# Networked Life Project

Gao Yuze 1002971
Koh Chuan Shen Marcus 1002764
Zhao Nailin 1003003
Zhou Zhi 1002641

Team: SUccess

April 27, 2020

Gao Yuze 1002971
Koh Chuan Shen Marcus 1002764
Zhao Nailin 1003003
Zhou Zhi 1002641

# 1 Introduction

This project aims to create a prediction algorithm to estimate the user rating for a movie in a Netflix setting.
The algorithms used are linear regression and Restricted Boltzman Machine with extensions.

# 2 Linear Regression

This algorithm is a simple linear algebra solution for the problem. Solving for the bias vector (b) that minise root mean square error (RMSE) in:

$$b = (A^T A + \lambda I)^{-1} A^T c$$

where:
c is the user ratings
A is the a binary matrix indicating if the user have rated a movie

## Question 1.1

With no regularisation, we obtain a training RMSE of 0.883550 and a validation RMSE of 1.031098

## Question 1.2

Trying for $\lambda$ between 0 and 30, the following RMSE result is obtained:

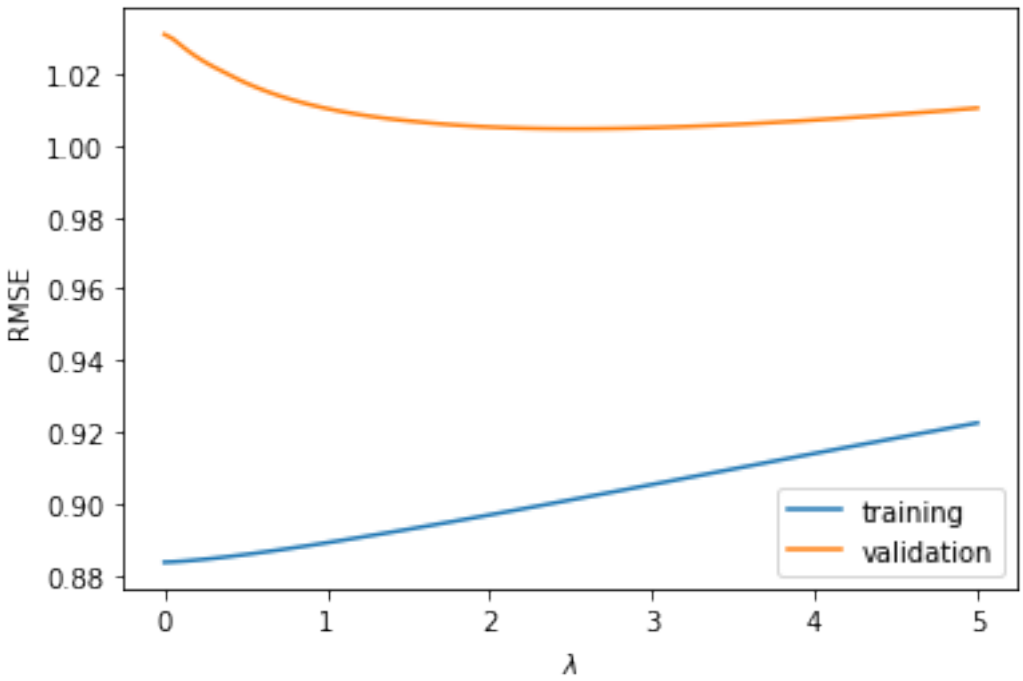


Figure 1: Training and Validation error

Using the argmin() function, it is found that the lowest validation error occurred when $\lambda$ = 2.586
As such With regularisation term = 2.586 , we obtain a training RMSE of 0.901717 and a validation RMSE of 1.00476

Gao Yuze 1002971
Koh Chuan Shen Marcus 1002764
Zhao Nailin 1003003
Zhou Zhi 1002641

# 3 Basic RBM

For the Baisc RBM, the mainRBM.py remain changed for training.
The main weight structure is m by F by K, where m is the number of visible parameters, F is the number of hidden parameters, and K is the number of ratings (5 in this problem)

**For Question 2.1 to 2.5**
In the rmb.py,the following functions are implemented to enable the process, **please refer to the attached python file for detailed implementation.**
Here are a brief description of our approach for each functions.

## sig(x)

A simple sigmoid function of $\frac{1}{1+e^{-x}}$

## visibleToHidden(v,w,bias=None)

Calculating $P(h_i^k = 1|v) = \sigma\left(\sum_{k\in K}\sum_{i\in m} v_i^k W_{ij}^k\right)$
Two for loops are used for the simple implementation for this multiplication.
A possible bias (b) is added for further extension as $\sigma\left(\sum_{k\in K}\sum_{i\in m} v_i^k W_{ij}^k + b\right)$

## hiddenToVisible(h,w,bias=None)

Calculating $P(v_i^k = 1|h) = \sigma\left(\sum_{j\in J} h_j W_{ij}^k\right)$
The function of np.tensordot is used here for faster summation of the element wise multiplication of the two matrix.
A possible bias (b) is added for further extension as $\sigma\left(\sum_{j\in J} h_j W_{ij}^k + b\right)$

## getPredictedDistribution(v, w, wq)

This function does the forward propagation from visible to hidden layer, then randomly sample a few hidden layers before backward propagate to the visible layer for prediction.

## predictRatingMax(ratingDistribution)

This function finds the rating with the maximum predicted probability. This is done through simple argmax.

## predictRatingMean(ratingDistribution)

This function finds the mean of predicted ratings as $\sum xp(x)$

## predictRatingExp(ratingDistribution)

This function finds the mean of softmax of normalised predicted ratings as $\sum x\frac{e^{p(x)}}{e}$

## predictForUser(user, W, training, predictType="exp")

This function is modified from the predict() function for a single user.

## Results

Using the edited rbm.py and setting the following parameters:
Number of hidden layers : F = 10
epochs = 10
gradientLearningRate = 0.01

Gao Yuze 1002971
Koh Chuan Shen Marcus 1002764
Zhao Nailin 1003003
Zhou Zhi 1002641

The validation error after 10 epochs are 1.21 and this is set as the benchmark for future Extension.

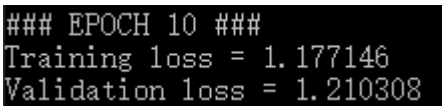


Figure 2: Baseline RBM Result

# 4 RBM Extension

Using the above baseline model, the following extensions are implemented in mainRBM.py for better performance

## Momentum

Adding momentum prevented the weights from oscillating especially when training with more stochastic smaller batches.
The momentum is added by considering a sizable portion of the past gradient in the new gradient for the weight propagation.
This is done by a simply change of 1 line of code

$$W+ = grad \qquad (old)$$
$$W+ = grad + momentum * prevGrad \qquad (new)$$

## Adaptive Learning Rate

Adding Adaptive Learning Rate allows the model to converge quickly and better refinement of the model at later epoches.
There are many choices of function for adaptive learning rate, such as exponential or power law. We opted for a power law function with k = 2, this allows a quick descent in the beginning, and non-vanishing learning rate at the end.
Thus, the adaptive learning rate is set to be $\frac{\alpha}{(epochs\_idx)^2}$ where $\alpha$ is a hyper-parameter to be tuned.

## Early Stopping

Due to the stochastic nature of RMB sample and our gradient descent approach, the final epoch might not produce the best solution.
In fact, as the epoch increase, the risk of overfitting increases as well.

Thus we keep track of the weights associated with the smallest validation RMSE during the training process and use it as the output weights for validation and testing.

## Regularisation

In the model, we have around 6 to 10 hidden parameters (depends on setting), 300 visible parameters and 5 rating layers. This leads to the weight matrix having around $\sim 10^4$ elements.
With only 300 rows in each of the training, validating and testing set, using over $10^4$ parameters could easily overfit the model.
Thus we added a regularisation term $\lambda$ when we update the gradient in each epoch or mini batch epoch, to reduce the magnitude of each parameters and prevent over training as following:

$$W \leftarrow W + \alpha\left((PG - NG) - \lambda W\right)$$

When $\lambda$ is large, the weight matrix tends to a zero matrix. Thus by choosing a suitable $\lambda$, each parameter in the weight matrix could be kept small and do not individually influence the result drastically leading to reduced overfitting.

## Mini Batch

Mini Batch is implemented to enable a streaming of training data for the model. However, this is not the main reason for this implementation as we could easily train on the entire training data due to its size.
The mini batch is implemented in this project as a form of cross validation when we continuously update the weights with a

Gao Yuze 1002971
Koh Chuan Shen Marcus 1002764
Zhao Nailin 1003003
Zhou Zhi 1002641

portion of training set and checking with the entire training data for RMSE.

To split the dataset into smaller batches, the readily available function np.split(data,numberOfBatches) is used.
The weight update function is then updated to accommodate the partial changes as :

$$W \leftarrow W + \alpha \left( \frac{PG - NG}{\#usersinbatch} - \lambda W \right)$$

## Bias

The motive of this extension is to grant the training gradient some directions to head for based on the error.
The bias is implemented with close reference to the given cfrbm.m file. We translated the Matlab code into python as of following:

$$\begin{aligned}
Setup & \\
hidbias &= np.zeros(W.shape[1]) \\
visbias &= np.zeros([W.shape[0], K]) \\
hidbiasinc &= np.zeros(W.shape[1]) \\
visbiasinc &= np.zeros([W.shape[0], K]) \\
Training\ with\ bias & \\
posHiddenProb &= rbm.visibleToHiddenVec(v, weightsForUser, hidbias) \\
poshidact &= np.sum(posprods, axis = (0, 2)) \\
posvisact &= np.sum(v, axis = 0) \\
negData &= rbm.hiddenToVisible(sampledHidden, weightsForUser, visbias) \\
negHiddenProb &= rbm.visibleToHiddenVec(negData, weightsForUser, hidbias) \\
neghidact &= np.sum(negprods, axis = (0, 2)) \\
negvisact &= np.sum(negData, axis = 0) \\
Update & \\
visbiasinc &= momentum * visbiasinc + (alpha/trStats[\text{"}n_users\text{"}]) * (posvisact - negvisact) \\
hidbiasinc &= momentum * hidbiasinc + (alpha/trStats[\text{"}n_users\text{"}]) * (poshidact - neghidact) \\
visbias &= visbias + visbiasinc \\
hidbias &= hidbias + hidbiasinc
\end{aligned}$$

In summary, we calculated the bias as a single value for each hidden parameter and a common value for all visible parameter.
We did not managed to have individual bias for each visible parameter due to mini batch. However, by taking a mean of all visible bias, we believe it is a reasonable estimate.
We further update the bias with momentum to prevent oscillation.

## Hyper-parameter tuning

To achieve a more consistent results, we tuned the following hyperparameters:

| Hyper-parameter | Tuning options |
|---|---|
| Momentum | [0.5,0.7,0.9] |
| Regularisation | [0.1,0.01,0.001] |
| Learning Rate | [0.1,0.01,0.001] |
| Number of Batches | [5,10] |
| Number of Hidden Variables | [6,8,10] |

Table 1: Hyper-parameter tuning options

We run the above mentioned 162 combination over 10 epochs each and measured their respective validation RMSE, the top 3 smallest RMSE combinations are of following:

Gao Yuze 1002971
Koh Chuan Shen Marcus 1002764
Zhao Nailin 1003003
Zhou Zhi 1002641

| momentum | regularisation | learning rate | batches | hidden variables | validation RMSE |
|---|---|---|---|---|---|
| 0.9 | 0.001 | 0.01 | 10 | 6 | 1.146451 |
| 0.5 | 0.010 | 0.10 | 10 | 6 | 1.146859 |
| 0.9 | 0.001 | 0.01 | 10 | 10 | 1.147624 |

Table 2: Best output combinations

To achieve a better confidence of estimates mentioned above, we found the average RMSE for each combination as of following:

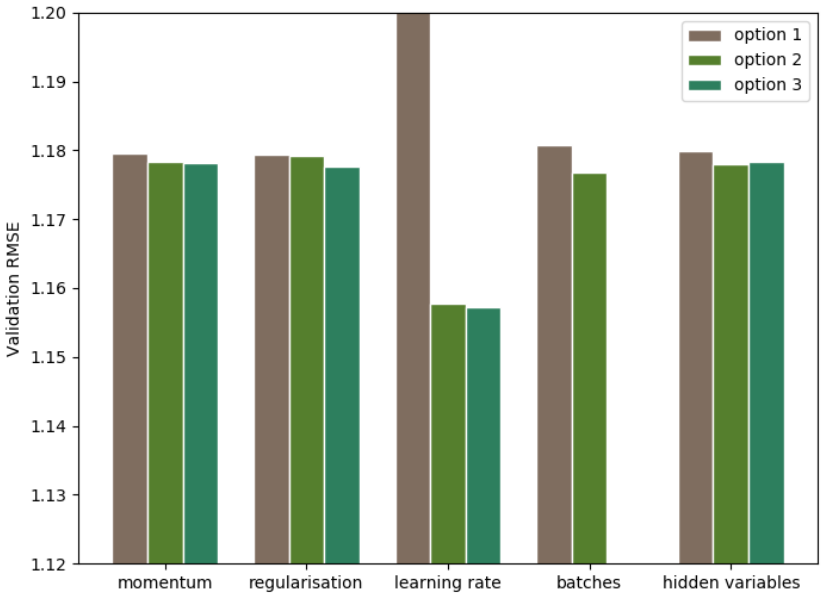


Figure 3: Mean RMSE

In Figure 3, the options are from Table 1's tuning options respectively.
From here, we observe a other than learning rate and number of batches, the difference in the mean is rather small.

Considering Table 2's result, we opted to choose the option of

| momentum | regularisation | learning rate | batches | hidden variables | Our validation RMSE |
|---|---|---|---|---|---|
| 0.9 | 0.001 | 0.01 | 10 | 10 | 1.147624 |

Table 3: Chosen combinations

This is the 3rd best option from our tuning set and we are using a slightly worse option of learning rate and hidden variable as shown in Fig 3. However the difference is marginal and could be due to stochastic error our tuning simulation.

## Others

We also use the built-in pickle package from python to save trained models for faster testing and recreation.
We used pandas library for result analysis and matplotlib library for visualisation.

Gao Yuze 1002971
Koh Chuan Shen Marcus 1002764
Zhao Nailin 1003003
Zhou Zhi 1002641

# 5 Results and Comments

We trained our find submission with 30 epochs.
As we use early stopping, the higher epoch number will not contribute to over fitting.

We submitted our results in 4 different stages as of following:

| Attempt | Description | Validation RMSE | Result |
|---|---|---|---|
| 1 | RBM + Momentum + Early Stopping + Adaptive Learning Rate + Regularisation | 1.2427 | 1.1889 |
| 2 | Attempt 1 + Mini Batch | 1.2342 | 1.1879 |
| 3 | Attempt 2 + hyper-parameter tuning | 1.158 | 1.0933 |
| 4 | Attempt 3 + Bias + hyper-parameter tuning | 1.157 | 1.0883 |

Table 4: Detailed Submission

In the first two submission, we opted for a worse output with extensions rather than basic RBM as we believe that the extensions make the model less variant. Since Attempt 3, with hyperparameter tuning, the extended model is always better performing than the basic model.

We believe the difference of our validation RMSE and resultant test RMSE is due to the small training and validation set of 300 each.

# 6 Conclusion

We greatly enjoyed this project as a prompted a deeper look into the netflix recommendation system.

# 7 Reference

1. Hinton, G. E. (2012). A practical guide to training restricted boltzmann machines. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7700 LECTURE NO, 599–619. https:doi.org10.1007978-3-642-35289-8-32

2. Louppe, G. (2010). Collaborative filtering: Scalable approaches using restricted Boltzmann machines. July. Retrieved from http:orbi.ulg.ac.behandle226874400

3. Qi, H., & Jin, H. (2009). Unsteady helical flows of a generalized Oldroyd-B fluid with fractional derivative. Nonlinear Analysis: Real World Applications, 10(5), 2700–2708. https:doi.org10.1016j.nonrwa.2008.07.008