Vassilios Galanos

Spyridon Mouselinos

Sofia Mpaltzi

Konstantinos Kyrkos

Thomas Saltos

Recommender Systems  
Team Project

Contents

[0. Introduction 2](#_Toc68687424)

[1. Part I: Reinforcement Learning Applications on Recommender Systems 3](#_Toc68687425)

[1.1. Data 3](#_Toc68687426)

[1.1.1. Preprocessing 3](#_Toc68687427)

[1.1.2. Session Clustering 3](#_Toc68687428)

[1.1.3. Train – Test Splits 4](#_Toc68687429)

[1.2. Theoretical Model 5](#_Toc68687430)

[1.2.1. Action, State and Reward Model 5](#_Toc68687431)

[1.2.2. Our Approach 5](#_Toc68687432)

[1.3. Evaluation Metrics 6](#_Toc68687433)

[1.3.1. Hit Ratio for top k recommendations 6](#_Toc68687434)

[1.3.2. NDCG for top k recommendations 6](#_Toc68687435)

[1.3.3. Engagement Metric 6](#_Toc68687436)

[1.4. Modeling 7](#_Toc68687437)

[1.4.1. Q-Learning 7](#_Toc68687438)

[1.4.2. Deep SARSA 10](#_Toc68687439)

[1.4.3. Deep Q-Learning 12](#_Toc68687440)

[1.5. Part I: Discussion 14](#_Toc68687441)

[2. Part II: Federated Learning Applications on Recommender Systems 15](#_Toc68687442)

[2.1. Data 15](#_Toc68687443)

[2.1.1. Preprocessing 15](#_Toc68687444)

[2.1.2. Train – Test Splits 15](#_Toc68687445)

[2.2. Main Problem 16](#_Toc68687446)

[2.2.1. Problem statement 16](#_Toc68687447)

[2.2.2. Metrics 16](#_Toc68687448)

[2.2.3. Full method 17](#_Toc68687449)

[2.2.4. Federated averaging 18](#_Toc68687450)

[2.2.5. Federated averaging on 100 clients 19](#_Toc68687451)

[3. References 21](#_Toc68687452)

# Introduction

The purpose of this document is to report the results for the Team Project of class Recommender Systems, February – April 2021. The objective is to create an engine for item recommendation. A recommendation engine or system is a subclass of information filtering system that seeks to predict items that are appealing recommendations to certain users. For the implementation of the recommendation system, we explore the applications of Reinforcement Learning and Federated Learning.

Reinforcement Learning is an area of machine learning concerned with how intelligent agents ought to take actions in an environment to maximize the notion of cumulative reward[1]. For the first part of this project, we define our problem using state, action, reward concepts and then attempt to solve it utilizing algorithms such as Q-Learning and SARSA, as well as deep learning architectures based on the idea of these algorithms.

Federated Learning (also known as **collaborative learning**) is a machine learning technique that trains an algorithm across multiple decentralized edge devices or servers holding local data samples. This approach stands in contrast to traditional centralized machine learning techniques where all the local datasets are uploaded to one server, as well as to more classical decentralized approaches which often assume that local data samples are identically distributed.

Federated learning enables multiple actors to build a common, robust machine learning model without sharing data, thus allowing to address critical issues such as data privacy, data security, data access rights and access to heterogeneous data. Its applications are spread over several industries including defense, telecommunications, IoT and pharmaceutics [2].

# Part I: Reinforcement Learning Applications on Recommender Systems

## Data

### Preprocessing

The data used are those of RecSys Challenge 2015. Dataset is session-based, and each session contains a sequence of clicks. These clicks may have led to purchases or not. For simplicity purposes, we utilize only the clicks sequences and attempt to give item recommendations appealing enough for the users to click, regardless of if they end up purchasing or not.

Original dataset consists of 9,249,729 sessions and 52,739 items. Since this approach would require approximately 11.125 GB for the Q table alone (Q-learning approach, Section 1.4.1), we reduced the dataset to the sessions that contained only one or more of the top 500 most popular (most clicked) items. Next, we filtered out sessions with less than 4 clicks and sessions with more than 10 clicks. That left us with 693,804 sessions and 500 items.

### Session Clustering

The dataset does not provide with any link between sessions and users. Therefore, we had to choose either to:

1. Assume that all sessions came from one user, or
2. Group the sessions into clusters of the same items.

The first option does not allow for any personalization and would lead to the creation of a system that recommends the most popular items to all users. The second option allows for more personalized recommendations but entails that we run the training process of our algorithm times equal to the number of clusters we choose to create.

Choosing to proceed with option (b), we created a TF-IDF encoding of each session based on both the unigram and bigram existence of items. In this way we wanted to encode both the sense of transition as well as the existence of the items themselves inside a session. Furthermore, we run Mini-Batch KMeans and tested for 10 to 20 clusters. The optimal number of clusters came up to be 13, as seen in Figure 1.

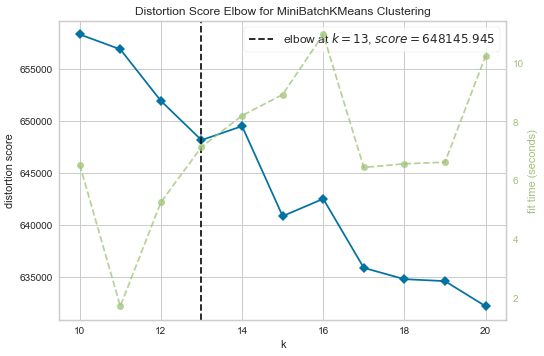


Figure 1: Elbow diagram

In Figure 2, we can see that clusters are for the most part well separated, and of similar size. For our experiments from now on, we use only cluster 7, but the process would be the same for all clusters.

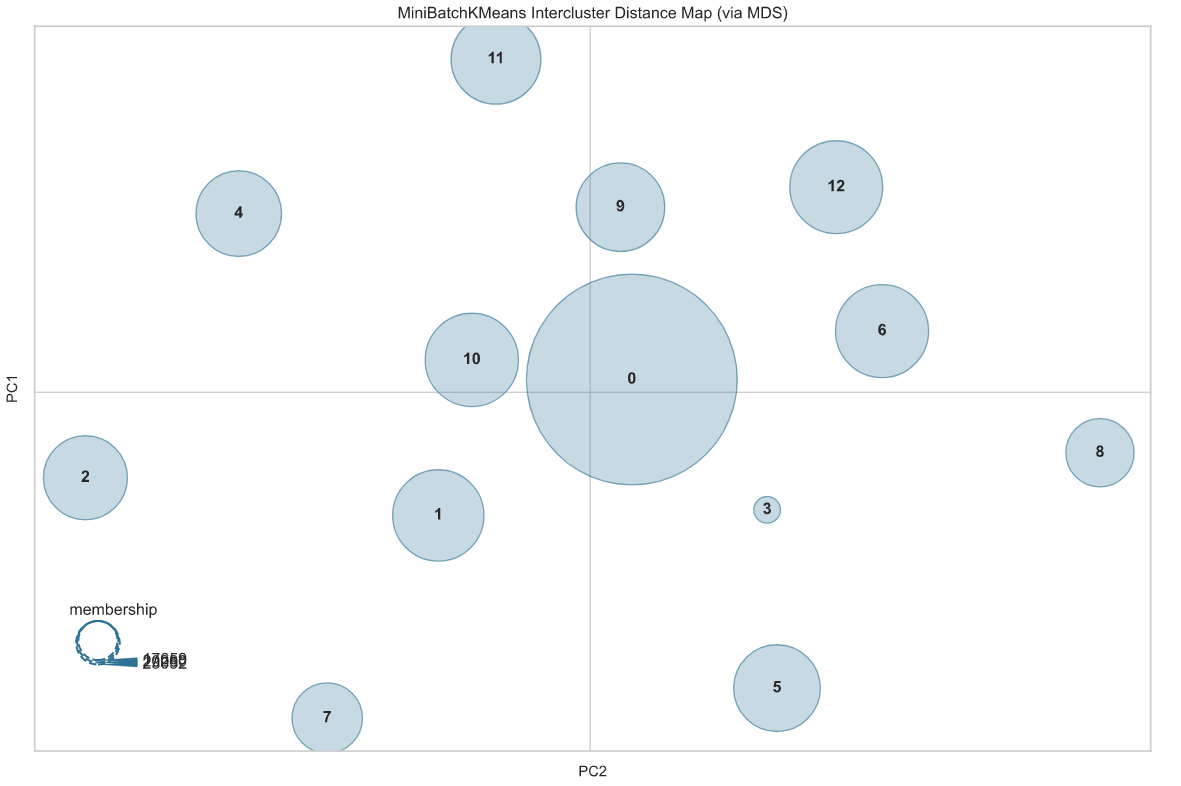


Figure 2: Cluster representation in two dimensions

### Train – Test Splits

The final part of the data manipulation process was to create our training and test sets. Cluster 7 consisted of 12,222 sessions and 295 different items. Using an 80 – 20 schema we ended up with 9,777 sessions for the training process and 2,445 for the testing.

## Theoretical Model

### Action, State and Reward Model

Let us define some basic concepts of Reinforcement Learning.

1. Agent: one that makes decisions in a predefined environment.
2. Environment: the universe of the agent – where the agent performs actions.
3. Action: a movement that takes the agent from one state to another.
4. State: the current situation of the agent.
5. Reward: when an agent performs an action in a state, receives a reward or a punishment (a negative reward).

Essentially, an agent must perform a series of actions, in a systematic manner, to learn the ideal way to go from state to state, by receiving guidance through rewards. The equation that expresses such scenario in mathematical terms is known as Bellman’s equation [3].

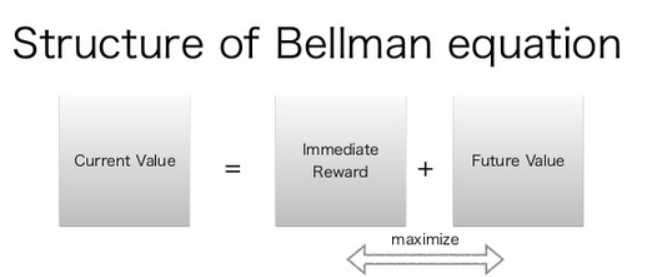


Figure 3: Structure of Bellman equation

### Our Approach

In our problem, we define as the state the current item that the agent is visiting, and therefore, there are 295 different states, one per item. As action, we define the act of clicking an item. In each state the agent can take one out of 295 different actions since it is possible to revisit the current item. In the training phase the agent is rewarded with +1 when choosing the right action and is punished with -1 when choosing any other action.

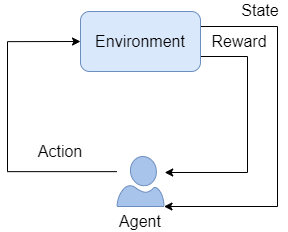


Figure 4: Action, State and Reward model

Ultimately, our objective is to create a system for next-item recommendations, meaning that given that a user clicked on a certain item we want to recommend a set of items including one that the user will click next. We train our algorithms to look only if the top recommended item was clicked but evaluate them using the top 5 and top 10 recommendations.

Our environment was created utilizing OpenAI Gym and is, effectively, a class inheriting from gym.Env. The following functions were implemented:

1. init(): defines the observation and action spaces, as well as other instance variables,
2. reset(): resets the environment by choosing the next session in each series of sessions and
3. step(): returns the reward of the current state and allows for the next step – next click in the current session to happen.

## Evaluation Metrics

### Hit Ratio for top k recommendations

The first metric we use to evaluate the performance of our algorithms is Hit Ratio for the top k recommendations (HR@k). HR@k measures whether the ground truth item is in the top-k positions of the recommendation list [4]:

In our experiments we calculate the metric for the top 5 recommendations and the top 10 recommendations for every click. Metrics lie in range 0-1.

### NDCG for top k recommendations

The second metric we use to evaluate the performance of our algorithms is Normalized Discounted Cumulative Gain for the top k recommendations (nDCG@k). NDCG is defined as the ratio of Discounted Cumulative Gain (DCG) to Ideal Discounted Cumulative Gain (IDCG)[5].

The DCG metric penalized erroneous ranking in a set of recommended items based on a relevance score.

However, in the case of Yes/No answers like recommendations of items this relevance score reduces from a scalar to a binary value. This simply means that .

The Ideal Discounted Cumulative Gain is simply the same score if the recommendations produced by the algorithm were in the correct ranking.

In this way the final metric is:

We calculate it for the top 5 recommendations and the top 10 recommendations for every click. Metrics lie in range 0-1.

### Engagement Metric

We define the engagement metric as the maximum correct subsequence between sessions. That is the maximum number of times the agent managed to recommend something in a row and kept the person engaged into browsing calculated per session. Our metric is a number that lies between 0 and the Maximum Session Length=10.

## Modeling

### Q-Learning

#### Overview

Q-learning is a model-free reinforcement learning algorithm. The algorithm is based on the state, action, reward model, and therefore, instructs an agent on what action to take on a certain state based on rewards. It does not require a model of the environment.

For any finite Markov decision process (FMDP), Q-learning finds an optimal policy by maximizing the expected value of the total reward over all successive steps, starting from the current state. "Q" names the function that the algorithm computes with the maximum expected rewards for an action taken in each state [6].

In the training process of the algorithm, we consider each session of the training set as an episode. Each episode consists of clicks that are represented as steps, meaning that each click of a user is a step in an episode. The algorithm visits each episode and for every step-click of the user attempts to make a recommendation. In our implementation, we take advantage of epsilon-greedy algorithm, in order to make random recommendations (explore options) in the beginning of the training process (we do not use epsilon-greedy in the testing phase at all). As episodes progress, the probability of exploring decreases and the algorithm recommends the item with the largest value in the Q table.

Below, there is an illustrative example of how Q-learning algorithm works. Supposedly, we begin with Item 2. Based on Q table, we recommend Item 1 and then Item 3.

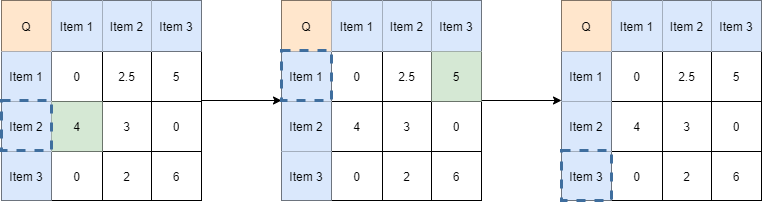
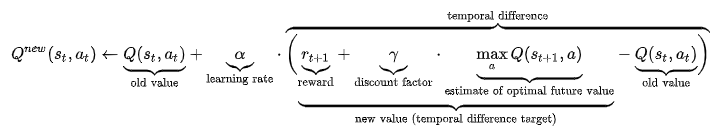


Figure5: Illustrative example of Q learning

In our experiments, we attempt to learn a table Q, of dimensions 295 items-states x 295 items-states. That table, given an item-state, dictates a policy that leads to successful recommendations of items. At the end of each episode, we update the q table based on the following rule:



#### Experiment 1: No Warm-up + Q-Learning

For our first experiment, we used all of the training set to learn the Q table, that we initialized with 0 values. For each correct recommendation, the algorithm was rewarded with +1 and for each wrong recommendation was punished with -1.

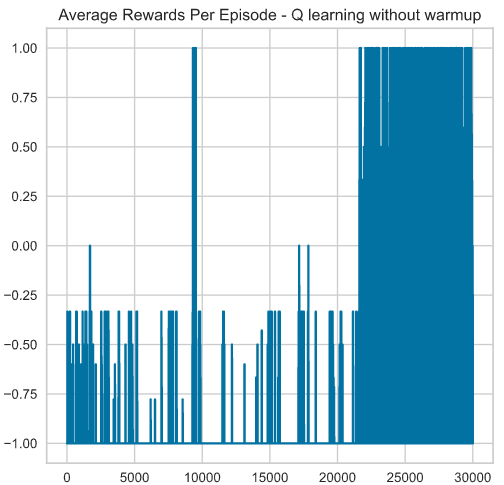
The following image shows the progress of the reward as the training process progresses. More specifically, the normalized reward per episode – total reward devided by number of clicks per episode. Note that we run for 30,000 episodes, meaning we pass the training set 3 times.

Figure 6: Reward progress in training set

We observe that in the beggining rewards are low, since the algorithm only makes wrong recommendations. Towards the end, rewards are significantly higher. We evaluate the algorithm using HR@5, HR@10, NDCG@5, NDCG@10 and Engagement metrics, reported on test set.

|  |  |
| --- | --- |
| Metric | Score |
| HR@5 | 0.06 |
| HR@10 | 80.43 |
| NDCG@5 | 0.04 |
| NDCG@10 | 11.66 |
| Engagement | 0.02 |

Observe that even though recommendations are poor in the top 5 recommendations, they are much better in the top 10. HR@10 metric shows that 80.43 % of the times that we recommend 10 items to the user, the user ends up on clicking one of those items. Engagement is only 0.02, but that is mainly due to the fact that it is a very strict metric.

#### Experiment 2: Warm-up on 70% + Q-Learning

For our second experiment, we used 70% of the training set to warm-up the Q table, that we initialized with bigram frequencies (item i to item j). The rest of the training set was used as in Experiment 1. For each correct recommendation, the algorithm was rewarded with +1 and for each wrong recommendation was punished with -1.

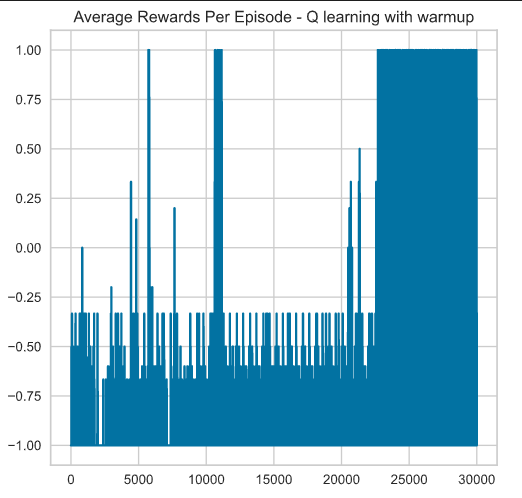
The following image shows the progress of the reward as the training process progresses. More specifically, the normalized reward per episode – total reward dvided by number of clicks per episode. Note that we run for 30,000 episodes, meaning we pass the training set 3 times.

Figure 7: Reward progress in training set

We observe that in the beggining rewards continue to be low, but better than in Experiment 1. Towards the end, rewards are significantly higher again. We evaluate the algorithm using HR@5, HR@10, NDCG@5, NDCG@10 and Engagement metrics, reported on test set.

|  |  |
| --- | --- |
| Metric | Score |
| HR@5 | 1.72 |
| HR@10 | 80.46 |
| NDCG@5 | 1.08 |
| NDCG@10 | 12.55 |
| Engagement | 0.02 |

The top 10 metrics have not changed significantly, but we have managed to get better scores in our top 5 recommendations. Engagement is low in this case too.

### Deep SARSA

#### Overview

State – Action – Reward – State – Action (SARSA) algorithm is an on-policy temporal difference (TD) control method. TD updates the knowledge of the agent on every action rather than on every episode. As mentioned, Q-Learning uses off-policy technique and a greedy approach to learn the Q-value. SARSA, on the other hand, is an on-policy algorithm that uses the action performed by the current policy to learn the Q-value [7].

As with Q-learning, in our experiments, we attempt to learn a table Q, of dimensions 295 items-states x 295 items-states. That table, given an item-state, dictates a policy that leads to successful recommendations of items.

Q-table update depends on the current state, current action, reward obtained, next state and next action, and is performed using the following rule:



Deep SARSA is an implementation of SARSA algorithm that uses Neural Networks. Even though it is a deep method, it does not require external memory (experience replay). It also works better using Boltzmann policy instead of epsilon-greedy. Finally, since it is an on-policy method we do not need 2 networks for the implementation, as with off-policy methods.

For the model implementation, we modified the original environment so it can provide a triplet of historic states as the input to our model. In this fashion, the model will have to consider the temporal dimension of previous clicks to differentiate as to what the next correct suggestion is.

Furthermore, each state is encoded into a 128-dimensional embedding and the 3 historic state embeddings are passed into an LSTM to create the final 128-dimensional embedding that has captured the causality through time. This final embedding is later passed through 2 fully connected layers and finally guess the next correct item / action to suggest.

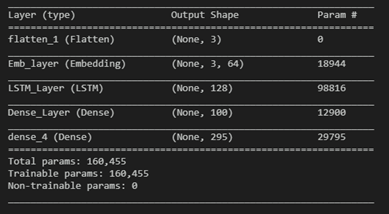


Figure 8: Deep SARSA model summary

#### Experiment: Deep SARSA

The following image shows the progress of the reward as the training process progresses. In this case, we do not report the normalized reward, but the total reward.

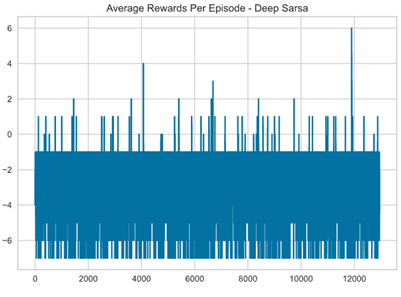


Figure 9: Reward progress in training set

As it can be observed, rewards are low throughout the experiment, with only few positive values.

|  |  |
| --- | --- |
| Metric | Score |
| HR@5 | 0.1 |
| HR@10 | 9.6 |
| NDCG@5 | 0.05 |
| NDCG@10 | 2.31 |
| Engagement | 0.0 |

All our metrics get very low values, even lower that the non-deep methods.

### Deep Q-Learning

#### Overview

In this section we created a Deep Q Learning solution. For this reason, we modified the original environment so it can provide a triplet of historic states as the input to our model. In this fashion, the model will have to consider the temporal dimension of previous clicks in order to differentiate as to what the next correct suggestion is. Furthermore, each state is encoded into a 128-dimensional embedding and the 3 historic state embeddings are passed into an LSTM in order to create the final 128-dimensional embedding that has captured the causality through time. This final embedding is later passed through 2 fully connected layers and finally guess the next correct item / action to suggest.

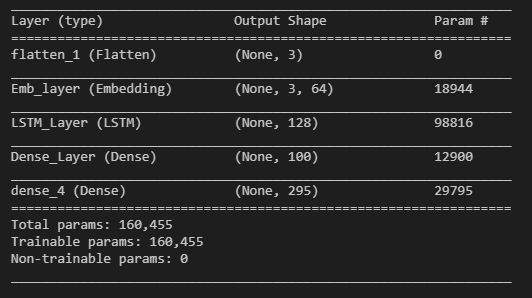


Figure 10: Deep Q-Learning model summary

We face once again the cold-start problem, now in the sense of the Item Embedding Matrix. That matrix would be randomly initialized and lead to poor early training performance. For this reason, before training the Deep Q Agent Model we create a copy of it and pretrain it in a classification objective. Inspired by the Word2Vec [8] approach as well as the MLM loss of the Bert architecture [9] we simply guess from a time window of 3 items what would the next be. That pretraining loss is named Next Word Prediction and we denote it by NWP. Note that we do not optimize aggressively since we are not interested of maximizing purely the next prediction rather than the sequence of next items. After pretraining for 10 epochs we transfer the weights of the model to our Deep Q Agent that theoretically is now capable of correctly guessing the next item in a sequence at around 78% (train set performance).

In order to keep experiments fair, the Deep Q Agent is parametrized with settings similar to the previous approaches. It uses a constant E-Greedy Policy of 10%, and gamma of 0.5.

We utilize Experience Replay with a Memory of 10,000 episodes and set the target network policy equal to the agent network policy after every 500 episodes. The agent uses the first 200 episodes as a warmup session to fill in the Memory.

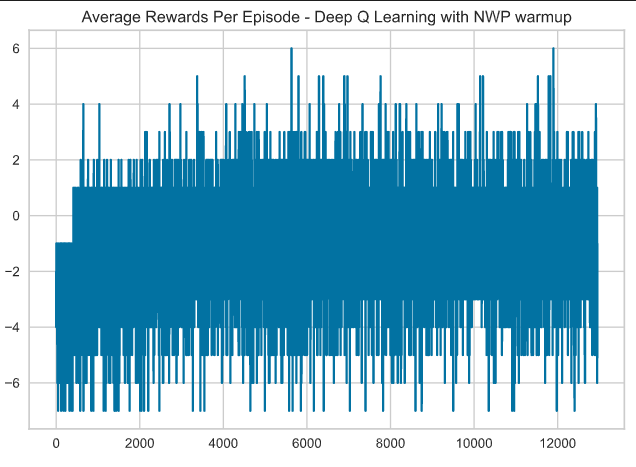


Figure 11: DQN model and training parameters

#### Experiment: DQN with pretrained embeddings

The following image shows the progress of the reward as the training process progresses. In this case, we do not report the normalized reward, but the total reward.

Figure 12: Reward progress in training set



Rewards start low but quickly begin to fluctuate around 0. We evaluate the algorithm using HR@5, HR@10, NDCG@5, NDCG@10 and Engagement metrics, reported on test set.

|  |  |
| --- | --- |
| Metric | Score |
| HR@5 | 38.24 |
| HR@10 | 42.27 |
| NDCG@5 | 37.92 |
| NDCG@10 | 38.93 |
| Engagement | 0.32 |

Interestingly enough, even though we get high scores for our top 5 recommendations, they do not get much better for our top 10 recommendations. Engagement is significantly improved compared to the rest of the experiments.

## Part I: Discussion

In the following table we present all the metrics for all the algorithms.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Metric | Q-Learning  Exp. 1 | Q-Learning Exp. 2 | Deep SARSA | DQN |
| HR@5 | 0.06 | 1.72 | 0.1 | 38.24 |
| HR@10 | 80.43 | 80.46 | 9.6 | 42.27 |
| NDCG@5 | 0.04 | 1.08 | 0.05 | 37.92 |
| NDCG@10 | 11.66 | 12.55 | 2.31 | 38.93 |
| Engagement | 0.02 | 0.02 | 0.0 | 0.32 |

As expected, our list of top 10 recommendations is always better than our list of top 5 recommendations. That is also apparent in NDCG metrics that get lower values, meaning that when a correct recommendation occurs it is not placed on top of the list. We also observed that using a portion of the dataset as a warmup can boost the algorithms into making better recommendations.

In the case of Q-learning, for both of our experiments about 80 % of the times that we recommend 10 items to the user, the user ends up on clicking one of those items. That is the best score that we manage to obtain in all methods for this particular metric.

In the case of Deep Q-Learning, we manage to get significantly higher hit ratio for our top 5 list of recommendations than the other three methods. We would expect that the corresponding metric for the top 10 list would be much better, but that is not the case since the two metrics get almost equal values. That means that when making a correct recommendation it is almost always in the top 5. What is more interesting is that NDCG@5 and NDCG@10 metrics also get similar values to the hit ratio ones. Consecutively to our previous observation, having high NDCG scores means that when a correct recommendation occurs it is almost always on the top of the list. Therefore, when using Deep Q-Learning algorithm, if we are to make a correct recommendation it will be on the top of the list and if we are not, it will not even be in the top 10 list. The engagement metric is also relatively higher for this method, meaning that we manage to make repeatedly good recommendations.

Among all methods, Deep SARSA performs the worst in all metrics. SARSA learns a near-optimal policy whilst exploring, while Q-Learning directly learns the optimal policy that is our goal. To use SARSA for optimal policy learning we would probably need a better strategy when choosing hyperparameters.

# Part II: Federated Learning Applications on Recommender Systems

## Data

### Preprocessing

The data used are those of the Book-Crossing dataset and can be found [here](http://www2.informatik.uni-freiburg.de/~cziegler/BX/). The dataset contains 3 tables. BX-Users which contains the users, BX-Books which contains the Books identified by their respective ISBN. Invalid ISBNs have already been removed from the dataset. Moreover, some content-based information is given (`Book-Title`, `Book-Author`, `Year-Of-Publication`, `Publisher`), obtained from Amazon Web Services. Finally, BX-Book-Rating is the most important of the tables and contains the book rating information. Ratings (`Book-Rating`) are either explicit, expressed on a scale from 1-10 (higher values denoting higher appreciation), or implicit, expressed by 0.

Dataset consists of 271,360 distinct books, 278,858 distinct users and a total of 1,149,780 book ratings. We scale our reviews from 0 to 1. To better understand the data we will work, we present the following histogram where we can see the distribution of the reviews.

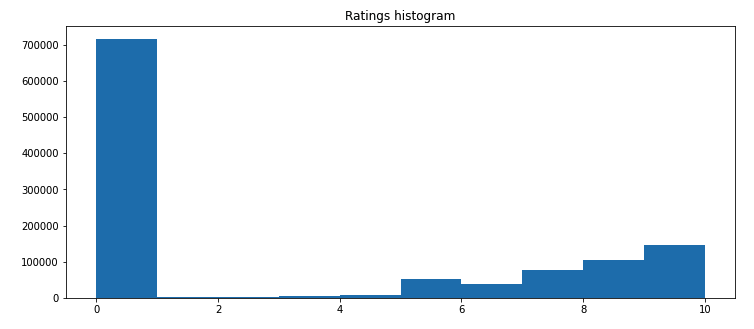


Figure 13: Ratings histogram

As we can see, more than half the reviews of our dataset are zeros. Interesting is also the fact that there are almost no reviews between 1 and 5. This is partially expected as people usually rate something if its particularly good or particularly bad. That is also why we can see an increasing number of reviews for each of following numbers after 5.

### Train – Test Splits

The final part of the data manipulation process was to create our training and test sets. We decided to split our dataset keeping the 70% of it for training and 30% for testing. This leaves us with 804,846 ratings for training and for 344,934 for testing.

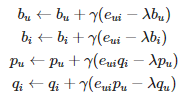
## Main Problem

### Problem statement

We want to factorize the Ratings matrix as A = P \* Q. Where P is the user matrix of dimension m \* k and Q is the item (book) matrix of dimension k \* n. Stating that we want to estimate the unknown parameters by minimizing the following sum.

Where:

*  is the predicted rating of user u to the item i.
*  is the rating of user u to the item i.
*  the user biases.
*  the item biases.
*  the user factors.
*  the item factors.
* λ the regularization hyperparameter term.
* μ is the global mean rating.

We perform the minimization, using stochastic gradient descent as advised by the assignment.

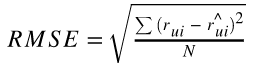
where the error is:

and the prediction is:

### Metrics

The metrics we used in any case are the following:

* *Rooted Mean Squared Error*. The [standard deviation](https://www.statisticshowto.com/probability-and-statistics/standard-deviation/) of the [residuals](https://www.statisticshowto.com/residual/) ([prediction errors](https://www.statisticshowto.com/prediction-error-definition/)). Residuals are a measure of how far from the regression line data points are; RMSE is a measure of how spread out these residuals are.



* *Mean squared error*. Measures the [average](https://en.wikipedia.org/wiki/Expected_value) of the squares of the [errors](https://en.wikipedia.org/wiki/Error_(statistics)) that is, the average squared difference between the estimated values and the actual value.



* *Accuracy*. The number of correctly prediction out of all predictions. More formally, it is defined as the number of true positives and true negatives divided by the number of true positives, true negatives, false positives and false negatives.



### Full method

We start as advised by the assignment to solve the optimization problem using the entire dataset in a centralized way. More information about the approach that we used can be found [here](https://surprise.readthedocs.io/en/stable/matrix_factorization.html). We train our model for 20 epochs, using 0.01 for our regularization parameter to avoid overfitting, 1e-1 for our learning rate parameter and as advised we use a small number of latent factors e.g. 4 for items and for users. We must note here that we don’t use any decay which would reduce our learning rate as the training progresses, but we have implemented the functionality to do so.

Below, we present the results from the execution of the method, using the parameters that we mentioned.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *ROUNDS* | 0 | 1 | 2 | ... | 14 | 15 | 16 | 17 | 18 | 19 |
| *RMSE* | 0.313 | 0.295 | 0.283 | ... | 0.201 | 0.197 | 0.194 | 0.192 | 0.190 | 0.187 |

We can see here, that with each epoch we seem to gradually reduce the RMSE. These can also be seen in the following figure, where we visualize this reduction. We can also see that our metric does not stop reducing which could be a sign that with more training epochs our model could achieve better results.

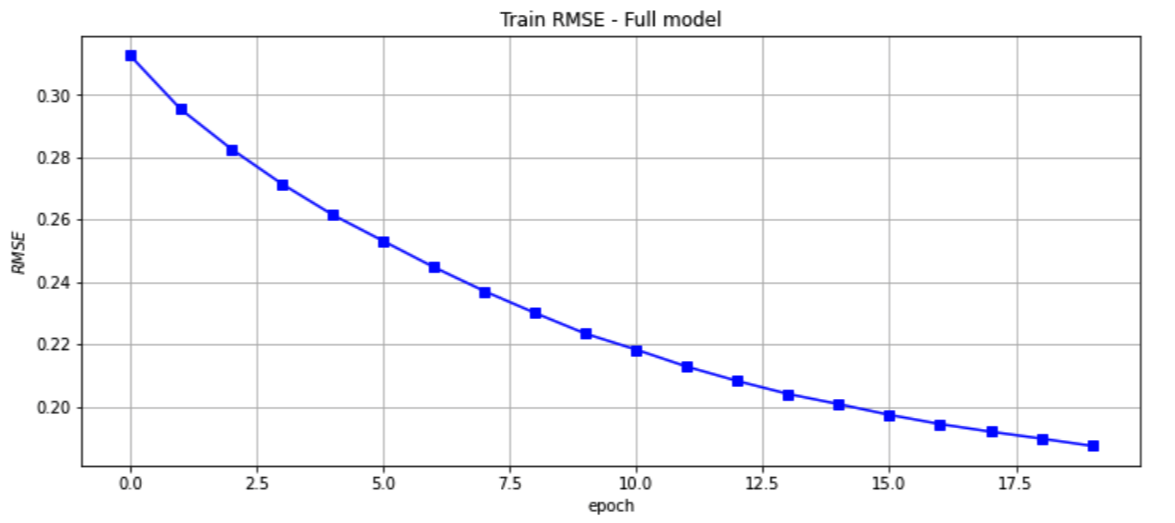


Figure 14: Train RMSE – Full model

Below, we also present the results of our method in the aforementioned metrics.

|  |  |  |
| --- | --- | --- |
| Metric | Train results | Test results |
| RMSE | 0.16 | 0.39 |
| MSE | 0.04 | 0.15 |
| Accuracy | 0.46 | 0.22 |

We can see that we obviously performed better on our training set, but our test set results are not that far away, which means that we have avoided overfitting sufficiently.

In order to validate our results, we have conducted an experiment where we predict the rating of a user for a specific book from our training set and then we cross-checked it with the true rating. Our method predicted 0.94 for an item whose original rating was 10. This experiment validated our metrics showing that our method works well.

At this point we present the top 10 predictions for a user under consideration. We believe that presenting these results validates the method usefulness and value. Along with our top 10 recommendations, we also show the users’ top 10 ratings on book. We can see that there is no clear correlation of the topics and types of books that are recommended to the user, but they are not that irrelevant to be considered random.

Top 10 recommendations for user:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *ISBN* | *Book-Title* | *Book-Author* | *Publisher* | *Year-Of-Publication* |
| *44521* | 1570827893 | Disney's Pooh's 123 (Learn and Grow.) | Lisa Ann Marsoli | 1998 |
| *140549* | 068982047X | The Bestest Mom (Rugrats) | Susan Hood | 1998 |
| *172880* | 0803281781 | Little Britches: Father and I Were Ranchers | Ralph Moody | 1991 |
| *199879* | 0762102977 | Yoga for Everybody/Not Everybody: Simple Routi... | Paul Harvey | 2001 |
| *200440* | 0425091465 | Inside the Green Berets | Charles M. Simpson | 1986 |
| *200969* | 0446949280 | Lots of Funny Riddles | Joseph Kiernan | 1981 |

### Federated averaging

For this part we utilized the method we created from the previous part. In original federated averaging method, the algorithm would be run on each client at the same time in different machines. This original approach surely exceeded the purposes of the assignment as this would need various machines of different specifications to run at the same time. To tackle this, we decided to run our method for each of the users one at a time. This of course is a time costly process due to the number of the users and the number of calculations being conducted on matrices. To that purpose we decided to use 3 training epochs for each user in a repetition of 10 rounds. This would result in a fair amount of training time for each users' weights. As advised by the method statement, in each round we take the users set, we update our local weights for each user and after all user data has been processed, we update the global weights.

We can see our method results on the RMSE metric on each round. As we can see the error seems to drop but with a too slow pace. This is logical as the process is slow, taking in account each time the weights changed by each user and not all the at once. As stated above, this process would make sense running a several machines. Along with that comment we can add that this process would make sense if it ran for a long period of time on each machine in order to achieve better performance.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *ROUNDS* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| *RMSE* | 0.386 | 0.386 | 0.386 | 0.385 | 0.385 | 0.385 | 0.385 | 0.385 | 0.385 | 0.384 |

More on the training and the drop of the RMSE we can see in the following figure.

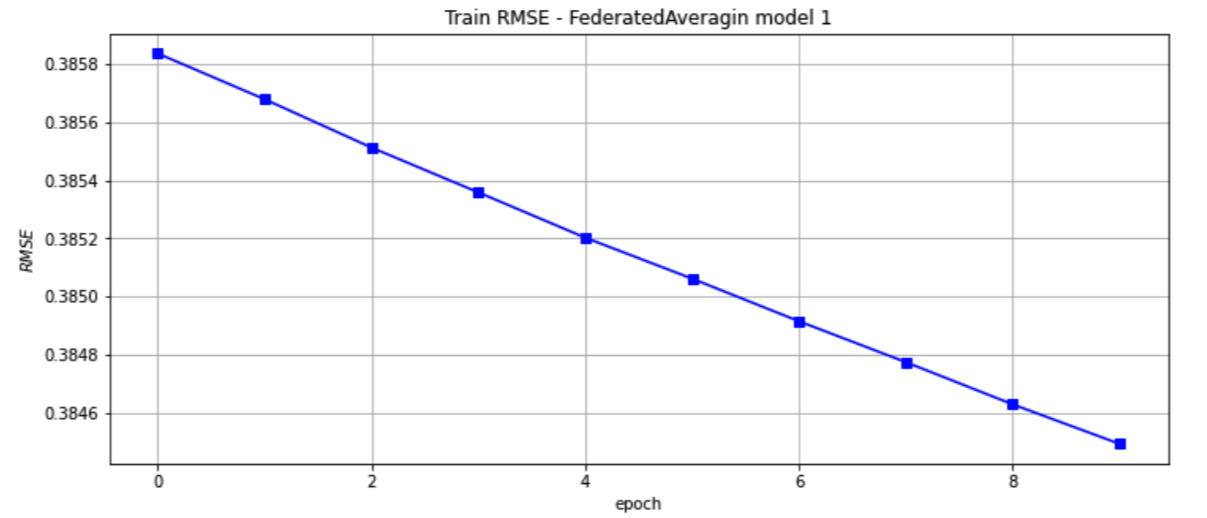


Figure 15: Train RMSE – Federated Averaging model 1

As we can see, the RMSE drop as stated above with each epoch. From the figure the drop seems to be great, but the difference between the first and last epoch is really small. In order to better judge the methods performance, we would need to train the model for more time and validate whether it would improve its results or if they would stay the same.

Below we present the train and test results of our method for the metrics we decided to use.

|  |  |  |
| --- | --- | --- |
| Metric | Train results | Test results |
| RMSE | 0.38 | 0.38 |
| MSE | 0.15 | 0.15 |
| Accuracy | 0.01 | 0.01 |

We can see that our train and test results match. This means that on the bright side we have avoided overfitting, but our model has not managed to get good results on either set.

### Federated averaging on 100 clients

For this part we decided to use another approach on the Federated averaging method. This time instead of running the same process for each of the users, providing them only 3 epochs, we decided to use only 100 of the users and provide them with more epochs. We increase the number of epochs to 5 for each user and the number of total rounds to 80 instead of 10. This means that we use less users but give them more time to provide the model with better updates. Practically this is the middle ground between the previous method where we run the process on each user and the first method where we didn’t split our ratings to each user.

We present again the RMSE results on each round.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Round* | 0 | 1 | 2 | ... | 76 | 77 | 78 | 79 |
| *RMSE* | 0.356 | 0.349 | 0.344 | ... | 0.287 | 0.287 | 0.287 | 0.286 |

We can see that this time providing our method more time and reducing the number of clients, we have managed to significantly reduce the RMSE after 80 rounds. This can also be seen in the following figure.

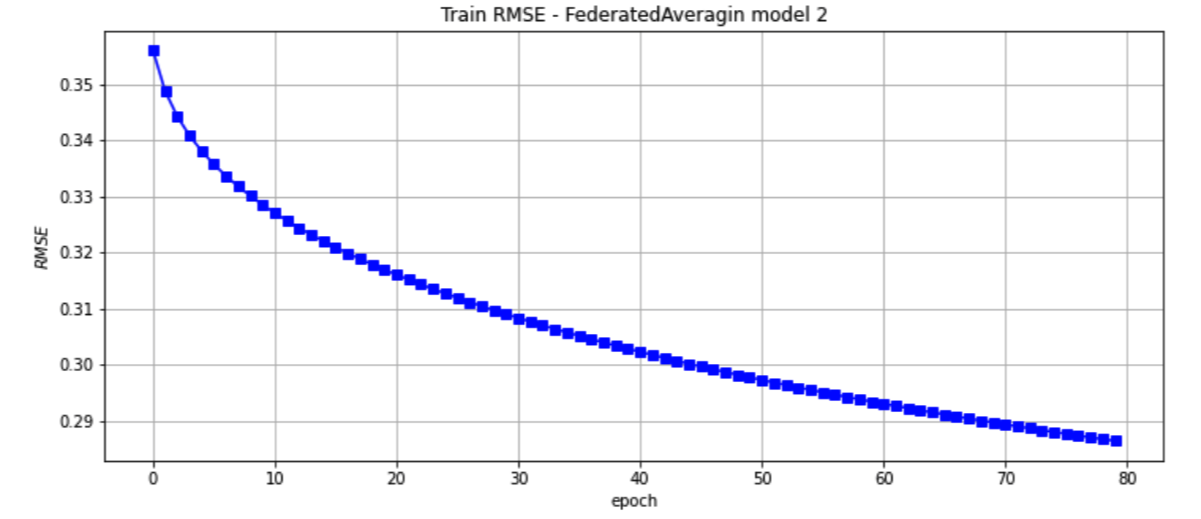


Figure 16: Train RMSE – Federated Averaging model 2

From the figure above we can see how the RMSE drops with each round. We can also see that the metric has reduced the rate with which it drops but has not yet stopped reducing in value which means that it could be further reduced.

Next, we present our metrics results on the train and test sets.

|  |  |  |
| --- | --- | --- |
| Metric | Train results | Test results |
| RMSE | 0.29 | 0.34 |
| MSE | 0.08 | 0.12 |
| Accuracy | 0.13 | 0.11 |

We can see that the results we have achieved this time are far better than last time in every metric we used.

# References

[1] J. Hu, H. Niu, J. Carrasco, B. Lennox and F. Arvin, "Voronoi-Based Multi-Robot Autonomous Exploration in Unknown Environments via Deep Reinforcement Learning," in IEEE Transactions on Vehicular Technology, vol. 69, no. 12, pp. 14413-14423, Dec. 2020, doi: 10.1109/TVT.2020.3034800.

[2] Wikipedia: Federated Learning: https://en.wikipedia.org/wiki/Federated\_learning

[3] Introduction: Reinforcement Learning with OpenAI Gym, Ashish Rana, Sep 2018 <https://towardsdatascience.com/reinforcement-learning-with-openaid445c2c687d2>

[4] Xin, X., Karatzoglou, A., Arapakis, I., & Jose, J. (2020). Self-Supervised Reinforcement Learning for Recommender Systems. Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval.

[5] B. Croft; D. Metzler; T. Strohman (2010). Search Engines: Information Retrieval in Practice. Addison Wesley.

[6] Strehl, Li, Wiewiora, Langford, Littman (2006). PAC model-free reinforcement learning

[7] Reinforcement learning: Temporal-Difference, SARSA, Q-Learning & Expected SARSA in python

<https://towardsdatascience.com/reinforcement-learning-temporal-difference-sarsa-q-learning-expected-sarsa-on-python-9fecfda7467e>

[8] Word2Vec: Distributed Representations of Words and Phrases and their Compositionality, Mikolov 2013.

[9] Bert: Pre-training of Deep Bidirectional Transformers for Language Understanding, Devlin J & MW Chang 2018.