Therefore, the improved models just figure out the problem correctly.

And we also build a REST API for KernelNet model so that some web developer can call our API for their recommendation system part of the website. From the results, we can see it is clear and easy to show the recommended items. And it can show the user's personal information as well. The REST API is the most common way in building API. It is easy for a programmer to implement and widely used by developers due to the development of JSON. Therefore, our API is closely following the development of technology.

9. LIMITATIONS AND FUTURE WORKS

There are three issues that need to be discussed regarding the limitations of this project. The first is that the selected three models have different performances in different scales of data sets. For example, RaCT performs best in large-scale data sets (ML-20M), and KernelNet performs best in medium-sized data sets (ML-1M), as well as GC-MC performs best in small-scale data sets (ML-100k). Although we finally chose the KernelNet model, as the size of the website data increases, the performance of KernelNet will gradually decline. As a result, KernelNet is not able to perform well on sites of different data sizes.

The second limitation is that the project is primarily aimed at film-related data sets and websites, and there is no test and evaluation model for the performance of data sets other than movies. Therefore, the API ultimately delivered by this project is theoretically applicable only to the film recommendation system. The performance of this model is not convincing for recommendation systems other than movies without testing other domain data sets.

The last problem is the lack of adjustments to the parameters during the testing of the three models. Due to the long training time, it is impossible to adjust the parameters in a wide range, and only the parameter selection can be slightly adjusted, so the final performance may be affected.

In the future work, our project team hopes to include all three models in the built API, and can detect the size of the website database, so that different models can be called as recommended systems according to different scales. As a result, the accuracy and performance of the recommendation system cannot be improved, and the compatibility of the API can be increased, and more models can be added to the API for the future.

Or, we will try another method to do a more in-depth study of each model, including parameter adjustments and functional extensions, so that a single model can perform well in data sets of different sizes. This also achieves the goal of improving the performance of the recommended system.

At the same time, the use of only film-related data sets limits the performance of the model, and we hope to improve these models and test more types of data sets. To process various datasets containing text, images and other information for use on more recommended platforms.

REFERENCES

Berg, R., Kipf, T., & Welling, M. (2017). Graph Convolutional Matrix Completion.

Ciaccia, P., & Patella, M. (2011). Metric information filtering. Information Systems, 36(4), 708-720. doi: 10.1016/j.is.2010.09.007

Eliason, S. (2009). Maximum likelihood estimation. Newbury Park, Calif.: Sage.

Hanani, U., Shapira, B., Shoval, P. (2001) Information filtering: Overview of issues, research and systems. User Modeling and User-Adapted Interaction, 11, pp. 203–259.

Hendry, D., & Nielsen, B. (2012). Econometric Modeling. New Jersey: Princeton University Press.

Herlocker, J. &. (2001). Explaining Collaborative Filtering Recommendations. Proceedings of the ACM Conference on Computer Supported Cooperative Work. Minneapolis.

Jian, W., Jianhua, H., Kai, C., Yi, Z., & Zuoyin, T. (2016, 11 28). Collaborative Filtering and Deep Learning Based Recommendation System For Cold Start Items. Expert Systems With Applications, p. 8.

Muller, L. K., N. P. Martel, J., & Indiveri, G. (2018). Kernelized Synaptic Weight Matrices. International Conference on Machine Learning.

Paolo, M., & Paolo, A. (2004). Trust-aware Recommender Systems. On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, OTM Confederated International Conferences (pp. 492-508). Cyprus: Agia Napa.

Rendle, S. &. (2019, 5). On the Difficulty of Evaluating Baselines: A Study on Recommender Systems. Israel. Retrieved from https://www.researchgate.net/publication/332897835_On_the_Difficulty_of_Evaluating_Baselines_A_Study_on_Recommender_Systems/citation/download

Rossi, Richard J. (2018). Mathematical Statistics : An Introduction to Likelihood Based Inference. New York: John Wiley & Sons. p. 227. ISBN 978-1-118-77104-4.

Sarwar, B., Karypis, G., Konstan, J., & Riedl, J. (2001). Item-based Collaborative Filtering Recommendation Algorithms. Proceedings of ACM World Wide Web Conference, (p. 3). Hong Kong.

Tauritz, D. (2002). Adaptive information filtering. [Netherlands]: [Institute for Programming Research and Algorithms].

Terveen, L. G., Herlocker, J. L., & Konstan, J. A. (2004, 1 22). Evaluating collaborative filtering recommender systems. ACM Transactions on Information Systems, pp. 5-33.

Wang, Y., Meng, X., & Hu, X. (2014). Information Aging-based Collaborative Filtering Recommendation Algorithm. Journal Of Electronics & Information Technology, 35(10), 2391-2396. doi: 10.3724/sp.j.1146.2012.01743

Weinan, Z., Tianming, D., & Jun, W. (2016). Deep Learning over Multi-field Categorical Data: A Case Study on User Response Prediction. In F. Nicola, & C. Fabio, Advances in Information Retrieval. ECIR 2016. Lecture Notes in Computer Science (pp. 45-57). Springer, Cham.

Ying, S., T. Ryan, H., Jian, J., Haijing, W., & Yu, D. (2016). Deep Crossing: Web-Scale Modeling without Manually Crafted Combinatorial Features. KDD '16 Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 255-256). California: ACM