Movie Rating Predictions

Group #8
Group Name: Nut
Thomas Chang, Trevor Holt, Sicheng Jia, Bradley Zhu, Kevin Chuang
Kaggle Rank: 2 | Score: 0.80573

ABSTRACT
With the increasing popularity of streaming sites, software that can recommend movies and TV shows based off of an user's previous watch history is a necessity. While there are certainly many ways to make recommendations, we want to determine which approach is best. Using a publicly available movie dataset, we created and compared models to accurately predict the ratings users will give to movies. In this paper, we explore several different approaches, Adaboost, matrix factorization, and neural networks, with the conclusion that matrix factorization using MCMC is the most accurate based off of the RMSE evaluation metric.

Keywords
Movies; Ratings; Prediction; Data Mining;

1. Introduction
The Internet Movie Database (IMDb) contains extensive information about movies and allows users to log in and rate movies they have seen. It contains information about hundreds of thousands of movies and can provide average ratings based on the collection of ratings provided by individual users.

Given the ratings that a user gives various movies, we conjecture that there are underlying user preferences which correlate to the ratings they will give other movies.

If this holds true, then a predictor can be developed. Using the data of previous users, we can generate a model that is able to leverage a user's tastes in movies to predict whether or not they will enjoy unseen movies and recommend highly rated ones to them.

2. Related Work
Recommending movies based off of previous movie ratings has been a very popular problem, and many approaches have been attempted. Several sites such as Netflix, IMDb, and others have their own implementation of a solution to this problem [22].

For example, Stanford's Adam Sadovsky and Xing Chen utilized a regularized logistic regression model to solve the Netflix challenge [5]. The Netflix challenge was a competition created by Netflix that challenged individuals to predict how a user will rate movies he or she has not seen, given ratings of movies he or she has seen. This problem is identical to ours, and the Computer Science Department at Stanford was able to generate a prediction method with logistic regression using L2 regularization. This method achieved approximately 0.93 root mean squared error on the probe set. Several approaches were attempted, including centering ratings around the user's average rating, centering around the movie's average, or a weighted combination of the two.

Another example of such an approach is Carnegie Mellon's Neil Armstrong and Kevin Yoon's work on a similar topic using a kernel regression and a linear model tree [4]. The kernel regression model uses an instance based learner to deduce the behavior of complex target functions. Using such a model, they were able to achieve prediction accuracy with 14.11% error. Using a model tree with an m5 algorithm, they were able to achieve an error of 14.40%.

3. Problem Definition and Formalization
The objective our our project is to accurately predict ratings that users will assign to movies. Our hypothesis is that users will rate certain genres more highly than other genres and also that certain movies just receive better ratings overall. If our hypothesis stands true, this would allow us to generate a model that will be able to predict new ratings given certain information.

Formally, we aim to minimize the mean squared difference between our predicted user ratings and their true ratings.

$$\Sigma_r (r - r')^2$$

In order to make predictions on how users will rate new movies, we first need preprocess the provided data and extract the information relevant to our problem, such as user IDs, movie IDs, ratings, and genre from the IMDb database. Afterwards, we need to create and train a model

with the information that we collected. Possible models that we can use include Adaboost, matrix factorization, and neural networks, each of which are explained in the following sections.

In conclusion, our approach contains the following steps:

- Preprocessing data from the provided MovieLens dataset to get the relevant information.
- Training and testing different fitting models including Adaboost, matrix factorization, and neural networks with the preprocessed data.
- Analyzing our findings and selecting the best model for our problem, taking into account our limitations such as time and computational power.
- Reporting our findings.

4. Data Preparation, Description, and Preprocessing

All data is based on MovieLens' latest datasets. This data includes movies, with their respective tag and genre. It also contains information about a user's ratings for a subset of movies. All of the original data files are given in CSV format. We describe the unprocessed data in the following table.

File Name	Header	Description
train_ratings.csv, val_ratings.csv	userId, movieId, rating	User ratings for particular movies
movies.csv	movieId, title, genre	Movie information
genome-tags.csv	tagId, tag	Fine-grained categories (tags)
genome-scores.csv	movieId, tagId, relevance	Degree of relevance of each tag, per movie
tags_shuffled_rehashed.csv	userId, movieId, tag	User assigned tags to movies
links.csv	movieId, imbdId, tmbdId	Movie ID mappings to external databases

In order to work with this information, we primarily use the Pandas and Numpy libraries to manipulate the data in code [1][2]. Pandas offers high-level abstractions while Numpy provides fast computations on vectors and matrices. By interpreting the provided data points as matrix entries, we are then able to leverage Numpy's power.

After reading our data into a Pandas dataframe or numpy array, we are then able to perform a few more necessary steps of preprocessing. Although we are already provided separate training and validation data, we believed it would be useful to reshuffled the data. This process helps to lower to impact of any imbalance in the original dataset.

We also have to do some preprocessing in order to use the libFm library (described in detail in later sections) [18]. With the aid of Pandas dataframes, we create new training and test CSVs. Both have removed headers (as required by the tool), and the test file is populated with dummy ratings. A peculiarity of the libFm library is that it does not provide a way to test on unseen data. The workaround is to feed it the test data with dummy values as "validation" data.

5. Methods Description

5.1 AdaBoost

One model our team considered was Adaptive Boosting, or AdaBoost, developed in 2003 by Yoav Freund and Robert Schapire. AdaBoost is an ensemble method, which is a strong classifier built from a combination of weak classifiers. Freund and Schapire's intuition behind the algorithm is that combining weak classifiers allow for modeling of complex decision boundaries. [8]

The high level idea of AdaBoost is to combine weak classifiers (denoted as h_t) to create a strong ensemble model (denoted as H). The loss function for our weak classifier is defined as follows:

$$\epsilon_t = \sum_n w_t(n)[y_n \neq (x_n)]$$

Where w denotes the weight of each point in our dataset. The loss function for AdaBoost is essentially the number of misclassifications, with each misclassification contributing a variable amount $w(n)$ to the error ϵ.

We begin with each point in the training data weighted equally, and each iteration we train a weak classifier on our training data. For each data point misclassified by h_t in iteration t, we increase the weight for iteration $t+1$, since

we want h_{t+1} to be able to correctly classify these points. Conversely, points that are correctly classified in iteration *t* have their weights reduced, since they are not as important to classify in iteration *t+1*. In the next iteration, our goal is to train another weak classifier to minimize the weighted classification error, with our new weights established after the previous iteration. We continue until the loss function of our ensemble classifier converges, or until we have reached the maximum number of iterations set by the hyperparameter *n_estimators.*

Figure 1 shows a simple example on a toy dataset. We see that by training many simple classifiers we are able to model complex decision boundaries. It is worth noting that while we have been describing AdaBoost as a classification method, it can be applied to regression as well.

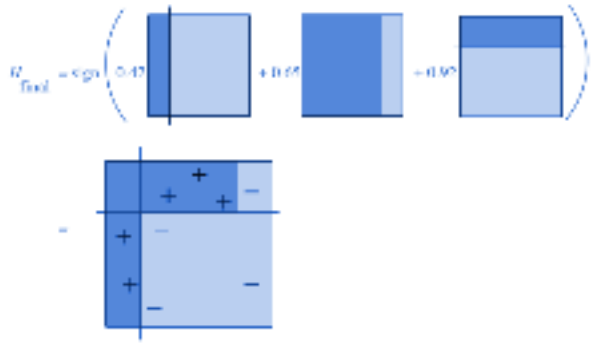

Figure 1: AdaBoost example. Note that weak classifiers for this example are one node decision trees. Each weak classifier output is weighted and summed to arrive at final decision boundary for an ensemble classifier *H*. [7]

One main benefit of AdaBoost is that it can be an ensemble of any weak classifier; since ϵ is defined only by the misclassification error, our weak classifier can be anything we define, such as a simple decision tree or linear regression line. This allows for lots of flexibility when building a model. We can further increase the complexity of our model by mixing weak classifiers for iterations. This allows us to mix many different classifiers to create a highly complex decision boundary.

While AdaBoost does allow us to create complex models, it has major flaws that we came across during execution. First is time; training a weak classifier takes a considerable amount of time when working with data as large as the given training data. Furthermore, we have to train multiple classifiers to arrive at our ensemble model, which can be extremely computationally expensive. Another drawback is overfitting. While it is impressive that AdaBoost is able to create complex decision boundaries, this may not be what we want, since it is possible that our input data is noisy. For each iteration, AdaBoost will try to classify as many points correctly as it can, and will increase the weights of misclassified points. In the case of noisy data, it is very possible for noisy points to become heavily weighted; this forces us to add a classifier to our ensemble that tries to fit noisy data, when we should ignore it altogether. One way to combat overfitting noisy data is to limit *n_estimators,* but as stated earlier, determining the optimal value is extremely computationally expensive.

5.2 SVD

SVD or Singular Value Decomposition is a powerful tool and one of the most widespread unsupervised learning algorithms. SVD states that any mxm matrix can be decomposed into three different matrices.

$$A = U\Sigma V^T$$

> *Where **A** is an m x n matrix, **U** is an (m x m) orthogonal matrix, **Σ** is an (m x n) nonnegative rectangular diagonal matrix, and **V** is an (n x n) orthogonal matrix.*
>
> *U is also referred to as the left singular vectors, **Σ** the singular values, and **V** the right singular vectors*

Intuitively, we can imagine each matrix as a geometric transformation. Any mxm matrix can be represented as a rotation, scaling, and then another rotation. In the case of machine learning, each matrix can have an associated meaning.

For example, let's consider the following example where

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 3 & 3 & 3 & 0 & 0 \\ 4 & 4 & 4 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 \\ 0 & 2 & 0 & 4 & 4 \\ 0 & 0 & 0 & 5 & 5 \\ 0 & 1 & 0 & 2 & 2 \end{bmatrix}$$

In this scenario, imagine each column representing a specific movie, and each row representing a user. Each user rates a movie from 0-5 where 0 means a user does not like a movie. We can imagine that the first three movies would be Avengers, Star Wars, and Avatar (Sci-fi), while the last two movies are the Notebook and Titanic (Romance).

We can perform singular value decomposition with sklearn and as a result have these matrices.

$$U = \begin{bmatrix} 0.13 & 0.02 & -0.01 \\ 0.41 & 0.07 & -0.03 \\ 0.55 & 0.09 & -0.04 \\ 0.68 & 0.11 & -0.05 \\ 0.15 & -0.59 & 0.65 \\ 0.07 & -0.73 & -0.67 \\ 0.07 & -0.29 & -0.32 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 12.4 & 0 & 0 \\ 0 & 9.5 & 0 \\ 0 & 0 & 1.3 \end{bmatrix}$$

$$V^T = \begin{bmatrix} 0.56 & 0.59 & 0.56 & 0.09 & 0.09 \\ 0.12 & -0.02 & 0.12 & -0.69 & -0.69 \\ 0.40 & -0.80 & 0.40 & 0.09 & 0.09 \end{bmatrix}$$

Each matrix has its own value, and in this case we delete the third column from the first two matrices and the last row in the third matrix because their values are smaller than a certain threshold. This leaves us with:

$$U = \begin{bmatrix} 0.13 & 0.02 \\ 0.41 & 0.07 \\ 0.55 & 0.09 \\ 0.68 & 0.11 \\ 0.15 & -0.59 \\ 0.07 & -0.73 \\ 0.07 & -0.29 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 12.4 & 0 \\ 0 & 9.5 \end{bmatrix}$$

$$V^T = \begin{bmatrix} 0.56 & 0.59 & 0.56 & 0.09 & 0.09 \\ 0.12 & -0.02 & -0.12 & -0.69 & -0.69 \end{bmatrix}$$

Finally, we can interpret the data given. For the matrix of U, we can interpret each row as how much a user likes a certain feature of movie (we will assume in this example that those features correspond to genres, though this is not always the case). For example, the first user prefers Sci-fi (0.13) slightly more than romance (0.02). In comparison, the second to last user prefers romance (-0.73) over sci-fi (0.07).

The matrix **Σ**'s diagonal entries contain the weights of sci-fi and romance respectively.

The columns of matrix V^T then represents the degree of which a movie belongs to a category. For example, the first movie (Avengers) has is more related to Sci-fi (0.56) rather than romance (0.12). This is a basic example and description of SVD, but to estimate movie ratings, more complicated algorithms using SVD are used.

In the context of movie rating predictions, we are posed with a new challenge: incomplete data. In the ideal scenario, we are given the matrix A used above. However, we do not have a user's rating for every movie (if we did, we would not have the problem of needing to predict user ratings to begin with), and instead we have a sparsely populated matrix as shown in the following example.

$$A = \begin{bmatrix} 1 & 1 & 1 & ? & 0 \\ 3 & ? & 3 & ? & 0 \\ ? & 4 & 4 & 0 & ? \\ 5 & ? & 5 & 0 & 0 \\ 0 & 2 & 0 & ? & 4 \\ ? & 0 & ? & 5 & ? \\ 0 & ? & 0 & 2 & 2 \end{bmatrix}$$

One popular technique, made famous by Simon Funk in the Netflix Challenge, finds a matrix decomposition using only sparse data [17]. Let $\hat{r}_{ui} = q_i^T \cdot p_u$ be our prediction for a single (user, movie) pair, where q_i is the feature vector for a single movie, and p_u feature vector for a user. We can then define our loss as follows:

$$\sum_{r_{ui} \in R_{train}} \left(r_{ui} - \hat{r}_{ui}\right)^2$$

Additionally, we may want to enhance our predictions with two biases: one that represents the individual movie bias and one that represents the individual user's rating bias. The movie bias is important, as a particularly well-made movie could be rated highly by someone who does not usually enjoy that movie's genre. The user bias is likewise important because different users have different preferences for weighting movies (some rate on a bell-curve distribution, some tend to rate all movies highly, etc.). We may also want to factor in the global movie average into

our predictions for even better results. Taking these modifications into account, our new prediction then becomes

$$\hat{r}_{ui} = q_i^T \cdot p_u + \mu + b_i + b_u$$

To avoid overfitting, we may also want to penalize large parameters with regularization, giving us our final loss function:

$$\sum_{r_{ui} \in R_{train}} (r_{ui} - \hat{r}_{ui})^2 + \lambda (b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2)$$

Using this loss function, we then have an iterative method for learning our decomposed matrices using the data we have:

$$b_u \leftarrow b_u + \gamma(e_{ui} - \lambda b_u)$$
$$b_i \leftarrow b_i + \gamma(e_{ui} - \lambda b_i)$$
$$p_u \leftarrow p_u + \gamma(e_{ui} \cdot q_i - \lambda p_u)$$
$$q_i \leftarrow q_i + \gamma(e_{ui} \cdot p_u - \lambda q_i)$$

Here, γ and λ are hyperparameters. Since the popularization of this technique, several new approaches to finding parameter values have arisen, such as Alternating Least Squares [18], and Markov Chain Monte Carlo [19].

5.3 Neural Networks

Neural Networks are a powerful machine learning tool that can be customized for many applications. For the purpose of this project, we used neural networks to implement matrix factorization. The layout of the neural networks consists of two inputs for the input layer, two hidden layers, with two nodes each, and one node in the output layer. The idea of using neural networks to implement matrix factorization was taken from Alkahest's blog post [9].

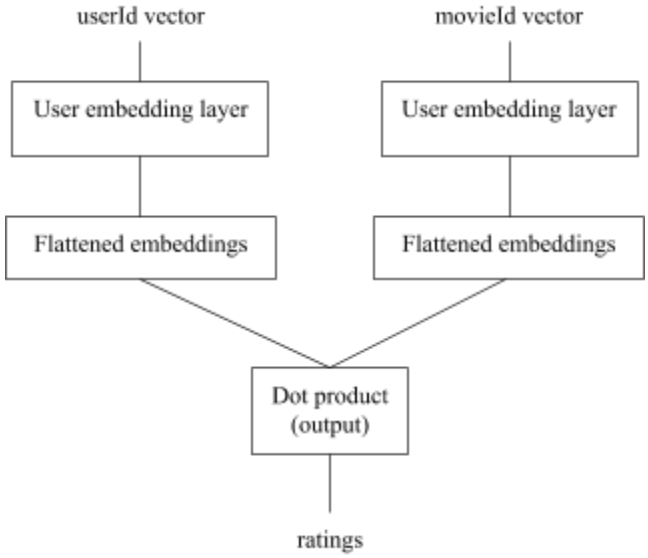

Figure 2: Neural network implementation of matrix factorization. The users and movies are passed into an embedding layer, which is then flattened, and multiplied together, returning the prediction rating.

First, the train ratings are split into three pandas dataframes, a userId dataframe, a movieId dataframe, and a rating dataframe. The userId and movieId are each passed in the neural network as inputs. The userId vector is then expanded into a (number of users) x (number of factors) user embedding matrix, where number of factors is a hyperparameter. Symmetrically, the movieId vector is expanded into a (number of movies) x (number of factors) embedding matrix. These two matrices are then flattened, and multiplied together through dot product to obtain a ratings prediction.

In order to implement this layout, the Keras library was used [10]. The Keras library has multiple predefined layers such as an "Dense layer" that can be used to build complex neural networks. In our case, we used the two of the default Sequential() models, then to each we added an embedding layer and then a flatten layer. At the end the two layers were merged using a dot product layer, which was able to directly output the ratings.

Inside the embedding layer, the hyperparameter, number of factors, must be specified. This hyperparameter determines the size of the latent factors matrix. Having a latent factors matrix that is too small will cause the model to underfit as

the latent factors matrix cannot capture the latent variables between the user-movie pair and its corresponding ratings. While on the other hand, having a latent variable matrix that is too large, would cause overfitting, while also increasing the run time significantly.

Other hyperparameters include the number of epochs, the learning rate, and learning rate decay. The epochs specifies the number of times to run the forward and backward passes through the neural network. While generally it's good to have a high number of epochs, having the number be too high can cause overfitting on the training data. The learning rate specifies how fast the model adjusts the weight of its layers. The learning rate is related to the number of epochs as smaller learning rates require more epochs, but are more detailed and fine tuned, while larger learning rates is able to get to the optimum quicker, thus requiring less epochs, but also being much coarser and not as accurate. The learning rate decay resolves this issue by decreasing the learning rate each epoch. This benefits our application as it allows use to set the learning rate to be high initially, allowing us to reach the general area of the optimum quickly, then lowering the learning rate in later epochs to fine tune the weights.

In addition to the Keras library, the fastai library was also explored for neural network implementations of matrix factorization [11]. The fastai library is built on top of the pytorch library, which is a deep learning platform [12]. The library actually already includes neural network modules that implement collaborative filtering. The library was easy to use, but was not as customizable as the Keras library. We used the EmbeddingDotBias class for collaborative filtering [13].

6. Experiments Design and Evaluation

6.1 Baseline

To analyze our results, we use a simple baseline predictor: individual movie average predictor. The individual movie average is the average movie rating for each movie. Given a (user, movie) pair, this predictor will simply predict that movie's average rating.

The results are summarized in the following table.

	Validation MSE	Public Test MSE
Per-movie average	0.905	0.955

6.2 AdaBoost

Sklearn provides a framework for regression with AdaBoost that we used to predict movie ratings. The default weak learner is a three-level decision tree, and the default *n_estimators* is 50. We fit the classifier on training data using a range of *n_estimators* of [1,3,5, ... 99] and used separate validation data to determine the Root Mean Squared Error. The table below shows the results for a subset of values for *n_estimators*. After testing all values within our range, *n_estimators = 41* yielded the highest validation score about 1.045 RMSE. [8]

n_estimators	RSME on validation Data	Time (s)
1	6.004	35
21	1.270	732
41	**1.045**	**1371**
61	1.201	1439
81	1.207	1800

As stated earlier, one major flaw in the AdaBoost algorithm is its susceptibility to noisy data. In our large dataset it is very possible to have noisy data points, and these points will eventually be included in our ensemble model Furthermore, the dimensions for our data were very large; the amount of features and training examples made it extremely difficult for a weak classifiers to be successful. This means we would have to add more weak learners to our ensemble, but this also increases overfitting by listening to noise. We also see that the time increases significantly as we add more estimators; even if the optimal models had a very high *n_estimators* parameter, it would take a very long time to construct. After a few days of tuning hyperparameters, we concluded that AdaBoost was not the best method for the data given.

6.3 Matrix Factorization

We used two implementations of matrix factorization / SVD algorithms: The Surprise library by Nicolas Hug [20] and the libFm library by Steffen Rendle [21].

The Surprise library's SVD algorithm decomposes a movie-user similarity matrix into a lower dimension using the methods described before. This algorithm runs SVD over many iterations, updating bias and the lower

dimension matrix by running Stochastic Gradient Descent to converge to an optimum. Surprise also has an implementation of this matrix factorization called SVD++ which takes into account the implicit rating of users when running SVD.

Our SVD results varied depending on whether or not the provided data was reshuffled between training and validation, and whether we were using standard SVD or SVD++. The RMSE for each dataset is as follows.

	SVD (Test RMSE)	SVD++ (Test RMSE)
Shuffled	0.83	**0.81**
Normal	0.86	0.85

Shuffling the data beforehand changes the way stochastic gradient descent converges, and as a result changes our RMSE. We find out this way that SVD++ with a shuffled dataset is our most successful implementation of SVD.

The problem with SVD is that not only is it not the best method with the lowest RMSE, it takes a long time to compute. While SVD can take around 10 minutes to compute, SVD++ took around 10 hours to compute. This in addition that it does not have the lowest RMSE means that SVD is not the best method to use.

In addition to the Surprise library, we employed the libFm SVD library, which includes additional methods for parameter learning. We did not directly test the Stochastic Gradient Descent method to compare with the Surprise library, since the algorithm has a large runtime and it is likely that these two implementations provide similar results. Instead, we used libFm's implementation of the Markov Chain Monte Carlo method for learning parameters. There are several inputs to run this method: number of iterations, initial standard deviation, and k (number of features). We set the number of iterations to 100 and the initial standard deviation to 0.6 (this only affects convergence speed, and thus not lead to worse or better results in the end). We show the results of several runs below:

Data	k	Public Test MSE
Not reshuffled	16	0.84490
Reshuffled	16	0.81758
Reshuffled	30	0.81526
Combined (train + val)	40	0.80601
Combined	35	0.80549

We see from these results that matrix factorization with MCMC method works very well. We attribute the better success of this method over Stochastic Gradient Descent to the fact that this method requires less input parameters. With Stochastic Gradient Descent, there are more hyperparameters, and given the long runtime of the algorithm, it is difficult to navigate the search space of hyperparameters to find the optimal settings.

6.4 Neural Networks

For the neural network implementation of collaborative filtering, we used two models: one defined in Alkahest's "Collaborative Filtering in Keras" blog post, and the other defined in the fastai library. Both models are similar in that they both involve the dot product of the user and movie latent factors matrix.

For the Keras model, we trained the model for 30 epochs, with the number of factors being 50, and the default learning rates and decay. The model took over 16 hours to train, and had a training RMSE loss of 0.7548 and a validation RMSE loss of 0.9161. On the test set, the RMSE error was 0.91835. This performed worse than SVD, which is to be expected initially as the hyperparameters were not tuned at all. We only ran this model once due to the large amount of time and computing power it took to train. This was a practical visualization of the downside to neural networks.

For the FastAi model, we trained the model for 5 epochs, with the number of factors being 50, and the learning rate being 1e-3 and learning rate decay being 1e-5. The resulting training error was 0.7260 and validation error was 0.8322. The test RMSE error was 0.88165, which was still not as good as the SVD result. For the FastAi model, the training time was 2 hours. We found that at epochs higher than 5, the model began to overfit, as validation error began to increase.

We believe that neural networks would be able to give us the most optimal results. In particular, with the customizability of the Keras library, we can add additional layers and change the collaborative filtering model to a deep learning model. For this assignment, we lack the computational power and time to train and tune the hyperparameters such a model, thus we look to other less expensive models.

7. Conclusion

The goal of our project was to create a model that can accurately predict ratings users would give to new movies based on the genre of the new movies, the ratings the user has given in the past, and the ratings the movies have already received.

We were most successful in our goal using matrix factorization, specifically the libFM implementation of the Markov Chain Monte Carlo method. We also tested Adaboost and neural network models, but we were not able to utilize them to create a better model than matrix factorization.

A common theme among all of our different methods was a lack of time and computational power to best finetune the hyperparameters. We were working with a very large dataset, so each test runthrough would take hours, which made it extremely difficult to thoroughly test all these different models.

While we obtained the best results using matrix factorization, we hypothesize that neural networks, in particular the Keras library, would give us better results if we had more time to experiment with additional layers and change the collaborative filtering models to a deep learning model. Unfortunately, we would not have been able to realistically finetune the hyperparameters of such a model with our personal laptops and limited time.

8. Task Distribution Form

Task	People
Collecting and preprocessing data	Thomas Chang, Trevor Holt, Sicheng Jia, Bradley Zhu, Kevin Chuang
Implementing / running Adaboost	Trevor Holt
Implementing / running matrix factorization	Sicheng Jia, Thomas Chang, Bradley Zhu
Implementing / running neural network	Sicheng Jia
Evaluating and comparing algorithms	Thomas Chang, Trevor Holt, Sicheng Jia, Bradley Zhu, Kevin Chuang
Writing report	Thomas Chang, Trevor Holt, Sicheng Jia, Bradley Zhu, Kevin Chuang

9. References

[1] Pandas: Python Data Analysis Library. 2019. https://pandas.pydata.org/.

[2] Numpy Scientific Computing Library. 2019. http://www.numpy.org/.

[3] sklearn.linear_model.LinearRegression. 2019. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html.

[4] Armstong, Nick. Yoon, Kevin. Movie Rating Prediction. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.1964&rep=rep1&type=pdf

[5] Sadovsky, Adam. Chen, Xing. Evaluating the Effectiveness of Regularized Logistic Regression for the Netflix Movie Rating Prediction Task. http://cs229.stanford.edu/proj2006/SadovskyChen-NetflixL1LogReg.pdf

[6] Luboobi, Patrick. Foundations of Machine Learning: Singular Value Decomposition (SVD). https://medium.com/the-andela-way/foundations-of-machine-learning-singular-value-decomposition-svd-162ac796c27d

[7] Sankararaman, Sriram. 2019. Ensemble Methods. *UCLA CM 146: Intro to Machine Learning*(February 2019).

[8] sklearn.ensemble.AdaBoostRegressor. Retrieved March 20, 2019 from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html

[9] Alkahest. 2016. Collaborative filtering in Keras http://www.fenris.org/2016/03/07/index-html

[10] Keras: The Python Deep Learning library https://keras.io/

[11] FastAi, Neural Network Library https://docs.fast.ai/

[12] Pytorch: Open Source Deep Learning Platform https://pytorch.org/

[13] FastAi Collaborative Filtering EmbeddingDotBias model https://docs.fast.ai/collab.html#EmbeddingDotBias

[14] Funk, Simon. 2006. Netflix Update: Try This at Home. https://sifter.org/~simon/journal/20061211.html

[15] PILASZY, I., ZIBRICZKY, D., AND TIKK, D. 2010. Fast als-based matrix factorization for explicit and implicit feedback datasets. In Proceedings of the 4th ACM Conference on Recommender Systems (RecSys'10). ACM, New York, NY, 71–78.

[16] SALAKHUTDINOV, R. AND MNIH, A. 2008b. Probabilistic matrix factorization. In Advances in Neural Information Processing Systems 20.

[17] Hug, Nicolas. 2017. Surprise, A Python Library For Recommender Systems. https://github.com/NicolasHug/Surprise.

[18] Rendle, S. 2012. Factorization machines with libFM. ACM Trans. Intell. Syst. Technol. 3, 3, Article 57 (May 2012), 22 pages. DOI = 10.1145/2168752.2168771 http://doi.acm.org/10.1145/2168752.2168771

[19] James Bennett; Stan Lanning. 2007. The Netflix Prize. Proceedings of KDD Cup and Workshop 2007. https://web.archive.org/web/20070927051207/http://www.netflixprize.com/assets/NetflixPrizeKDD_to_appear.pdf