Recommendation System for MovieLens Dataset

DS-GA 1004 Big Data Final Project (Group 30)

Disha Lamba
Electrical and Computer Engineering
New York University
dl4747@nyu.edu

Prakriti Sharma
Electrical and Computer Engineering
New York University
ps4425@nyu.edu

Sai Kiran Challa
Electrical and Computer Engineering
New York University
vc2118@nyu.edu

1 Introduction

A recommender system is a subclass of information filtering systems that seeks to predict the "rating" or "preference" a user would give to an item. Recommender systems are utilized in various areas, including movies, music, news, social tags, and products in general. Recommender systems typically produce a list of recommendations and there are few ways in which it can be done. Two of the most popular ways are – collaborative filtering and content-based filtering.

In the project, we aim to solve a realistic, large-scale problem of building and evaluating a collaborative-filter based recommender system using the MovieLens dataset. We first start with implementing the popularity baseline model followed by collaborative filtering approach with latent factor decomposition, a matrix factorization approach to decompose user-item interaction information to recommend users' most relevant movies, with Alternative Least Square (ALS) method. We further implement a model on single-machine to compare its performance from Spark ALS, and finally conduct compound scaling and plot various visualizations for hyper-parameters to detect hidden correlations between latent factors and other features. For this project we have used PySpark - 3.0.1.

2 Data

2.1 MovieLens Dataset

For this project, we will use the latest MovieLens dataset collected and made available by GroupLens Research, a research laboratory at the University of Minnesota. MovieLens provides an online movie recommender application that uses anonymously-collected data to improve recommender algorithms. The data sets were collected over various periods of time, depending upon the size of the set.

Two versions of the dataset have been used:

1. Small: ML-latest-small dataset has 100,000 ratings and 3,600 tag applications applied to 9000 movies by 600 users.
2. Large: ML-latest dataset has 27,000,000 ratings 1,100,00 tag applications applied to 58,000 movies by 280,000 users.

The small dataset is a subset of the full dataset. Generally, it's a good idea to start building a working program with a small dataset first to get faster results and better performance while interacting and exploring the data and models.

2.2 Data Visualizations and Manipulations:

First, for exploratory data analysis, we loaded the data into pandas dataframes. We wanted to first understand our data and draw some key insights before going further into recommendations. Like different features in each dataset and the relationship between them, handling missing values, identifying outliers, etc.

First, we checked for NaN values, and both versions of the datasets had zero NaN values. Next, from figure 1, we can observe that most of the users have reviewed the movies they have watched with a 4-star rating, followed by 3-star and then a 5-star.

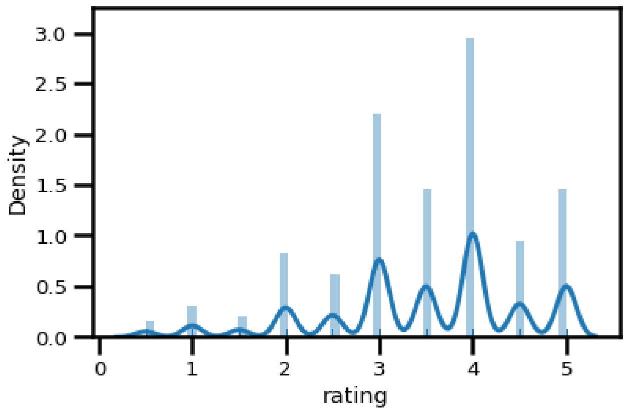


Figure 1: Count plot for user ratings

Next, we have plotted the genre based count of movies using a bar graph. From figure 2, we observe that 'Drama' genre is the most favorite genre among the users followed by 'comedy' and 'action'.

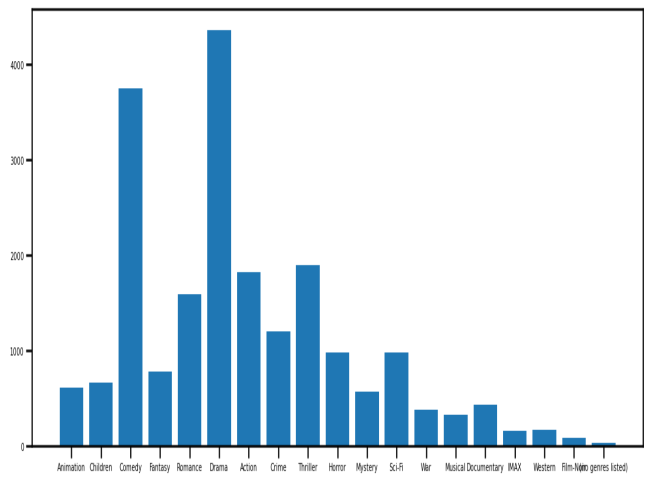


Figure 2: Genre base number of movie count

2.3 Data Processing

For better efficiency, we converted the dataset format from CSV to Parquet as the very first step, then completed the entire data processing procedures. Due to the limit of computing resources, we downsampled the large dataset into 1%, 5%, and 25% apart from the full dataset. The procedures described in the report apply to all sizes of data.

Next, for each unique user, we split the small dataset based on the year of release. Interaction where the year is 2018, data will randomly get divided into validation and test datasets respectively otherwise into the training dataset. For the full dataset, we randomly split 70% of users from the unique valid user pool for the training set, 20% users for the validation set, and the rest 10% users for the test set.

3. Model Implementation

For the models we have used two main libraries:

1. pyspark.ml: which contains newer APIs for constructing ML pipelines to handle data in DataFrames and contains various recommendation functions.

2. pyspark.mllib: which contains original APIs that handle data in RDDs. pyspark.mllib.evaluation has evaluation metrics like Pecision_at_k and *M*ap.

Since our original dataset was in data frame format, we adopted pyspark.ml to fit the recommender model, and used pyspark.mllib for evaluation**.**

We have used the Alternating Least Square (ALS) algorithm from the PySpark library. It divides the utility matrix into two parts: a user-to-item matrix U and a user-to-item matrix V. The ALS algorithm attempts to determine ideal factor weights to minimize the least squares between expected and actual ratings by uncovering latent factors that explain observed users' item ratings. It's also built to run in parallel, which greatly improves training efficiency.

4. Evaluation Metrics

In order to measure the overall performance of our recommender system, we tried to measure the performance of our model using all the metrics that are popularly used for evaluations in the recommender systems. Following are the metrics that we used for measurements:

1. RMSE: It stands for rooted mean square error and it gives the RMSE between the predicted values and the ground truth rating of all the combinations of user-item interactions.

2. NDCG: It stands for Normalized Discounted Cumulative Gain which gives the count of top k recommended items on the basis of true relevant item sets, averaged across all the users by taking into account orders.

3. MAP: It stands for Mean Average Precision and it gives the count of all the recommended items that are present in the true relevant item sets, averaged across all the users. It also assigns a penalty in accordance with the order.

4. Precision at k: This method gives the count of all the recommended items that are present in the true relevant item sets, averaged across all the users. Unlike MAP, the order of recommendation of the items is not relevant here.

In addition to these metrics mentioned above, we also recorded the time taken to fit the test data on our model and evaluate the efficiency. We have also calculated the accuracy of our results in most of the measurements.

5. Results

In this section, we will explain our model implementation, choices, and the techniques used to obtain results.

5.1 Hyperparameter tuning

We have used grid search to tune the following hyperparameters: max iterations (maxIter), regularization parameter (regParam), and rank (rank) of the latent factor matrix. We have also used these hyperparameters for evaluating our models - Baseline popularity, ALS, LightFM, and Annoy based on evaluation metrics such as RMSE, MAP, Precision at k, and NDCGs. In the next sections, we will discuss the qualitative analysis of the hyperparameters for different evaluation metrics.

5.1.2 LightFM Model: LightFM provides the python implementation of a number of popular recommendation algorithms for both implicit and explicit feedback. It generates the traditional matrix factorization algorithms by incorporating both the item and user metadata. In our project, we have implemented LightFM in a jupyter notebook and then we have calculated the precision values, accuracy (on both training and test data), and time taken for all the hyper-parameter combinations using our LightFM model.

The summary of our model implementation and choices is as follows:

- Input a dataframe of interactions, then we have used pandas.pivot_table to convert to an initial utility matrix, then we have used scipy.sparse.crs_matrix to convert to coordinate format, which is the allowed format for fitting LightFM models.
- Random training and test dataset split with built-in function Lightfm.cross_validation.
- Next, we fit the model over both Weighted Approximate Rank Pairwise (WRAP) Loss and Bayesian Personalized Ranking (BPR) for better performance comparison.
- Finally, we get our precision at k, total time taken, and accuracy result for both training and test datasets.

For an ideal model, the aim is to have high accuracy and high precision value. Keeping that as the baseline, our next step was to hyper-tune our parameters on the following values:

maxIters = [5, 10, 15]

regParams = [0.1, 0.01]

Rank = [4, 5, 6, 7, 8, 9, 10, 11, 12]

Fig 3, 4, and 5 show the grid search we performed on all possible hyper-parameters combinations. There were a total of 54 combinations like this.

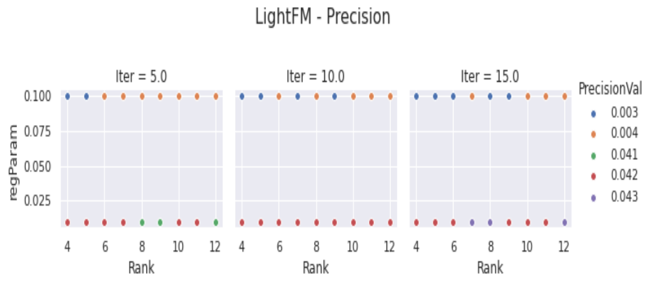


Figure 3: Compound scaling for MaxIter w.r.t Rank and RegParam for precision

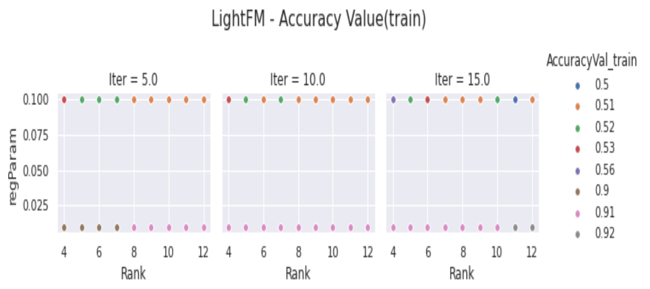


Figure 4: Compound scaling for MaxIter w.r.t Rank and RegParam for accuracy on the training set

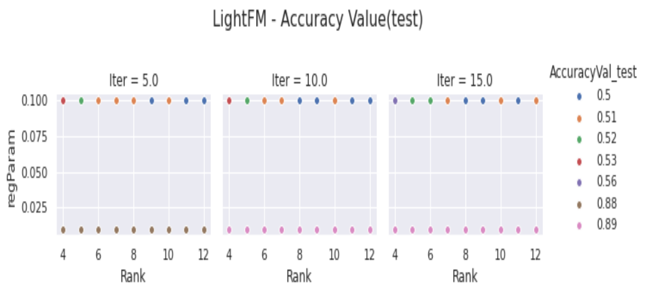


Figure 5: Compound scaling for MaxIter w.r.t Rank and RegParam for accuracy on the training set

From figures 3, 4, and 5 we observed that to achieve high accuracy and high precision we can set our hyperparameters as regParam = 0.01 and MaxIter = 15. We observe that metrics improve with a smaller regularization parameter(0.01) and maximum iteration does not have much impact on the performance. Lastly, for rank, we observe that the score of all metrics improves along with an increase in rank. For rank, we settle for the best accuracy, precision, and run-time tradeoff at rank = 12 from figure 6.

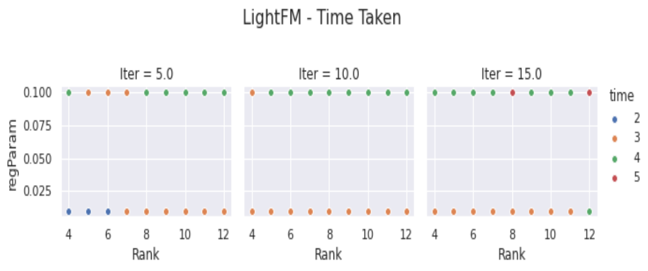


Figure 6: Time tradeoff w.r.t Rank and RegParam

5.1.3 ALS Model: Alternating Least Square model also generates the traditional matrix factorization algorithms and runs itself in a parallel fashion. It is implemented in Apache Spark ML and is built for large-scale collaborative filtering problems.

The summary of our model implementation and choices is as follows:

- First, we input the training and validation datasets in parquet format. Then, we used the spark.ml library to implement the ALS algorithm to learn the latent factors which have the following parameters with explicit feedback.
- We have declared the values for MaxIter, regParam, and rank as defined in section 5.1.2.
- We have also used coldStartStrategy to drop rows in the dataframe of predictions that contain NaN values.
- And lastly, we have defined 'nonnegative = True' to use negative constraints for least squares.

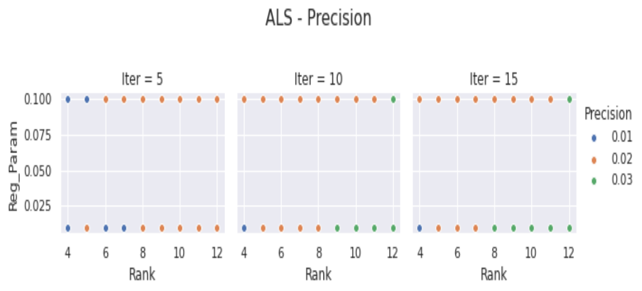


Figure 7: Compound scaling for MaxIter w.r.t Rank and RegParam for precision

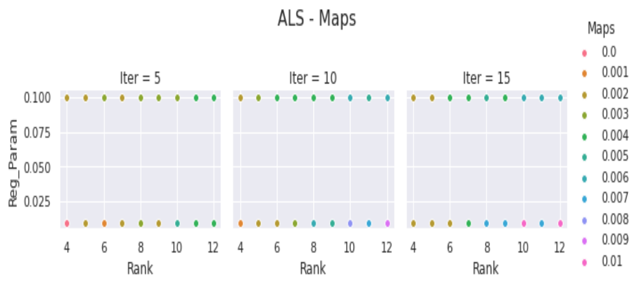


Figure 8: Compound scaling for MaxIter w.r.t Rank and RegParam for Maps

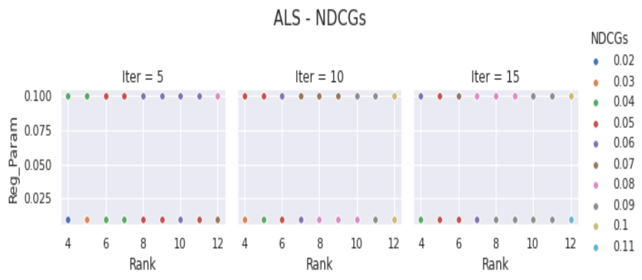


Figure 9: Compound scaling for MaxIter w.r.t Rank and RegParam for NDCGs

For an ideal model, it is desired to have high precision, Maps, and NDCGs values. And quite similar results can be observed in figures 7, 8, and 9 where regParam = 0.01, MaxIter = 15, and scores of the metrics improve along with the increase in rank.

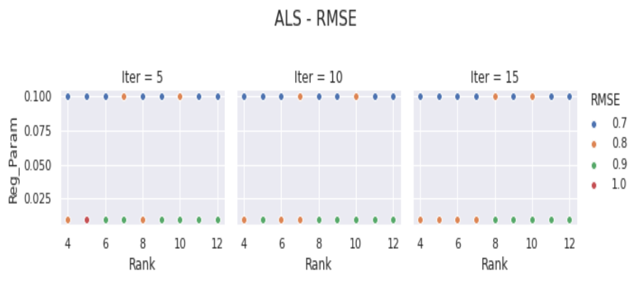


Figure 10: Compound scaling for MaxIter w.r.t Rank and RegParam for RMSE

Similarly, an ideal model should have a lower RMSE score. From figure 10, quite similar results can be observed at regPrama = 0.01, 0.1, and MaxIter = 15 we have lower RMSE.

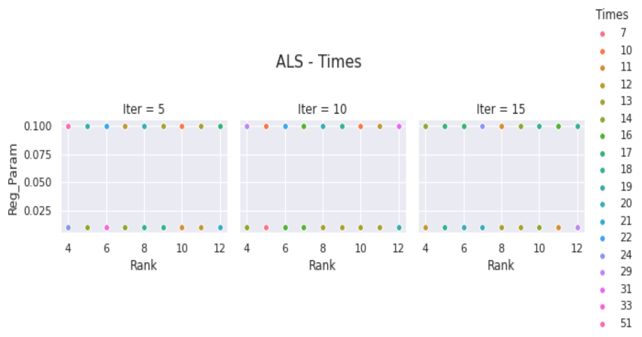


Figure 11: Time tradeoff w.r.t Rank and RegParam

5.2 Model Performance

In this section, we will discuss the performance of each of the recommender system models implemented in our project on multiple evaluation metrics.

From section 5.1, after compound scaling, we have our hyperparameter values as MaxIter = 15, regPram = 0.01, and rank = 12 to get better performance. In further sections, we will show our results for these hyperparameter values only.

5.2.1 Performance of LightFM model on full dataset:

Comparing SparkALS and LightFM on precision at k and time over ranks, regularization parameter.

Table 1: Training and test data results for LightFM model

At regParam = 0.01, MaxIter = 15, rank = 12

Precision (test)	**Accuracy (train)**	**Accuracy (test)**	**Time (ms)**
0.043	0.92	0.89	4

5.2.2 Performance of ALS model on full dataset:

Table 2: Training and test data results for ALS model

At *regParam = 0.01, MaxIter = 15, rank = 12*

Precision	**MAP**	**NDCG**	**Time (ms)**
0.03	0.01	0.11	29

We observe that LightFM has a limitation for large scales of datasets. The transforming step to pandas pivot table and scipy sparse coo_matrix, both contain dimension restrictions. Also, LightFM contains limited built-in evaluation metrics. These factors may also impact choice of tools to use for other systems.

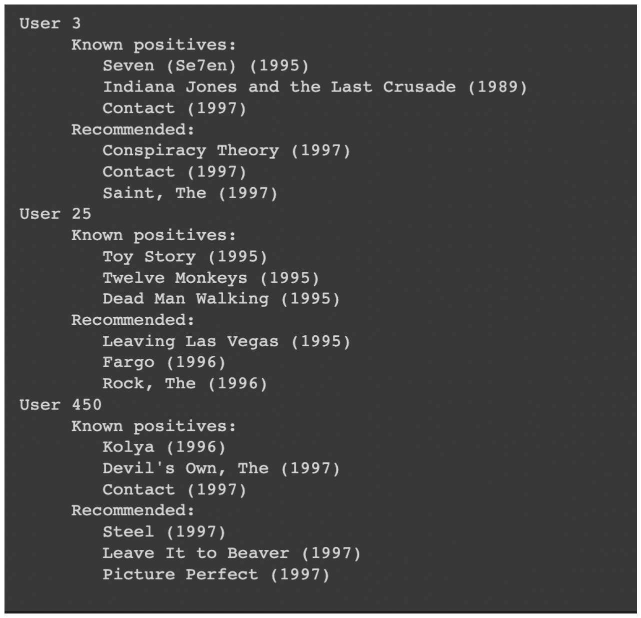
5.2.3 Baseline Popularity Model: The popularity model is a very common (and usually hard-to-beat) baseline approach for recommender systems. It does not generate personalized results for the users but rather recommends to a user the most popular item (or any entity that the model is recommending) that the user has not yet consumed.

Table 3: Popularity baseline model scores on small and full datasets

Dataset version	RMSE scores
Small	0.979
Full	0.963

5.2.4 Annoy (Approximate Nearest Neighbors): It is a C++ library with python bindings to look for points in the space that are close to a given query point. It also creates large read-only file-based data structures that are memory mapped so that many processes may share the same data.

Recommendations without ANN:



```
User 3
     Known positives:
        Seven (Se7en) (1995)
        Indiana Jones and the Last Crusade (1989)
        Contact (1997)
     Recommended:
        Conspiracy Theory (1997)
        Contact (1997)
        Saint, The (1997)
User 25
     Known positives:
        Toy Story (1995)
        Twelve Monkeys (1995)
        Dead Man Walking (1995)
     Recommended:
        Leaving Las Vegas (1995)
        Fargo (1996)
        Rock, The (1996)
User 450
     Known positives:
        Kolya (1996)
        Devil's Own, The (1997)
        Contact (1997)
     Recommended:
        Steel (1997)
        Leave It to Beaver (1997)
        Picture Perfect (1997)
```

Figure 12: User recommendations without ANN (Brute Force)

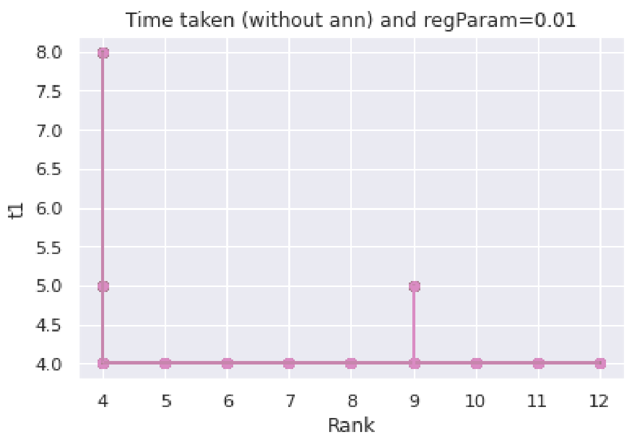
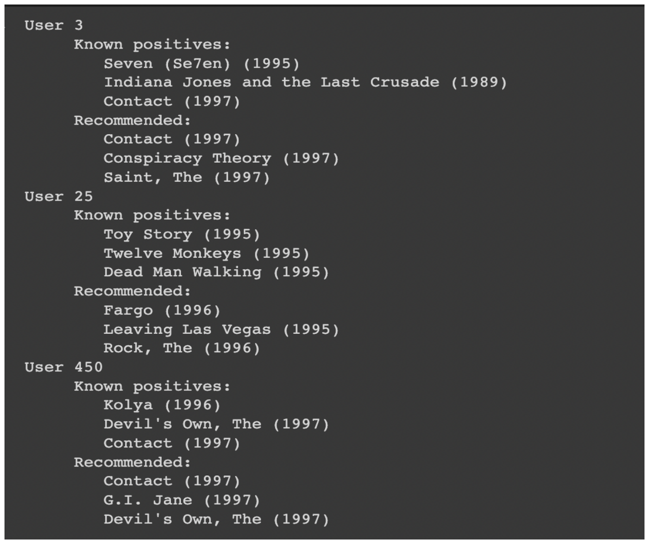


Figure 13: Time taken for user recommendations without ANN (Brute Force)

Recommendations with ANN:



```
User 3
     Known positives:
        Seven (Se7en) (1995)
        Indiana Jones and the Last Crusade (1989)
        Contact (1997)
     Recommended:
        Contact (1997)
        Conspiracy Theory (1997)
        Saint, The (1997)
User 25
     Known positives:
        Toy Story (1995)
        Twelve Monkeys (1995)
        Dead Man Walking (1995)
     Recommended:
        Fargo (1996)
        Leaving Las Vegas (1995)
        Rock, The (1996)
User 450
     Known positives:
        Kolya (1996)
        Devil's Own, The (1997)
        Contact (1997)
     Recommended:
        Contact (1997)
        G.I. Jane (1997)
        Devil's Own, The (1997)
```

Figure 14: User recommendations with ANN

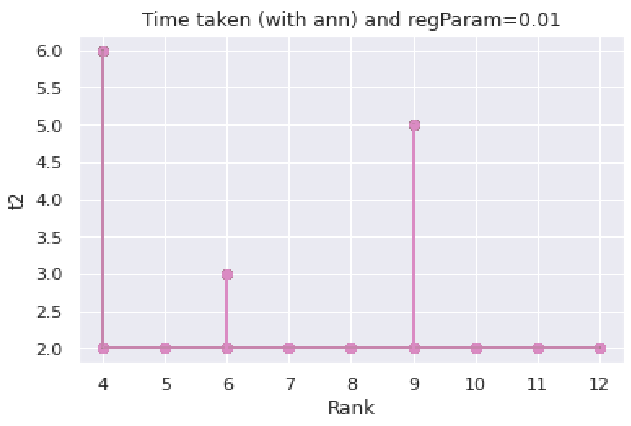


Figure 15: Time taken for user recommendations with ANN

Best time taken with and without ann:

Table 4: Time taken by both Annoy models

Model	**Time Taken (best of 5)**
Without Ann	5.24ms per loop
With Ann	2.93ms per loop

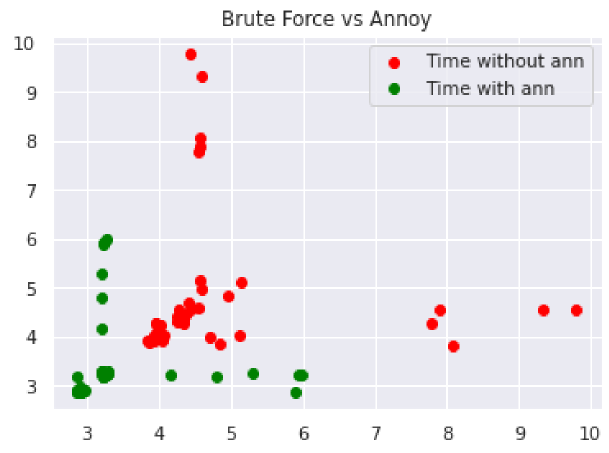


Figure 16: Comparison between Brute force and ANN models

As we can see from figure 16 and table 4, the time taken with ann is much lesser than the time taken for models without ann. Models with ann provide better performance because it follows an approach where an approximate nearest neighbor is almost as good as the exact one. Particularly, if the measurement of the distance accurately captures the notion of user quality, then the small differences in the distance will not matter. Additionally, the biggest factor that sets ANNOY apart from the other models is that it uses static files as indexes. This means that we can share these indexes across multiple processes. Also since creation of the indexes is decoupled with loading them, we can pass around these indexes as files and can also be mapped into the memory quickly. All of these reasons combined give a better performance with using ANNOY in the recommender systems.

6 Contribution and Collaboration

- **Data processing and splitting:** Disha Lamba, Sai Kiran Challa
- **Baseline model and tuning:** Sai Kiran Challa
- **Model extension:** Disha Lamba, Prakriti Sharma, Sai Kiran Challa
- **EDA:** Disha Lamba
- **Model comparison and visualizations:** Prakriti Sharma
- **Final report:** Disha Lamba, Prakriti Sharma, Sai Kiran Challa

7 References

[1]https://making.lyst.com/lightfm/docs/examples/warp_loss.html
[2] https://spark.apache.org/docs/latest/mllib-collaborative-filtering.html
[3]https://www.elenacuoco.com/2016/12/22/alternating-least-squares-als-spark-ml/