Experimental Log and report findings

Intro to collaborative filtering – model based.

Collaborative Filtering – Model Based looks at the user-item interactions where users and items representations have to be learned from interactions matrix. Assumes a latent interactions model that needs to learn both users and items representations from scratch. High bias, low variance.

Model based collaborative approaches only rely on user-item interactions information and assume a latent model supposed to explain these interactions. For example, matrix factorization algorithms consists in decomposing the huge and sparse user-item interaction matrix into a product of two smaller and dense matrices: a user-factor matrix (containing users representations) that multiplies a factor-item matrix (containing items representations).

**Surprise library**:

Data:

Using the surprise library for python does not require much data preparation. The needed data is user, movie and rating. For the smaller dataset, 100,000 ratings by 943 users on 1682 items, each user has rated at least 20 movies. For the bigger dataset, 1,000,000 ratings, 6040 users, on 3952 items, ratings are made on a 5-star scale (whole-star ratings only), each user has at least 20 ratings.

Method/preliminary experiments:

The first attempt was to run the codes using the whole dataset and it took too much time. So I stopped the process and did it in increasing samples hoping to find a peak based on the lowest RMSE, the metrics chosen to evaluate the model. The results was consistent with what was suggested. Being a non-parametric, the bigger the dataset, the lower the RMSE and thus the better the algorithm performed. [see appendix 301]

The second attempt was the comparison across models. Surprise library has multiple input models. Also, since this is only for model comparison, instead of using our original dataset of 270k users, I have decided to use the smaller dataset because it requires less resources and takes a shorter time.

Looking at the results [Appendix 302], all the algo perform better than the the normal predictor. For the surprise library, the normal predictor is an algorithm predicting a random rating based on the distribution of the training set, which is assumed to be normal. T

SVD++ model has the best results but also the longest time. To have a balance between time used and minimal RMSE, I have added a new efficiency calculation which is time\*RMSE. The time taken, adds a weight to the RMSE, which looking at the efficiency figure, although the RMSE has a good result, the time taken to run the algorithm is too long.

This is probably because SVD++, the extension of SVD, adds a factor vector for each item and these item factors are used to describe the characteristics of the item. [See Appendix 303 for formula]. It takes into account the user preference so that a better user bias can be obtained.

The models with the next best results are SVD and KNNBaseline. Also, their efficiency is significantly better than SVD++.

The third attempt was to combine the 2 experiments that we have done. This time, a bigger dataset is used – this should improve the RMSE of all the models. And the results reduce the RMSE further which shows a more accurate model but it takes a lot more time. Also notice that with a bigger dataset, the difference between SVD and KNNBaseline is not clear. SVD or SVD++ outperforms any other model in this case.

If we ran an RMSE and efficiency comparision, we notice that the RMSE for SVD has the most improvement and the NMF has the least weighted resources consumed. Since ultimately our model aims for the lowest RMSE, the extra time taken to run is negligible and the weighted factor does not increase exponentially, we will look into SVD model.

Briefly, NMF works like SVD

Fourth, to look for the peaking point of using SVD model. To do this, we will attempt to plot a chart with n-samples against RMSE. This will help to look for the dimishing/increasing return of the model as more data is added. We plot a chart of Datasize VS RSME to get a visual look of the relationship between the 2 [see appendix 305]. From the chart, we can see that there is a diminishing return in adding more data. Due to the intensive computing power needed, we are unable to determine the point to stop.

Next steps:

There are no easy way out of this issue. Like seen above, the way to reduce the RMSE and increasing the reliability of model would be to add more data. This might require cloud computing resources and more money. Another way is to find other libraries which can handle bigger data more efficiently. This will be kept in view for further exploration.

**Singular Value Decomposition (SVD) using Tensorflow**:

Like discussed earlier, SVD provides the best results although it requires more resources. Also, it is the award winning method for the recommended systems from Netflix one million challenge. So I am going to experiment another SVD with another library.

SVD constructs a matrix with the row of users and columns of items and the elements are given by the users’ ratings. Singular value decomposition decomposes a matrix into three other matrices and extracts the factors from the factorization of a high-level (user-item-rating) matrix. Model is in A = USVT. U is the characteristics of the users, VT is the inverse characteristics of the items and S the diagonal matrix is the correlation of latent factors.

Data:

We are going to use the same set of data from movielens. This time round, we are going adjust the amount of data as we progress. Noting the differences in RMSE as the model decreases.

Method/Preliminary experiments:

Singular Value Decomposition (SVD). SVD reduces the dimensionality by constructing a matrix with the row of users and columns of items and the elements are given by the users’ ratings. It is similar to PCA but it works on better on sparse matrix. In this method, the library used is Tensorflow. Tensorflow works by creating relational graphs. The underlying code is ran in C++ while python is used to direct the flow of the traffic. Like previously with Surprise database, as the number of sample increases, the RSME decrease. At 40 epoch, and n=3000, we got to an RMSE of 1.80, at n=10000, 0.619 and at n=100,000, it is at 0.8. [See appendix 307-309]. Not sure why this is happening.

Next steps:

Study why Tensorflow is weaker at higher n. There are hyper-parameters tuning to look at, including Batch size, learning rate, and regularization term.

**Truncated SVD with Gradient Boosting**

The difference between SVD and truncated SVD is given an n x n matrix, truncated SVD generates the matrices with the specified number of columns, whereas SVD outputs n columns of matrices.

Data:

From previous experiments, we have determined that the datasize matters. However attempted to change the code fail.

Method/Preliminary experiments:

On the first run, we ran a modest sample of 3000. It gives the results in Appendix 307. However, when the datasize was changed to 10000, an error occurred.

Next steps:

We will focus on Surprise library.

**Surprise library Part 2**:

Data:

Like mentioned previously, the more data we use, the better the results get – RMSE and MAE both reduce as sample size increase. (See Appendix 311). For hyperparameter tuning, we used the sample size of 1,000,000. This is because using the full dataset caused it to crash (See appendix 312).

Setting *shuffle=True*. One thing is our data is sorted by UserId. Setting *shuffle=True* helps to shuffle the UserId. Difference is shown in appendix 319. Notice if *shuffle=False*, the number of user add back to the original data. This is because our data is sort by user.

Surprise library has their own train\_test\_split. The testset is output as array while the trainset is output as surprise.dataset. The surprise dataset has an inner id which makes the library function better. This makes data manipulation harder when we want to do our manual testing. There are a lot of challenges in manipulating the data in the surprise library when we want to implement our custom test.

The reason why sample size is important (see appendix 341). As the sample increases, the algo learns more about the latent factors and error is reduced. The final dataset our predictions is ran on has 26,024,289 ratings, 270,896 users and 45,115 movies. A train\_test\_split is applied - 70% trainset, 30% testset, random\_state 42.

Attempted to show that as sample size increase, the top n predicted changes. [See Individualresults1.csv] [For full results see results.csv]. The first run of this was not successful [partial results in rmse reduction.csv]

Method/preliminary experiments:

An attempt was made to show how the best prediction changes as sample increases (code can be found in MLProject\_Surprise\_SVD\_HPtuning.ipynb. As the sample increases, the algo learns more about the user and more items is predicted.

Hyperparameter tuning to minimize the error. The model for minimizing error for SVD in Surprise library is shown in appendix 313. There are 4 keys hyperparameters:  
n\_epochs = the number of times the SGD procedure is iterated.   
lr\_all = the learning rate. The learning rate determine how fast the algo moves from 1 epoch to the next.  
n\_factors = the number of factors in the matrix.  
reg\_all = regularization factor. Higher means higher variance of predictions

In surprise library, there is a built in GridSearchCV which allows different sets of hyperparameters to be ran in a cross validation method and it will return the best model based the parameters with the least error. In addition, it has a hyperparameter “n\_jobs” which allows all CPUs to be used in the computation when set to “-1”. The use of multiple CPU is shown in appendix 314.

There is a difference between cross validation and training the model. In CV, the data is one set, splited and part of it is used for validation. In this sense, there is no segregation between the test and training set. However, when we are training the model, the train and test set is completely separated.

We ran a few variations (appendix 315) and took the hyperparameters of the best model and put it fit it into the full dataset using the trainset/testset but the values generated are worse than the default setting. (appendix 316-318). One reason is due to the segregation of the train and test set. CV is perfected on itself but changes happen when a completely different set of data is applied. Another reason we can think of is the parameters are not tuned to the smaller dataset used in the tuning phase. A third reason and it is highly probably which i thought of after the presentation is that we use 1,000,000 for gridsearchCV. Our dataset is 270k users and 45k movies. This means that the 1million cannot represent our full dataset because the chance of having repeated user is 27/100 and movies being 45/100. This does not include the permutations of user and movies yet. Given this factor, our 1mil sample data will not be sufficient to represent our 26mil full dataset. As shown in appendix 318, increasing the hyperparameter of “n\_factors” from 80 to 120 actually increases the model performance.

To execute better fine-tuning of hyperparameters, we will try running the code on colab with GPU on. But sadly, it crashed. (Appendix 320) So, we will attempt to tune it in Jupyter notebook.

MAE vs RMSE. As the variance increases, the difference between MAE and RMSE will increase (appendix 321). So we want to get a balance of it. Notice in Appendix 319 and 322, although the RMSE decrease when reg\_all is set to 0.04 from 0.02, the MAE increases. This means the uncertainty of the predictions increases. Looking at appendix 319, 321 and 322, the reg\_all has the best performance at 0.03.

As we test other parameters, the higher the numbers, the better the test results. But at some point, it will decrease like in the case of the reg\_all. So we will attempt to continue testing with limited permutations since like mentioned in Appendix 312, the PC crashed. Right now, we are trying to find the point where given our dataset, the point where increase a hyperparameters does not cause a decrease in performance.

After multiple testing and days and hours of manually changing parameters because of PC RAM limitations issue, we have arrived at a good model. Like the gridsearchCV, every parameter will reach a point where it no longers add value. Our final SVD model has a MAE of 0.5996 and RMSE of 0.7934. Appendix 330 shows how we derive at the hyperparameters.

From this model, we will generate the predictions based on the highest estimated rated movie unique to each individual. Example shown on Appendix 331 [fullset in nprecommendedmovies.csv] [individual results in np\_prediction\_final.csv]

I attempted to confirm if the poor tuning is caused by training data on itself but the program ran out of memory again. (appendix 340)

**Predictions:**

Recommender systems ultimately predict a set of movies that users want to watch. Therefore, I built a function on top of the predicted results to look at the hit rate of the results. Accuracy score is the number of correct predictions in the actual prediction. Since this is based on historical data, we are going to assume that users and movies with ratings are what they have watched. Looking at the results (appendix 342), it is good but not convincing because of the model.

Prediction test 1: Recommender systems ultimately predict a set of movies that users want to watch. Therefore, I built a function on top of the predicted results to look at the hit rate of the results. Accuracy score is the number of correct predictions in the actual prediction. Since this is based on historical data, we are going to assume that users and movies with ratings are what they have watched. Looking at the results (appendix 342), it is good but not convincing because of the model. Reason being that we are asking the model to pick 10 out of a sample in the testset, not the full movie.

Prediction test 2: Given time constraints, we did another test on the prediction results. This time, we predict the top 10 movies for 1 user, userId 101382, by generating an estimating rating for all movies and then compare it to the top rated movies for the same user. This user has 48 5 stars rating and 8 of our movies are in the list of 48, giving us an accuracy of 80%. Did it on another user, 270893, and it gives 40%. This is a huge variance given only 2 samples.

Prediction test 3: Previously we mentioned that we want to recommend movies that the user has not watched. To do this test, we assume that the training set is what the movies have watched and the testset is what they would watch later. So we apply the model on the movies that the user have not watched [total movie set - movies watched by user in trainset] and compare the result against the testset. Sadly, user 270893 shows a result of 0. (appendix 343), 101382 = 0.4, 70284 = 0.0, 69691 = 0.2. A further test was done on 100 users and the result is 8.6%. This is a meaningful result because the predicted movie list is not in the train set.

**Future work:**

Like in the 3 prediction tests, there were flaws in each other and with brainstorm, it became better. We can look for a better measure of accuracy. Another work to be done is to introduce a decay to the ratings because of preference changes. Plus, when we were trying out the tensorflow, the RMSE did go down to 0.62 which is better than what we currently have. This is worth a study as well.

**Conclusion**

We attempt to hire and train the salesman with a logical approach. Firstly, which model is the best and yet does not cost a bomb. Secondly, how do we best train this salesman(model) we hired. More data? Better learning rate? More factors? This is the model that we have with the hyperparameters tuned.

While collaborative model based is effective and we can see its effectiveness, the resource needed to run the program is huge. Furthermore, there is a lot of tuning to be done. In this project, we typically look at reducing the RMSE because it is the metric to gauge if the model is good. However, there are a lot of other metrics to look at like MAE which was used briefly. Given more resources, running gridsearchCV automatically is definitely a plus. Given more time, using Tensorflow seems to be faster but this has yet to be tested. Also, there are other more things to study like prediction accuracy and preference changes by incorporating time decay.

References

[301]"Recommender System — singular value decomposition (SVD) & truncated SVD", Medium, 2021. [Online]. Available: https://towardsdatascience.com/recommender-system-singular-value-decomposition-svd-truncated-svd-97096338f361. [Accessed: 24- Aug- 2021].

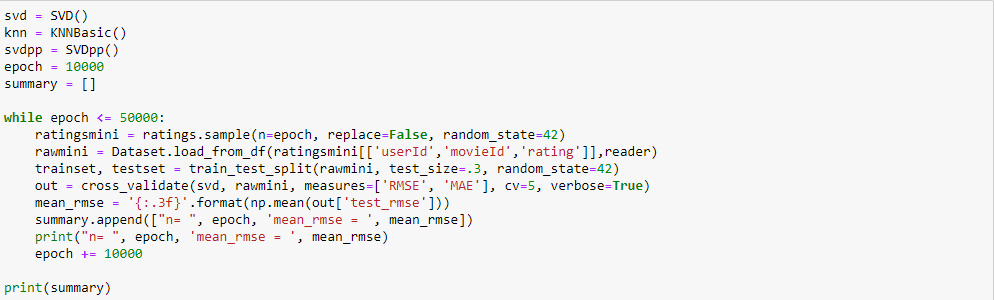
[302] Z. Xian, 2021. [Online]. Available: https://www.hindawi.com/journals/mpe/2017/1975719/. [Accessed: 24- Aug- 2021].

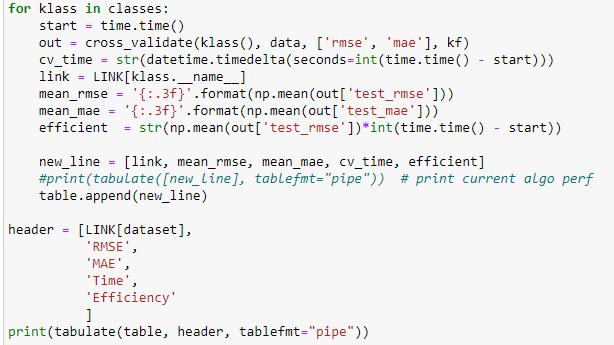
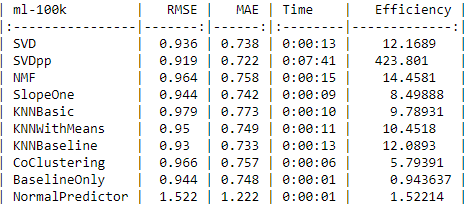
[303]"Welcome to Surprise’ documentation! — Surprise 1 documentation", Surprise.readthedocs.io, 2021. [Online]. Available: https://surprise.readthedocs.io/en/stable/index.html. [Accessed: 24- Aug- 2021].

[304]"Matrix Factorization-based algorithms — Surprise 1 documentation", Surprise.readthedocs.io, 2021. [Online]. Available: https://surprise.readthedocs.io/en/stable/matrix\_factorization.html#unbiased-note. [Accessed: 24- Aug- 2021].

[305] "Matrix Factorization-based algorithms — Surprise 1 documentation", Surprise.readthedocs.io, 2021. [Online]. Available: https://surprise.readthedocs.io/en/stable/matrix\_factorization.html#unbiased-note. [Accessed: 24- Aug- 2021].

[306]D. Kumar, "Singular Value Decomposition (SVD) & Its Application In Recommender System", Analytics India Magazine, 2021. [Online]. Available: https://analyticsindiamag.com/singular-value-decomposition-svd-application-recommender-system/. [Accessed: 24- Aug- 2021].

Appendix 301 - Surprise dataset size results  
  

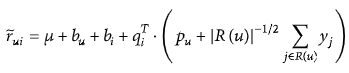

Appendix 302 - Surprise model results 100k dataset  
  
  


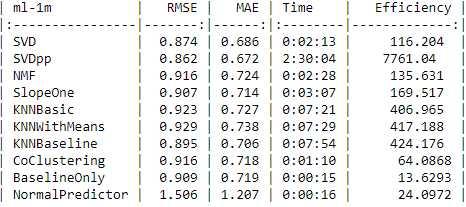
Appendix 303 – Surprise formula SVD vs SVD++

SVD

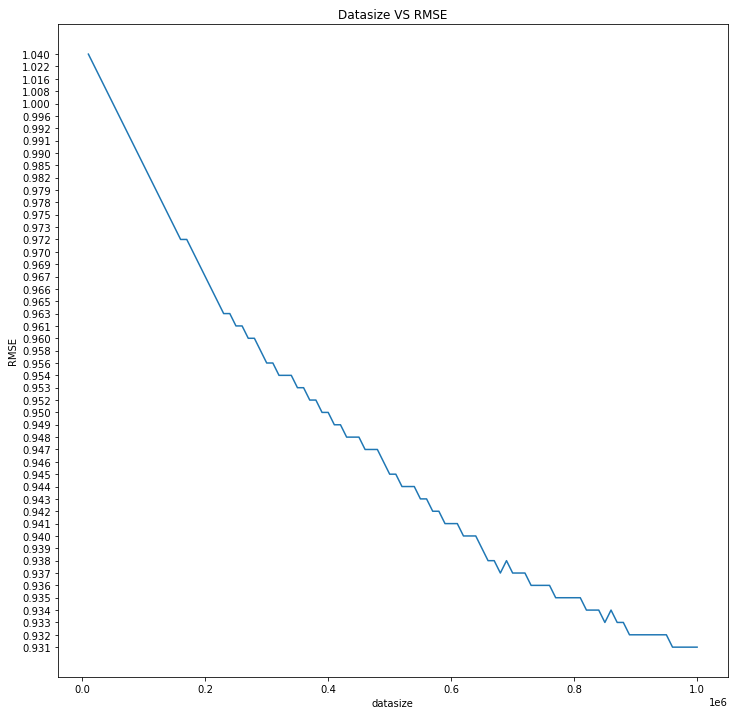


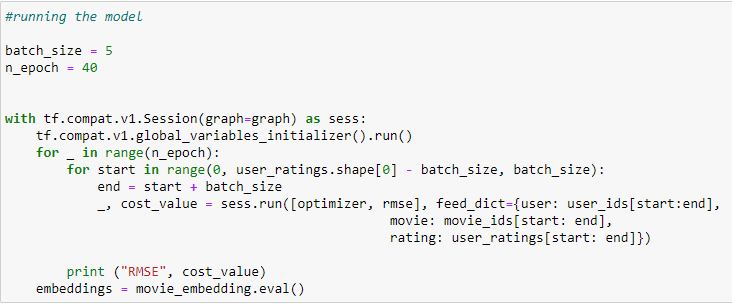
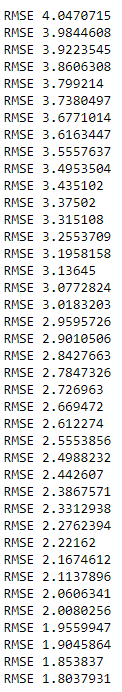
SVD++

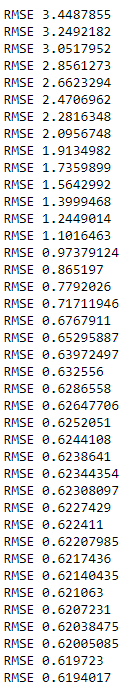


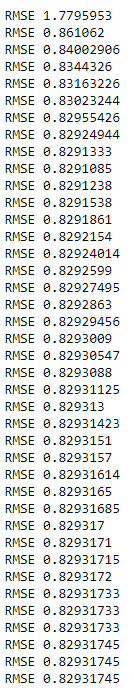
Appendix 304 – Surprise model results with 1m dataset  
  


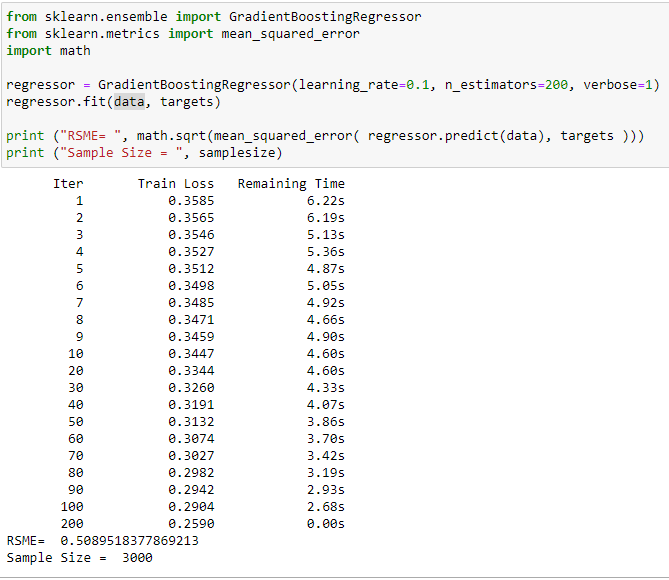
Appendix 305 – Comparison of 100k and 1m  

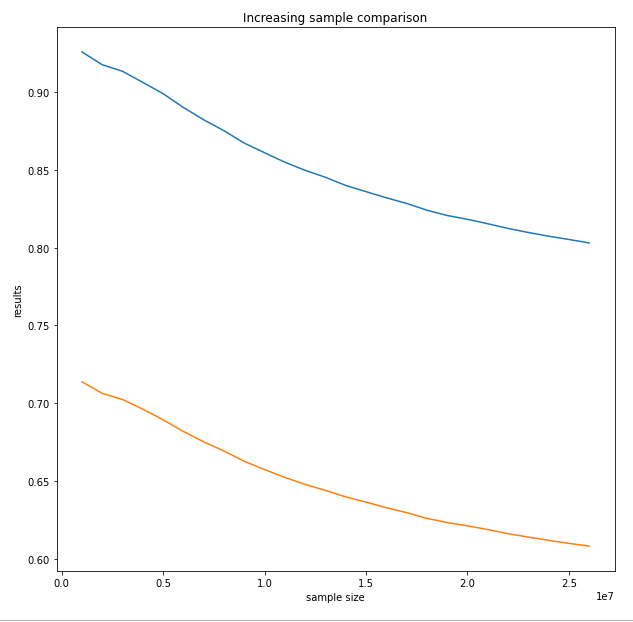

Appendix 306 – Chart of Datasize vs RMSE  


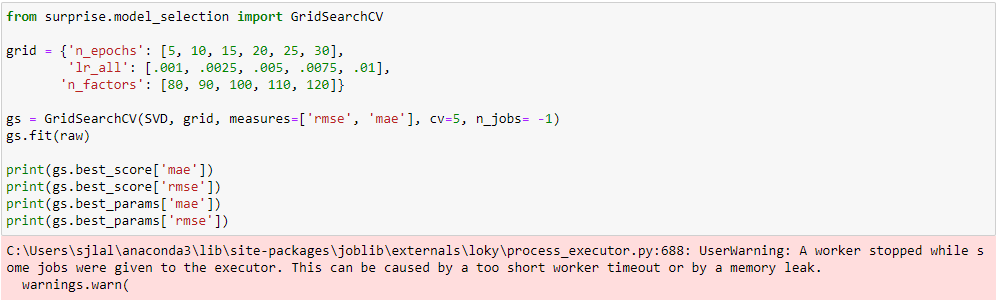
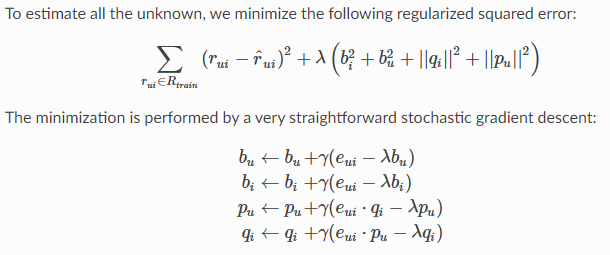
Appendix 307 – SVD with Tensorflow n=3000  
  


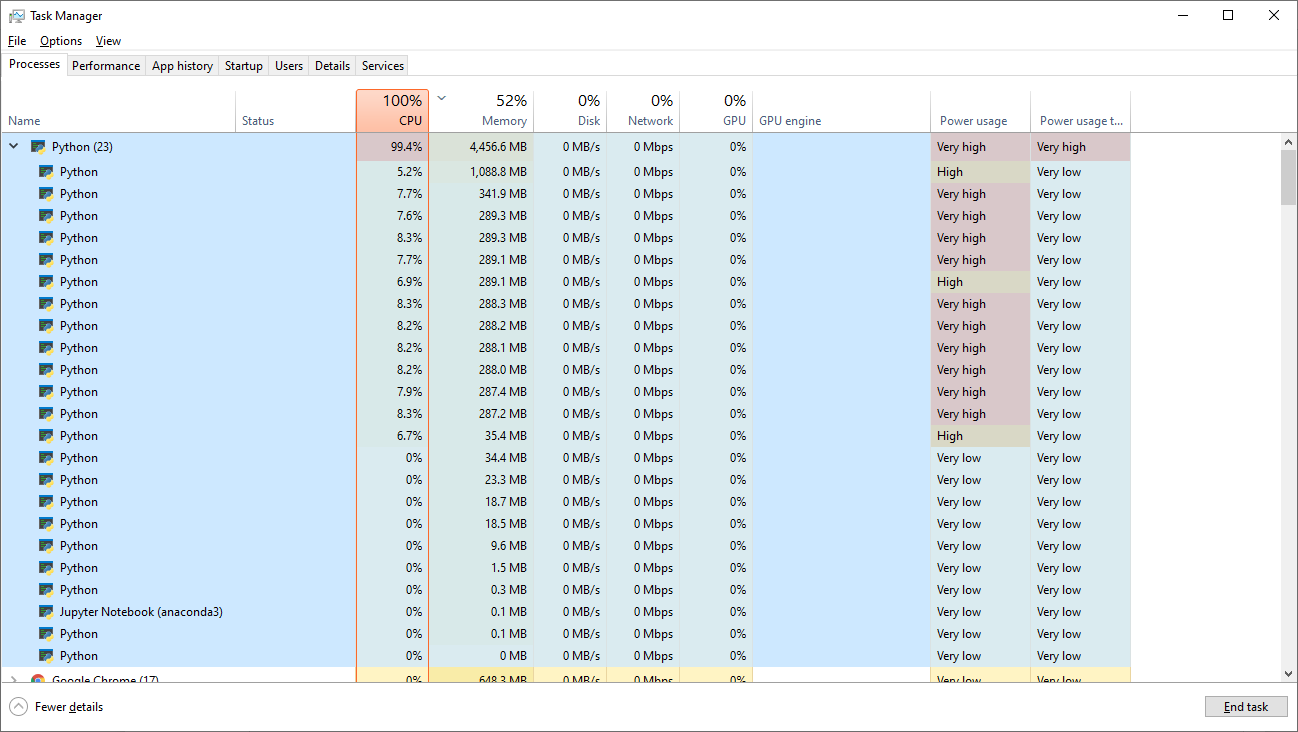
Appendix 308 – SVD with Tensorflow n=10000  


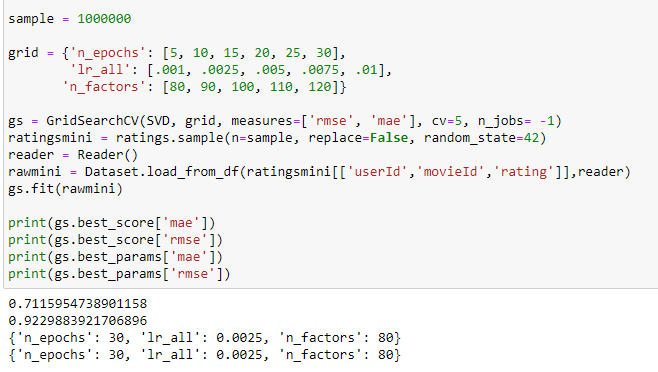
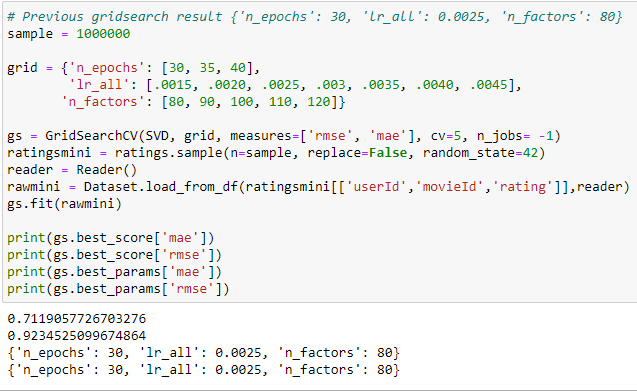
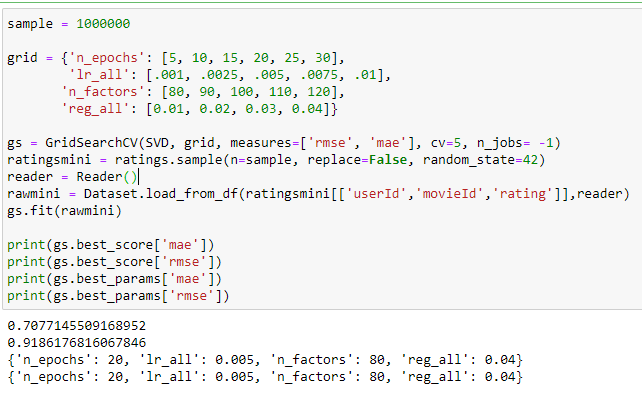
Appendix 309 – SVD with Tensorflow n=100000  


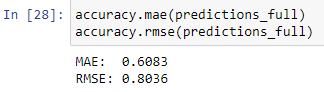
Appendix 310 – Truncated SVD with GB n=3000  


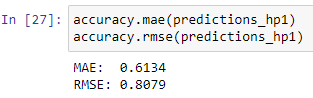
Appendix 311 – Surprise increasing sample - RMSE, MAE  


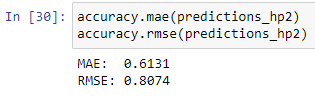
Appendix 312 – Gridsearch crash (*raw* is the full dataset of 27,000,000 ratings)  
  
Appendix 313 – Surprise library SVD hyperparameter tuning  


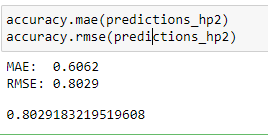
Appendix 314 – surprise n\_jobs = -1 (Task Manager)  


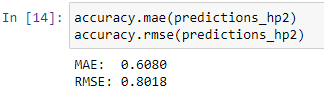
Appendix 315 – GridSearchCV results  
  
  


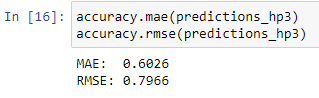
Appendix 316 – SVD with default settings n\_epochs = 20, lr\_all= 0.005, n\_factors = 100, reg\_all=0.02  


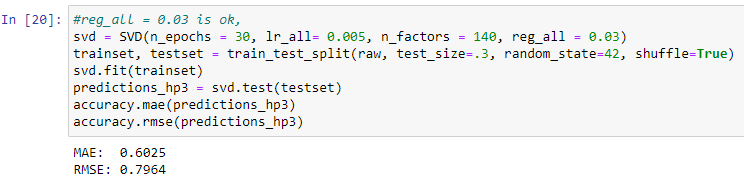
Appendix 317 – SVD with n\_epochs = 30, lr\_all= 0.0025, n\_factors = 80, reg\_all=0.02  


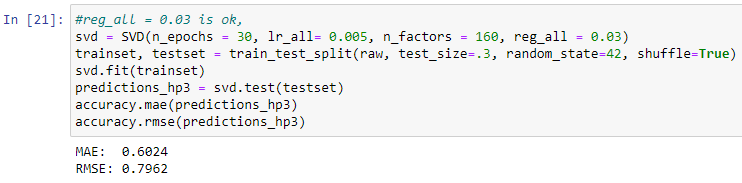
Appendix 318 – SVD with n\_epochs = 30, lr\_all= 0.0025, n\_factors = 120, reg\_all=0.02  


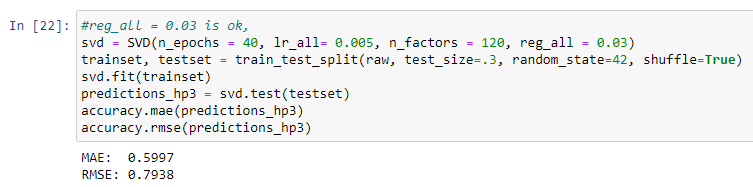
Appendix 319 – SVD with n\_epochs = 30, lr\_all= 0.005, n\_factors = 120, reg\_all=0.02  


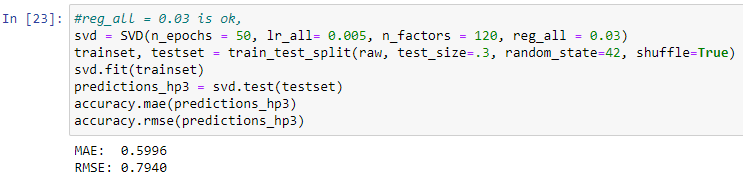
Appendix 321 – SVD with n\_epochs = 30, lr\_all= 0.005, n\_factors = 120, reg\_all=0.04  


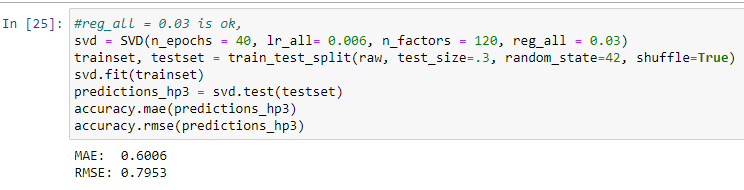
Appendix 322 - SVD with n\_epochs = 30, lr\_all= 0.005, n\_factors = 120, reg\_all=0.03  


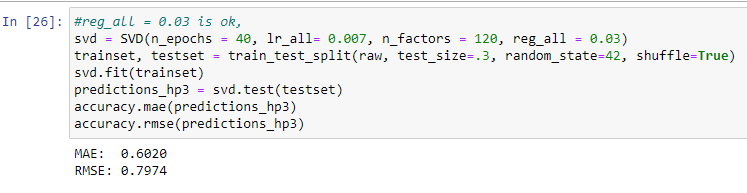
Appendix 324 - SVD with n\_epochs = 30, lr\_all= 0.005, n\_factors = 140, reg\_all=0.03  


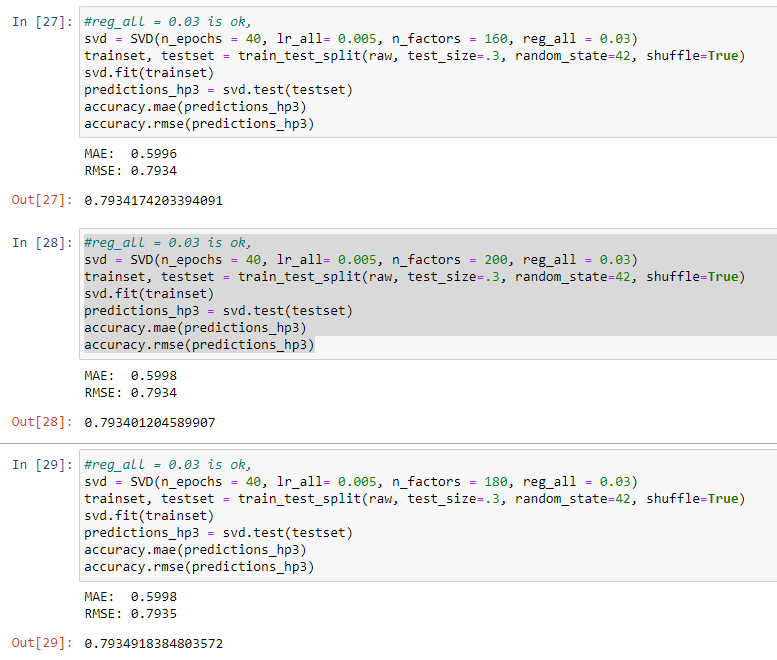
Appendix 324 - SVD with n\_epochs = 30, lr\_all= 0.005, n\_factors = 160, reg\_all=0.03  


Appendix 324 - SVD with n\_epochs = 40, lr\_all= 0.005, n\_factors = 120, reg\_all=0.03  


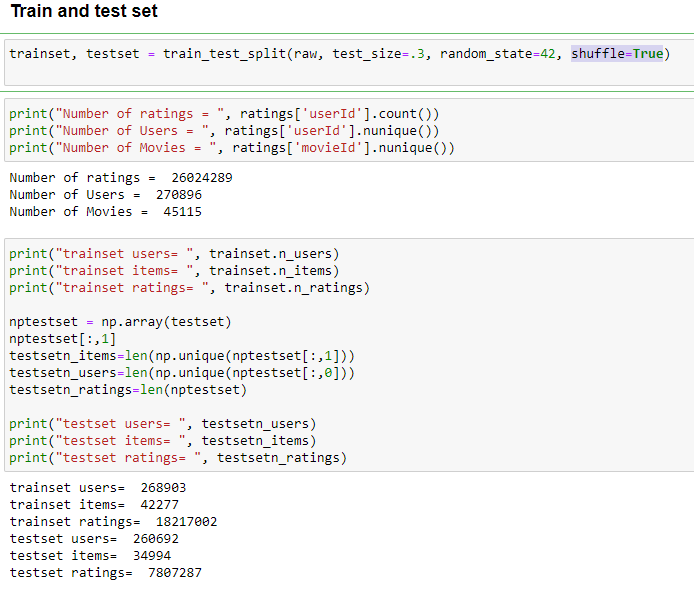
Appendix 324 - SVD with n\_epochs = 50, lr\_all= 0.005, n\_factors = 120, reg\_all=0.03  


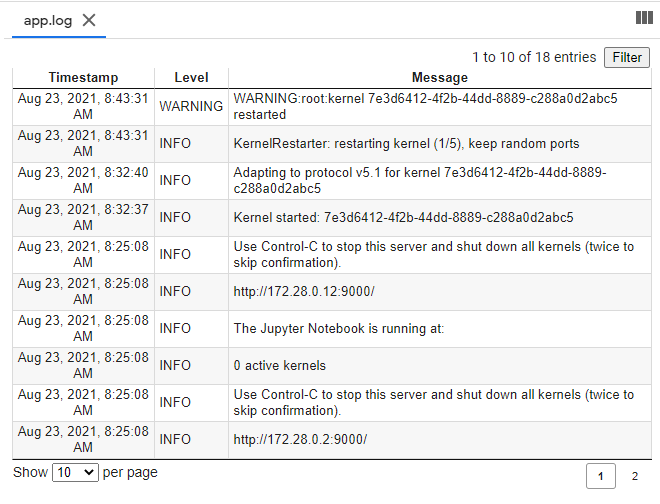
Appendix 324 - SVD with n\_epochs = 40, lr\_all= 0.006, n\_factors = 120, reg\_all=0.03  


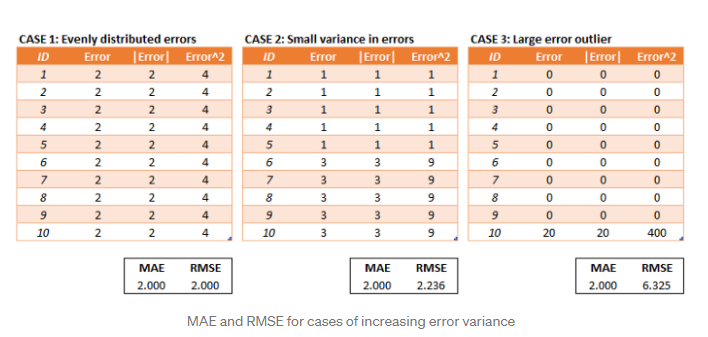
Appendix 324 - SVD with n\_epochs = 40, lr\_all= 0.007, n\_factors = 120, reg\_all=0.03  


Appendix 323 – SVD with n\_epochs = 40, lr\_all= 0.007, n\_factors = X, reg\_all=0.03. Looking for the best n\_factors  


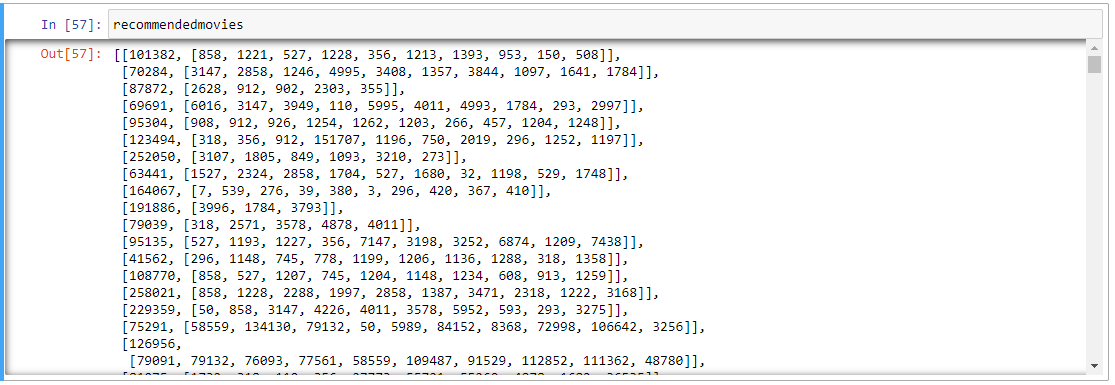
Appendix 333 – GridsearchCV crash  

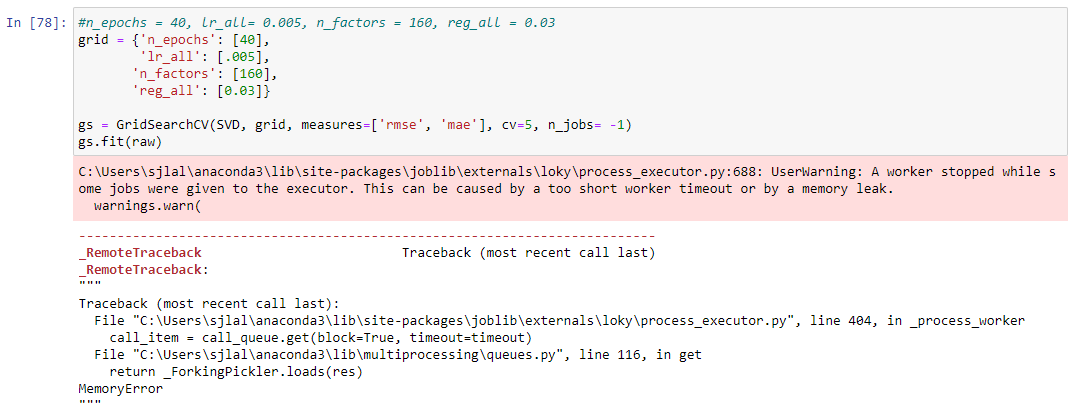

Appendix 319 – Shuffle=True  


Appendix 319 – Shuffle=False  
  
  
Appendix 320 – colab crashed  


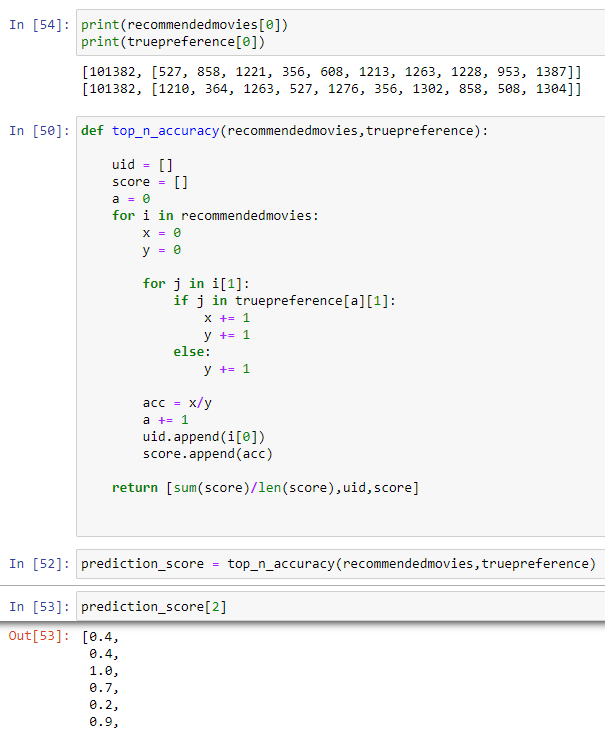
Appendix 322 - MAE vs RMSE  


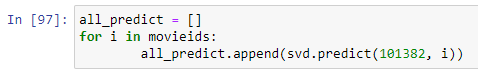
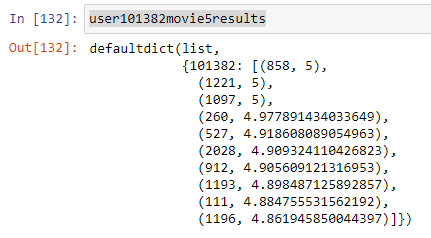
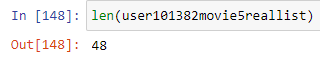
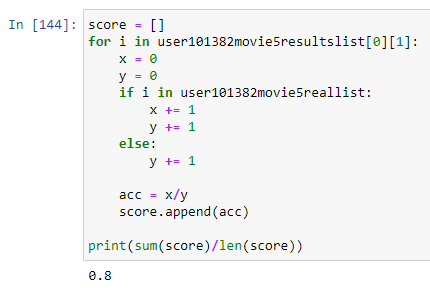
Appendix 330 – SVD manual testing summary and conclusion  


Appendix 331 – predictions  


Appendix 340 – tried to confirm poor tuning from gridsearchcv  


Appendix 341 - samplesize and predictions  


Appendix 342 – prediction test 1 Hit rate how many of the testset is accurately picked  


Appendix 343 – prediction test 2 Hit rate how many movies from the model is in the top rated of 1 user  
  
  
  


Appendix 344 – Prediction test 3 Hit rate the predicted movie list is not in the trainset  
