ADVDM Assignment 1

25th September 2018

1 Introduction

This is the report of the first assignment in the course advances in data mining (4343ADVDM). The task is to implement different algorithms for predicting ratings for certain items by certain users while minimizing the prediction error and optimizing the performance. In this report, we will present our solution by explaining the task, our goals and our implementation of different algorithms, their results. These results are discussed in detail to communicate our findings regarding the algorithms and which approach is best suited.
The goal of this assignment is to study the different approaches to recommender systems. By applying these approaches to a large data set consisting of 1 million entries together with the cross-validation, we compare the difference in accuracy and performance of different algorithms. Utilizing our own implementation, we can calculate approximately how much the CPU runtime and memory are affected by the size of the data set and the approach used. The Root Mean Squared Error (RMSE) and the Mean Absolute Error (mea) are used as standardized measures for assessing the accuracy of the approaches. Rather than analyst every single RMSE and mea, comparison with other algorithm's RMSE and mea was done to prove the difference in the accuracy of different algorithms.

2 Background

2.1 Cross validation

Since we want to check the accuracy of our approaches on data that has not been used during the training process, an implementation of 5-fold cross validation is necessary. With this technique the data set is split into 5 different parts, 4 out of 5 parts at random are used as the training set and the remaining one as the test set.
For our implementation, the whole data set was randomly shuffled, split into 5 parts evenly and then every 1 of the parts becomes the test set while the rest becomes the training set. However, compared to the given implementation from the enclosed python file, both our training and test sets were not sorted. Knowing that we can make use of the sorted data sets in both the implementation of the algorithms, testing and error searching, we decided to use the given cross validation implementation instead. This implementation simply assigns each row in the data a set number which represents the set that the row belongs to (0 to 4). This set number is calculated by taking the row's order number modulus 5. After that, a shuffle is done to make sure that the whole data set is randomly and evenly split into different sets. Finally, we loop through all possible set numbers (0 to 4) using the current set number as the test set and the rest as a train set.

2.2 Naive Approaches

- R_{global}(User, Item)=mean(all ratings)
- R_{item}(User, Item)=mean(all ratings for Item)
- R_{user}(User, Item)=mean(all ratings for User)
- $R_{user\text{-}item}(User, Item)=\alpha * R_{user}(User, Item) + \beta * R_{item}(User, Item) + \gamma$

(parameters α, β, γ estimated with Linear Regression)

First, the global mean is calculated as the mean of all the data points in the set. The user and item-specific means are similar in both theory and implementation, where all the data points which belong to the same user or item are grouped and their mean calculated. These two approaches give us the mean rating of every user or on every item. The last approach is more complex, different and requires the result from the above approaches. Using a linear combination of the user and item means, a prediction matrix is made in order to estimate the ratings for each user and item.

2.3 Matrix Factorization

An alternative to the naive approaches for estimating the rating of an unknown item for a certain user is the algorithm called "Matrix Factorization"[1] coupled with "Gradient Descent" [2]. Matrix factorization means that one matrix M is broken down into two matices which - if multiplied with one another using the dot product approach- result in M.

Gradient descent is an iterative approach, where the minimum of a function is estimated over time. It can thus be used to find a local minimum. This is accomplished by adding a value to the target variable that is negatively proportional to the gradient of the function of the current value of the target variable. Combining these two approaches, one can estimate the cell values of the two Factor Matrices by minimizing a cost function that computes the difference between the true value of the large matrix and the dot product of the two factor matrices. The approximation approach is useful in situations where the performance and storage requirements are too high for the exact matrix factorization to run. This algorithm is executed together with cross-validation in order to better estimate the algorithm's accuracy and address the problem of overfitting. Thus, the ratings data-set is divided into five blocks containing random entries (the whole data set is shuffled beforehand). Afterwards, four blocks together are used as training data, while the remaining one is used to test the accuracy of the model. This is repeated five times, with the test set being a different one each time. The accuracy of the algorithm is presented in the following chapter. The algorithm is structured as follows:

- Set parameters for learning rate and regularization, set parameter K (parameter which can be chosen freely, in this implementation its value is 10)
- Create two matrices U and I with the following dimensions. U: [rows of M, K], I: [K, columns of M]
- Initialize values of U and I randomly
- For each item of M m_{ij}:
  - Compute the difference e_{ij} of the dot-product of $u_{i:,}$ (i^{th} row of U), $i_{:,j}$ (j^{th} column of I), and the value of m_{ij}
  - Update values of $I_{:,j}$ and $U_{i:,}$ using the gradient of ${e_{ij}}^2$

$$U_{ik} = U_{ik} + l * (2e_{ij} * I_{ij} - r * U_{ik})$$

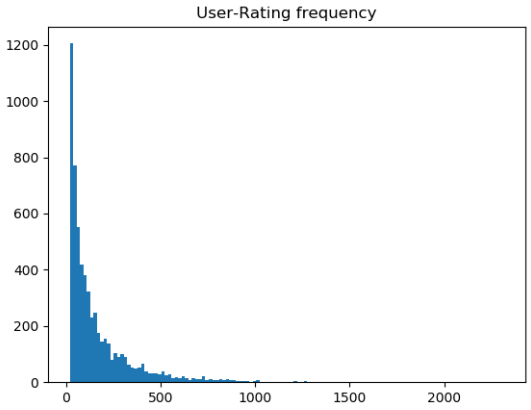
$$I_{ik} = I_{ik} + l * (2e_{ij} * U_{ij} - r * I_{ik})$$

- Calculate Root Mean Squared Error (RMSE) and Mean Average Error for the test set

3 Implementation

3.1 Data Exploration

The dataset that was made available contains 1,000,209 ratings with four columns detailing its properties. The first column contains the Id of the user who rated the item. The item's id is contained in the second column, whereas the rating itself is saved in the third one. A UNIX-timestamp is also provided which is stored in the fourth column. While there are 1 million rating records, the number of items and users are less, since multiple users rated multiple items and the same item can be rated by different users. After counting the distinct number of values, the results show that there are 3706 item and 6040 users. On average, a user rated 165 items. However, this number does not accurately show the distribution of the user's rate frequency and its outliers. In this case, a histogram could be helpful.



The histogram shows that there is a long tail distribution of the user ratings. While the minimum frequency is 20, the maximum frequency is 2314. The data shows that there are 41 users with more than 1000 ratings where the top 1% of users have at least 825 ratings. Looking at the side of the spectrum, the least 1% have at most 20 ratings, which also is the minimum value. Increasing the interval to 30, the number rises to 809. From this, we can assume, that the distribution is front-loaded as shown in the histogram.

3.2 Naive Approaches

3.2.1 Algorithm

After the implementation of the 5-fold cross-validation, an implementation of the first 4 formulas of the naive approach was required. After calculating the global mean, RMSE and MAE formulas are applied for the difference between the actual ratings and the global mean.

$$\text{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - y_j| \qquad \text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(P_i - O_i)^2}{n}}$$

After that, the calculation of the mean for all ratings for each item and user is done with help of numpy_indexed.group_by() function. This function gathers all ratings which belong to the same user or item and calculate the mean. However, knowing that the cross-validation might give us a training set with missing all data points for some users or items, a fall-back algorithm using the global rating value was implemented. By creating a list of all existing users and items in the data set, a comparison with the users and items exist in the training set could be done with help of the set.symmetric_difference() function. The missing element is inserted in its corresponding index with the value of a global mean rating. The next step is to generate two replicated lists of users and items using a list comprehension. They follow the order of the training set

but with the mean of the user or item instead of the Id. These two lists will be used to calculate the RMSE and MAE by using vector subtraction and as input for the last naive approach. Using the same user mean and rating mean from the training set but the new replicated lists from the test set, the test RMSE and MAE can be calculated.

Before generating new replicated lists of users and items from the test set, we can use the available replicated lists as input for the np.linalg.lstsq() function to perform a linear regression. By concatenating the two replicated lists and a 1's vector, we have the three column matrix required to find the most fitted linear expression. α, β, γ are given by the linear expression and also the parameters which we need to predict the rating of a user on an item. Since the prediction can exceed the 1 to 5 limit of a rating, a constraint was added to convert all predictions greater than 5 to 5 and predictions less than 1 to 1. The final step is to approximate the ratings on the test set and calculate the RMSE and MAE.

3.2.2 Results

- First approach:

	Train RMSE	Train MAE	Test RMSE	Test MAE
1	1.11731523	0.93396319	1.11624744	0.93422056
2	1.11701016	0.93394206	1.11746575	0.93343997
3	1.11699529	0.9337667	1.11752522	0.93411512
4	1.11745511	0.93419246	1.11568491	0.93242426
5	1.11672984	0.93343936	1.11858662	0.93510661
mean	1.11710113	0.93386075	1.11710199	0.9338613

- Second approach:

	Train RMSE	Train MAE	Test RMSE	Test MAE
1	1.02785404	0.82238788	1.03473763	0.8293142
2	1.02778599	0.82291956	1.03500099	0.82831617
3	1.02773065	0.82296344	1.03527038	0.82887167
4	1.02781842	0.82302252	1.03492534	0.82776207
5	1.02717027	0.82239199	1.03752907	0.83057375
mean	1.02767187	0.82273708	1.03549268	0.82896757

- Third approach:

	Train RMSE	Train MAE	Test RMSE	Test MAE
1	0.9744298	0.77824597	0.97849937	0.78209687
2	0.97409031	0.77836482	0.98006458	0.78219076
3	0.97433361	0.77824482	0.97906966	0.78270683
4	0.97460831	0.77893227	0.97780916	0.78082438
5	0.97362513	0.77790328	0.98185046	0.78420163
mean	0.97421743	0.77833823	0.97945865	0.7824041

- Fourth approach:

	Test RMSE	Test MAE
1	0.92285057	0.0.7320158
2	0.92513506	0.7329174
3	0.92361311	0.73227772
4	0.92349392	0.73169071
5	0.92694415	0.734877
mean	0.92440736	0.73275573

3.3 Matrix Factorization

3.3.1 Algorithm

The matrix factorization algorithm is constructed of two main functions. The first one is called `sparseMatrix` and takes the rating data (subset) as its parameter.

```
def sparseMatrix(ratings_data, num_users, num_item):
    sparseM = lil_matrix((np.max(ratings_data[:,1]),np.max(ratings_data[:,0])))
    for i in range((ratings_data.shape[0])):
        sparseM[ratings_data[i, 1]-1, ratings_data[i, 0]-1] = ratings_data[i, 2]
    return sparseM
```

First, the original matrix M is constructed from the given rating data set, since its structure is unsuited for index calls and reads. A sparse matrix is created since there are many empty/None values in the matrix where there are no ratings. These empty spaces are compressed with a sparse matrix, thus requiring far less RAM than if a dense matrix was used. The sparse matrix has the class `lil_matrix` since with it two-dimensional indices can be used just like with a dense matrix. This property is important for the implementation of the matrix factorization algorithm that is described below. The dimensions of the matrix are defined by the maximum value of both the UserId and the ItemId. The reason for this is that the following approach is implemented: Each Id is represented by one row or column in the matrix. Thus, the first two columns of the rating data set are used as pointers/indices through which the matrix M can be accessed. If there are missing Ids in a sequence, then these rows remain empty and are compressed by the sparse matrix property.

For each UserId and ItemId the (true) rating is then entered into the respective cell. Since in the ratings data set the Ids start from the value 1 and in matrices indeces start from 0, 1 is subtracted from the Ids to make use of the 0^{th} row and column of the matrix.

In the second function, called `MF`, the matrix factorization algorithm is performed. As described in the background, first, two matrices are created and randomly initialized.

```
# Initialize user and item matrices with random values
UserMatrix = np.array(np.random.normal(scale=1/num_factor,
                        size = (num_users, num_factor)), dtype=np.float64)
ItemMatrix = np.array(np.random.normal(scale=1/num_factor,
                        size = (num_factor, num_item)), dtype=np.float64)
```

In this case, the dimensions of the matrices are impacted by the parameter `K`, here named **`num_factor`**, and the number of users/items in the data set.

```
for p in range(num_iter):
    for j,i in ratings_data[:,(0,1)]:

        pred = np.float64(np.dot(UserMatrix[j-1,:],ItemMatrix[:,i-1] ))
        rate = RatingsMatrix[i-1,j-1]
        err = rate - pred
        UserMatrix2 = UserMatrix[j-1,:] + np.float64(learning) * np.float64((np.float64(2.0)
                        * np.float64(err) * ItemMatrix[:, i-1]) - np.float64(regularization) * UserMatrix[j-1,:])
        ItemMatrix2 = ItemMatrix[:,i-1] + np.float64(learning) * np.float64((np.float64(2.0)
                        * np.float64(err)) * UserMatrix[j-1,:] - np.float64(regularization) * ItemMatrix[:, i-1])
        UserMatrix[j - 1, :] = UserMatrix2
        ItemMatrix[:, i - 1] = ItemMatrix2
    Prediction = np.transpose(np.dot(UserMatrix,ItemMatrix))
```

Second, the ratings for each user and item are predicted by performing a dot product operation using the respective j-1th row and i-1th column of the User- and Item-Matrices. These indices are fetched from the rating data. The first column contains the UserIds (j) while the second one contains the ItemIds (i). These Ids are then used as indices for the User- and Item-Matrices, respectively as described in the **`sparseMatrix`** section. The difference between the true rating and the predicted one is then added to each row/column of the User- and Item-Matrices, multiplied with the **`learning rate`** and **`regularization`** parameters. The value of the respective Item/User-Matrix is also (regularized) added to the current User/Item-Matrix. Here, it must be considered that the values that are added come from the previous iteration, before the adjustments above were performed. For this reason, a temporary variable is used to store the new value to prevent the values from growing out of bounds (value greater than double maximum boundary would result otherwise). The axes of the Prediction matrix are transposed so that the rows are the items and the columns are the users.

```
for j, i in ratings_data[:, (0, 1)]:
    rate = RatingsMatrix[i - 1, j - 1]
    predict = np.dot(UserMatrix[j-1,:],ItemMatrix[:,i-1] )
    if(predict > 5):
        predict = 5
    if (predict < 1):
        predict = 1
    errsum += math.pow(rate - predict, 2)
    maesum += math.fabs(rate - predict)
    counter += 1
rmse = math.sqrt(errsum/counter)
mae = maesum/counter
```

In the last part of the function, the RMSE and MAE of the training set predic-

tions are computed. To accomplish this, the indices of the users and items are used to access the sparse Rating matrix and compare its values to the predictions resulting from the dot product of the two matrices. First, the errors are added up and their (rooted - RMSE) mean calculated. The RMSE, MAE and Prediction matrix are then returned.

```
for fold in [3,4]:
    train_sel=np.array([x!=fold for x in seqs])
    test_sel=np.array([x==fold for x in seqs])
    train=ratings[train_sel]
    test=ratings[test_sel]
    rmseTrain[fold], maeTrain[fold], Prediction = MF(train, 10, 0.005, 0.05, 75)
    skip = 0
    for i in range(test.shape[0]):
        try:
            rmsetemp += (test[i,2] - Prediction[test[i,1]-1,test[i,0]-1])**2
            maetemp += math.fabs(test[i,2] - Prediction[test[i,1]-1,test[i,0]-1])
        except:
            skip += 1
    rmseTest[fold] = math.sqrt(rmsetemp/(test.shape[0] - skip))
    maeTest[fold] = maetemp/(test.shape[0]-skip)
```

At last, the 5-fold cross validation for the training and test sets are performed. First, the training model (Prediction Matrix) is calculated. This matrix is then used with the test set to gauge its accuracy since the test set has never influenced the training model. Like with the training data, the difference (RMSE, MAE) of the true ratings and predicted ones is computed. In case there are errors due to an index existing in the test set that does not in the training model (e.g. test index out of bounds of Prediction Matrix), this code block is surrounded by a try-except block. If an exception occurs, the program is not interrupted but the current index skipped without influencing the RMSE and MAE values. However, the counter needs to be decreased because of the skip. Thus, for each error, the `skip` variable is increased by one and later subtracted from the final count value used to calculate the mean. The training and error calculation is performed five times with the train and test sets changing each time. Finally, the results are printed to the console.

3.3.2 Results

The training and test accuracy of the naive approaches are as follows:

- The training and test accuracy (RMSE and MAE) of the Matrix Factorization approach are as follows.

	Train RMSE	Train MAE	Test RMSE	Test MAE
1	0.7699876	0.6040823	0.8722132	0.6830742
2	0.7697262	0.6040925	0.8716176	0.6826139
3	0.7681491	0.6013795	0.8739020	0.6836038
4	0.7695243	0.6036975	0.8724290	0.6827733
5	0.7695618	0.6039401	0.8724174	0.6834295

4 Discussion

From the result tables above, one can clearly see the difference in RMSE and MAE value between the algorithms. It indicates which algorithm is more relevant, has better accuracy in this task. We can split the implemented algorithms into two different groups. First, the three naive approaches, focused on finding the mean on the data set without learning from it, produce quick but relatively inaccurate predictions. The second group consists of the last naive approach and the MF, where predictions based on both the train and test data set are generated.

Since we have tried to use as few loops and nested loops as possible to improve the runtime and waste of resources, the performance is rather great:

- First naive approach: O(R) is needed to loop through all the ratings and find the mean. However, we only need to store the mean then the memory required is O(1)
- Second and third naive approach: similar to the first naive approach, a loop through all the ratings is necessary to group, add fall-back value and then calculate the mean. This gives us a cpu runtime of O(R+U) or O(R+M) which is equivalent to O(R). However, the memory required is difference, since we have one mean for every single user or movie, the memory required is O(U) or O(M).
- Fourth naive approach: Since we need both the mean vectors from the second and third naive approach, the cpu runtime and memory required will start at O(R) and O(U+M). To solve the linear expression, a loop through three vectors of size R is done to find the most fitted linear combinations. The runtime won't increase since it's still O(R) but the memory to O(U+M+R). This means the cpu runtime and memory required for this is both O(R).
- Matrix Factorization: The complexity of the MF algorithm depends on the number of ratings R and the parameters for iterations num_iterations and the parameter `K` in addition to U and M. Since the whole algorithm is repeated num_iterations times we multiply the time complexity with this parameter. For each rating, a vector dot product is calculated (complexity for a dot product with length n is O(n+n)), where the length of the vectors is `K`, thus leading to a complexity of O(2 * K * R) for the dot product calculation. Next, the Prediction Matrix is calculated which requires O(U* K *M) operations. At last the vector-dot-product operation described above is performed again to calculate the error which leads to a runtime complexity of O((2*2*K*R+U*K*M)* `num_iterations`). Depending on the data set the runtime complexity is thus O(R) or O(U*M) if R or the product of U and M, respectively, are larger than the other.
  The space complexity, on the other hand, is defined by the size of the User

and Item matrices since the two smaller matrices are kept in memory. They are also used to calculate the prediction matrix Also, the matrix which contains the true values of the ratings needs to be considered, which results in a complexity of O(U*K+M*K+U*M*U*2). For the current data set, the memory complexity is thus O(M*U).

When cross-referencing the results of our implementation to those found in MyMediaLite[1], there are small both positive and negative differences in accuracy of up to 0.03 for RMSE and MAE. There could be multiple reasons for this. First, we used a certain seed to ensure that our results can be reproduced which could have been different in their implementation. Second, rounding could have been applied at different stages which could have influenced the results. For example during the RMSE calculation, the predicted values are rounded so that they are kept in the interval between 1 and 5. Possibly, this is not done in other implementations. Third, the outliers that were found during the data exploration phase were not removed. It is possible that these outliers, which are represented by a few (user) rows with many column (item) values in the Prediction Matrix, influence the ratings of other users with less ratings (low or normal amount). Since each Item Matrix value is influenced by the corresponding User Matrix value, these outliers could have slightly influenced the results.

Comparing the accuracy of the approaches, one can see that there is a trade-off between accuracy and performance (runtime and memory requirements). As mentioned above, looking at the results, MF produces with the highest accuracy and is the best suited for predicting ratings. However, if the hardware or other resources are limited, the fourth naive approach should be chosen. The reason is that the difference in accuracy is rather small compared to the difference in runtime and memory requirement. Especially when the data set grows, the difference in resource requirements will also grow significantly for the matrix factorization.

References

[1] G. Takacs, I. Pilaszy, B. Nemeth, and D. Tikk, "On the gravity recommendation system," in *Proceedings of KDD cup and workshop*, vol. 2007, 2007.

[2] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[1] http://www.mymedialite.net/examples/datasets.html