Introduction

In recent decades, online platforms such as Youtube, LinkedIn, Google, and other web services have been on the rise. This has led to the development of recommender systems on many platforms. In fact, it is now impossible to avoid these systems in our day-to-day lives on the internet, as recommendation systems range from matching user preferences to advertisements to suggesting potential items of interest to buyers on the e-commerce market. Recommendation systems are algorithms that provide suggestions, or recommendations, of relevant items to users - this can range from products to buy to movies to watch to people to follow on social media, depending on the industry. They aim to discover valuable information or patterns without explicit input from users. An efficient and good system will not recommend items that are too similar to things users have seen before - recommendations should be diversified. There is more of an emphasis on personalization, thus making it more vulnerable to data sparsity. Examples of recommendation systems include Netflix's "Other Movies You May Enjoy" feature, Amazon's "Customers Who Bought This Item Also Bought" feature, "Facebook's "People You May Know" feature, and many others. From the business perspective, these systems are critical in certain industries because they have the ability to generate a significant amount of income if they are efficient, and also allows the business to differentiate themselves from competitors. The purpose of this project is to develop our own version of a recommendation system using the data given. The problem we will be addressing is to determine a user's rating on a movie by figuring out how similar a user is to other users and how similar a movie is to other movies. Thus, our aim is to analyze all ratings, then determine some user X's rating on some movie Y, by looking at the similarity between users and movies. The questions we will be tackling include the following: How do we preprocess the data? How do we determine how similar users are to other users, and how similar movies are to other movies? What are the end results, and could it be done better? In this paper, we start with related works to explain how baseline methods differ

from our method. Next, we describe our solution/method on what we did as well as the tools and techniques used. After, in data and experiments, we specify what preprocessing was done to the data. Lastly, we report on the evaluation and results of our method.

Related Work

In the beginning, we found that there were multiple ways of going about this problem. In our case, we solely used users' ratings on movies in order to create our model and predictions. In larger corporations, this method factors in a user's preferred genres and also has the movie genres tagged in the set as well. It has often been said that a larger amount of high-quality data provides more value and insights than a model with lesser data but more complex algorithms. One such method would be Latent Factor Models, which was the winner of Netflix's movie recommender challenge. Please refer to *Image A* in the appendix - there are two thresholds that this model factors in. One being gender preferences, which is understandable since men and women often have different and polarizing interests. The other one measures how serious a movie might actually be, from funny to serious. Through the use of this map, we can pinpoint more specific recommendations. We could also add another dimension to the model, having three thresholds that judge movie tastes. This new addition, however, would make the model much more complex.

Solution/Method

Our solution implements the Matrix Factorization Method. We rely on the assumption that users that rated movies in a similar way will also have similar preferences. In other words, if someone likes Shrek 3, The Matrix, and Star Wars, we assume there exists another individual that has the same tastes. Therefore, we can compare each individual and place them in groups to determine a predicted rating per group. As seen in *Image B*, the matrix is sparsely distributed - there are lots of missing values. The strength of our method is that it uses a form of Collaborative Filtering, which will take advantage of the network effect of having multiple data points (individuals rating movies). Please refer to *Image C* - the following diagram represents a visual representation of how collaborative filtering works. In the first slide, we have a matrix of user preferences for different objects. With this, our goal is to figure out

whether the last user likes videos. The solution is to find users that have similar likes to the last user and determine the most likely rating. This same idea applies to our situation, where users rate different movies. The method we chose follows a general data mining methodology. First, we split our original dataset into two, leaving an 80/20 training/testing split. We also ensure that userIds and their corresponding movieId ratings can be treated as contiguous values in order to apply our Matrix Factorization model. Please refer to *Image D*. The resulting matrix encodes userId and movieId as categorical values. The matrix is our training dataset. We then create a model using pytorch libraries which is described next in steps. Please refer to the code in *Image E* in the appendix. Initially, we are setting random weights for each item in the matrix. These weights are a uniform distribution ranging between 0 and 0.05. We also create an embedding matrix for the userIds and the movieIds. This allows us to quickly lookup rows in the matrix and save on memory as well as execution time, since we are factoring out the need to perform matrix multiplication. The final step in our model is defining the forward function. This function essentially takes a dot product of embedding matrices to return the estimated rating of each individual user to movie mapping. This can be seen in *Image F* in the appendix. The next step in our method is to use this model to train a network by modifying the weights of each individual item in the embedding matrices. As seen in *Image G* in the appendix, we create four matrix factorization models to perform experiments on different optimization methods. The function in our implementation trains the models. The inputs are MFmodel, which is one of the models we created, optimizer name, which defines the optimizer we will be using for the current call, epochs, the number of passes to make over the dataset, learning rate, and weight decay. Please see *Image H*. Pytorch has many built-in methods to train our network, and the training is initialized by MFmodel.train(). Each iteration in the for loop defines a complete pass over the training set. We start off by creating user and item embeddings, getting their ratings, and creating a tensor vector of actual rating values from the original training set. We then create a prediction, y, to use the matrix factorization method and dot product to return a tensor vector of predicted rating values from the forward function. Next, we take a step in the direction of the local optimum (lowest error rate) and the for loop executes once again. The RMSE is

printed to the console and after the for loop finishes, we call EvaluateLoss to find the overall RMSE of the trained model. Finally, we need a way to evaluate the results. This is where our 20% testing dataset comes into play. As seen in *Image I* of the appendix, we made a call to the function. This function creates a new Matrix Factorization model based on the input model to our training function described previously. It does this by putting the model into evaluation mode, then comparing the results of the input and output models. The result is the root mean squared error. Now that we have defined our solution and the methods we used, it is time to show more about the provided datasets and the experiments.

Data and Experiments

We started off by visualizing the data to better understand what kind of patterns and properties we are dealing with. Our first step was to visualize the distribution of user rating, which can be seen in *Image N*. It is clear that the distribution is skewed left, with most of the user ratings being above 3. The second experiment was to visualize the distribution of how many ratings a single user has. The result is obvious, most of the users have a low number of ratings, which corresponds to the real world scenario. There exist some outliers with more than 500 ratings, which is surprising & likely represents movie critics or lovers. This again points to the idea of a sparse matrix, which is where Collaborative Matrix Factorization truly shines. Our final experiment on the data was to visualize the distribution of the number of ratings a single movie has. Once again, most of the movies have a low number of ratings, with a few outliers having over 100 ratings.

We chose not to remove any data from our experiment because our goal is to keep the dataset as close to the original as possible. The only preprocessing we performed was categorical encoding of userIds and movieIds for compatibility with pytorch libraries. In a real world scenario, it is likely that users with 0 ratings are not considered at all, but services like Netflix actually encourage new users to rate 3 movies. They also have access to user demographic information, as well as third party data collected and sold by Facebook.

Evaluation and Results

We use different gradient descent methods in order to test our cases. One of these is Adams, this is an adaptive learning rate optimization algorithm that's been designed specifically for training deep neural networks. Another one was Stochastic Gradient Descent (SGD), which is an iterative method for optimizing an objective function with suitable smoothness properties. We also used the Average Statistic Gradient Method. Lastly, we have Adadelta, which is a more robust extension of Adagrad that adapts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. We ran and tested multiple parameters for each MFmodel, to see what the best parameters are and visualize their performance on a 2D chart. This is an attempt to tune each model to optimal conditions. Please see *Image J* for the experiments on optimal number of epochs, as well as *Image K* for the learning rate. It seems that Adams algorithm is the only choice that truly sees a change in RMSE from tuning the number of epochs and the learning rate. SGD, ASGD, and Adadelta do not see a big improvement from changing these parameters. We started with 1 epoch all the way to 20. The best choice for Adams was 8 epochs. After that the RMSE peaked at a local low and began increasing. The best optimization method for us was the Adams method with an optimal RMSE of 0.881! This is actually a great result which proves the effectiveness of collaborative filtering. This was the case of an epoch number of 8 and a learning rate of 0.05, the optimal learning rate for Adams. The Adams method implicitly performs coordinate-wise gradient clipping and can thus, unlike SGD, tackle heavy-tailed noise. We prove that using such coordinate-wise clipping thresholds can be significantly more effective than using a single global one. Additionally, we can see that the root mean squared error is low for the Adams case, which means that the average prediction is pretty close to the actual value (as shown in the figure). Please refer to *Image L* for the optimal learning rates. We then tune the final parameter, weight decay. We will be using an epoch value of 8 and a learning rate of 0.05. The weight decay variable will change by increments of 0.05 from 0 to 1. Please refer to *Image M* - this showed us that weight decay of 0 was the most optimal for our experiments. Please refer to *Image Q*, *Image R*, and *Image S* to see the results of all our experiments aggregated.

Appendices

Image A

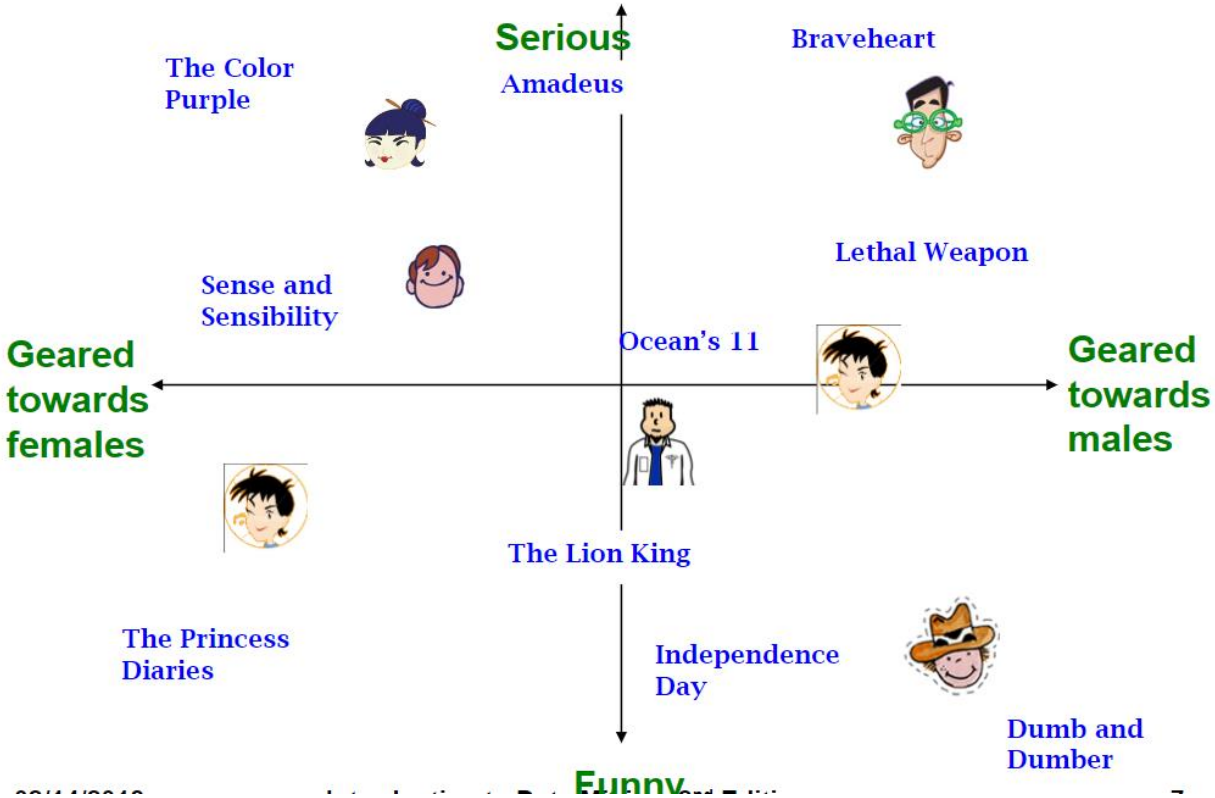

Image B - Visualizing The Sparse Matrix

movieId	1	2	3	4	5	6	7	8	9	10	...	193565	193567	193571	193573	193579	193581	193583	193585	193587
userId																				
1	4.0	NaN	4.0	NaN	NaN	4.0	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
606	2.5	NaN	NaN	NaN	NaN	NaN	2.5	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
607	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
608	2.5	2.0	2.0	NaN	NaN	NaN	NaN	NaN	NaN	4.0	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
609	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	4.0	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
610	5.0	NaN	NaN	NaN	NaN	5.0	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Image C - Example of Collaborative Filtering

Image D - Visualizing Our Working Matrix

	userId	movieId	rating
0	0	0	4.0
1	0	1	4.0
2	0	2	4.0
3	0	3	5.0
4	0	4	5.0
...	...	...	...
100831	609	2785	4.0
100832	609	2786	5.0
100833	609	2787	5.0
100834	609	1199	5.0
100835	609	2550	3.0

Image E - Matrix Factorization Model

```
class MatrixFactorizer(nn.Module):
    '''
    Initializer: Creates an instance of MatrixFactorizer with random weights between 0 and 0.05.
    '''
    def __init__(self, nusers, nitems, emsize=100):
        super(MatrixFactorizer, self).__init__()
        self.emitem = nn.Embedding(nitems, emsize)
        self.emitem.weight.data.uniform_(0, 0.05)
        self.emuser = nn.Embedding(nusers, emsize)
        self.emuser.weight.data.uniform_(0, 0.05)
```

Image F - Forward Function

```
'''
Return: Returns the dot product of an index, corresponds to the rating of a certain user to a certain movie.
'''
def forward(self, x, y):
    x = self.emuser(x)
    y = self.emitem(y)
    return (x*y).sum(1)
```

Image G - Four Different Models For Experimentation

```
MFmodelA = MatrixFactorizer(nusers, nitems, emsize=100)
MFmodelB = MatrixFactorizer(nusers, nitems, emsize=100)
MFmodelC = MatrixFactorizer(nusers, nitems, emsize=100)
MFmodelD = MatrixFactorizer(nusers, nitems, emsize=100)
```

Image H - Training Function

```
def EpochTrainer(MFmodel, optimizername, epochs=10, learningrate=0.01, weightdecay=0.0):
    if optimizername == "SGD":
        optimizer = torch.optim.SGD(MFmodel.parameters(), lr=learningrate, weight_decay=weightdecay)
    elif optimizername == "Adam":
        optimizer = torch.optim.Adam(MFmodel.parameters(), lr=learningrate, weight_decay=weightdecay)
    elif optimizername == "ASGD":
        optimizer = torch.optim.ASGD(MFmodel.parameters(), lr=learningrate, weight_decay=weightdecay)
    elif optimizername == "Adadelta":
        optimizer = torch.optim.Adadelta(MFmodel.parameters(), lr=learningrate, weight_decay=weightdecay)
    MFmodel.train()

    for i in range(epochs):
        users = torch.LongTensor(trainset.userId.values)
        items = torch.LongTensor(trainset.movieId.values)
        ratings = torch.FloatTensor(trainset.rating.values)
        y = MFmodel(users, items)
        loss = F.mse_loss(y, ratings)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        print(sqrt(loss.item()))

    EvaluateLoss(MFmodel)
```

Image I - Loss Function

```
def EvaluateLoss(MFmodel):
    MFmodel.eval()
    users = torch.LongTensor(testset.userId.values)
    items = torch.LongTensor(testset.movieId.values)
    ratings = torch.FloatTensor(testset.rating.values)
    y = MFmodel(users, items)
    print(y)
    print(ratings)
    loss = sqrt(F.mse_loss(y, ratings))
    print("Root mean squared loss %.3f " % loss)
```

Image J - Comparing RMSE Changes Resulting From Epoch Changes

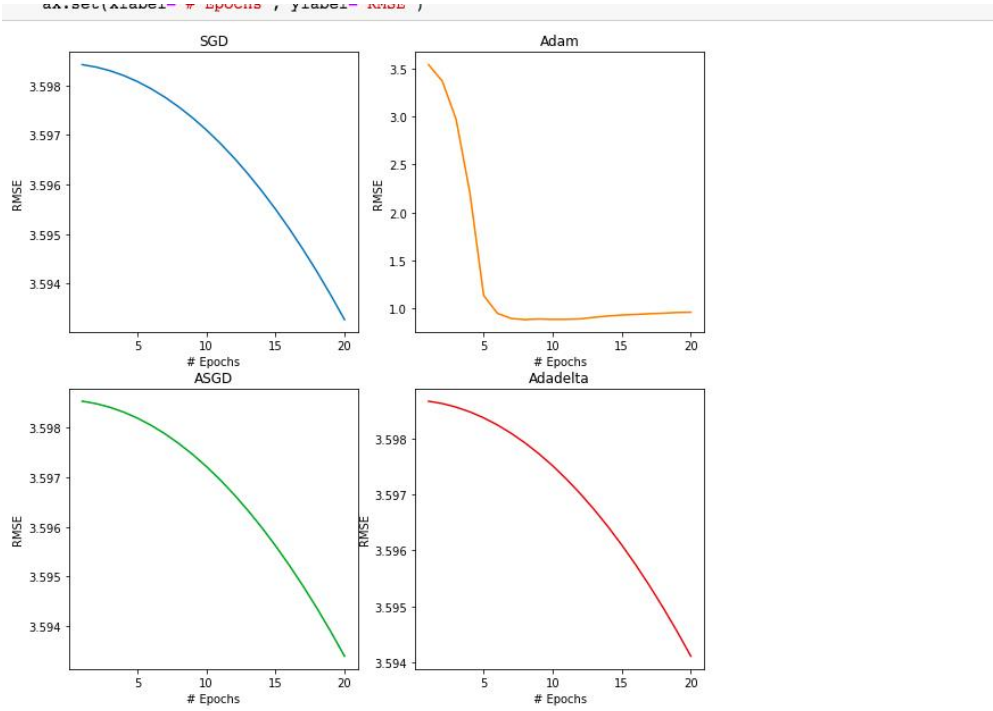

Image K - Comparing RMSE Changes Resulting From Learning Rate Changes

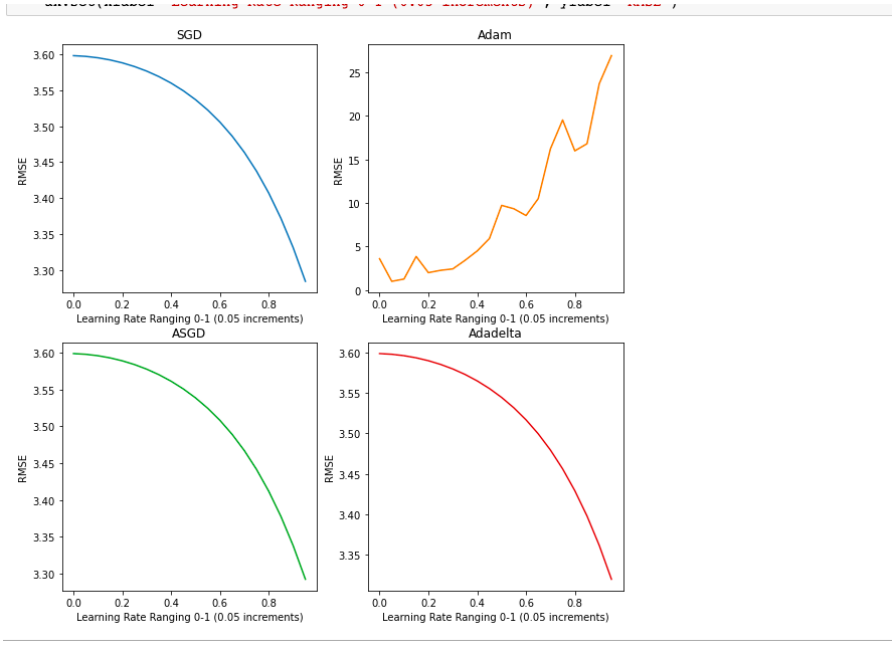

Image L - Showing Optimal Learning Rate

```
Optimal Learningrate for SGD: 0.95 RMSE: 3.2841586345686347
Optimal Learningrate for Adam: 0.05 RMSE: 1.0028005366385948
Optimal Learningrate for ASGD: 0.95 RMSE: 3.2920922756681157
Optimal Learningrate for Adadelta: 0.95 RMSE: 3.3197306762748826
```

Image M

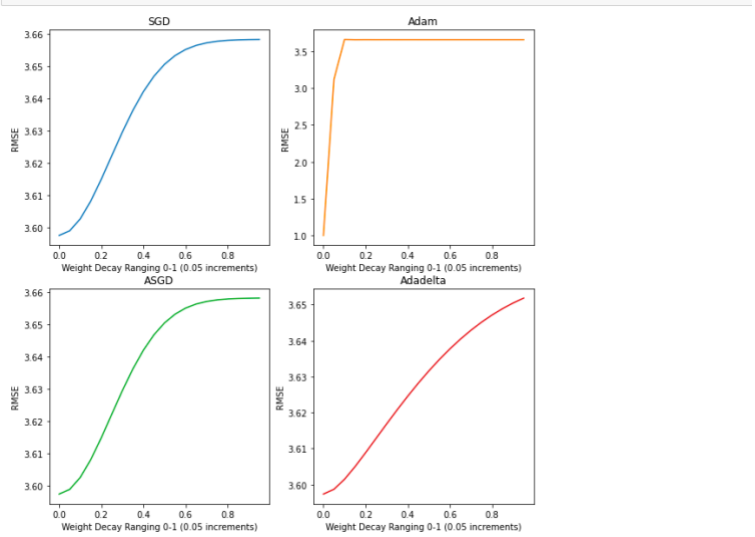

Image N - Distribution Of Ratings

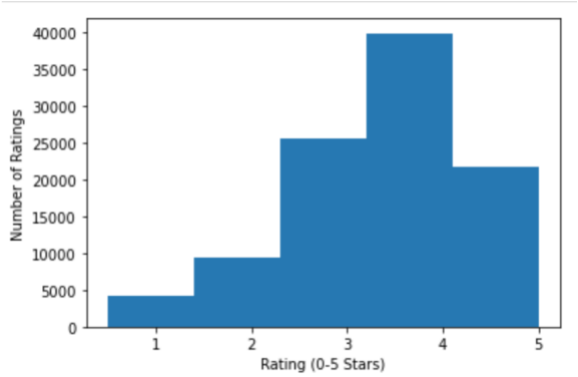

Image O - Distribution Of User Ratings

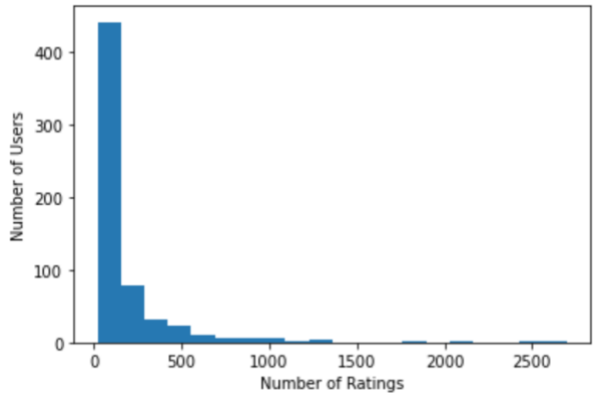

Image P - Distribution Of Movie Ratings

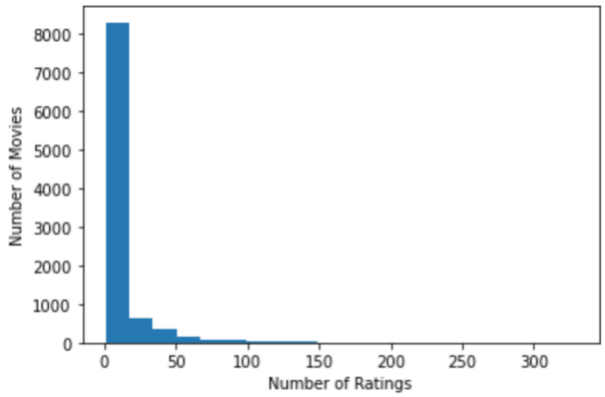

Image Q

```
Optimal Learningrate for SGD: 0.95 RMSE: 3.2841586345686347
Optimal Learningrate for Adam: 0.05 RMSE: 1.0028005366385948
Optimal Learningrate for ASGD: 0.95 RMSE: 3.2920922756681157
Optimal Learningrate for Adadelta: 0.95 RMSE: 3.3197306762748826
```

Image R

```
Optimal Epoch's for SGD: 20 RMSE: 3.593265666666621
Optimal Epoch's for Adam: 8 RMSE: 0.8815988283133119
Optimal Epoch's for ASGD: 20 RMSE: 3.59339013993108
Optimal Epoch's for Adadelta: 20 RMSE: 3.5941034557840577
```

Image S

```
Optimal Weight Decay for SGD: 0.0 RMSE: 3.5975946178810827
Optimal Weight Decay for Adam: 0.0 RMSE: 1.0025131712735575
Optimal Weight Decay for ASGD: 0.0 RMSE: 3.5974073292856135
Optimal Weight Decay for Adadelta: 0.0 RMSE: 3.597340258245493
```