Collaborative Filtering

LSINF2275
Data mining and decision making

By

William Glazer-Cavanagh
Vincent Tessier
Université Catholique Louvain

18 May 2020

1 Introduction

Being tasked with infering values for user-ratings given the *MovieLens-100k* dataset we implemented varied algorithms. We will first desrcibe the choices we made during this project which was quite open-ended.

We decided to implement only two models: user based K nearest neighbor and the latent matrix factorization model with a stochastic gradient descent solver. To make up for the low number of algorithms implemented we chose to focus on two aspects: testing thoroughly the hyper paramter space to optimize the error on our validation set and implementing the code on our own to better grasp the concepts of this exercise.

We will explain the theoretical background behind both algorithms as well as explain our training loop to explore the hyper parameter space. Then we will compare both models together. We assume the reader knows the problem setting given in the project guidelines and will not waste time re-explaining it.

2 User Based K Nearest Neighbor

2.1 Intuition

The intuition behind this algorithm is that users with the same tastes for movies should rate movies similarly. This does not make predictions easy however since different users have different tresholds for rating what is a good movie and what is a bad movie (*E.G. a user might think 3 is an average score and another might rate 4 as average*). There are ways to mitigate this problem which we will discuss.

The first step should be to compute how similar different users are to each other. We call this measurement the similarity metric and we will use it to weigth the predicted users ratings. Thus a user with a high similarity will affect a lot the predicted rating and a user which always gives the same score or is not at all similar will not affect the prediction much.

The reader should note that there is a difference between *not at all similar* and users with opposite tastes, which should also impact the score. We should use the oposite of this user's rating to make our prediction (*E.G. a user loves action movies and our target user that we are trying to predict typically hates action movies, thus we should use the fact that they have opposite tastes to make our prediction*)

2.2 Similarity Metrics

To follow our idea of thoroughly testing the hyper parameter space, we chose to implement several similarity metrics. While scouting the web, we found a paper (Agarwal & Chauhan, 2017) which listed several different algorithms and we chose four of those to implement in our model.

2.2.1 Pearson Similarity

This metric uses pearson's correlation coefficient between two population to determine wether they are correlated or not as a similarity metric. We use as r_u and r_v the items which have been rated by both users only.

$$\text{sim}(u,v) = \rho_{r_u,r_v} = \frac{\text{cov}(r_u, r_v)}{\sigma_{r_u}\sigma_{r_v}} \tag{1}$$

Since this coefficient has a tendency to give a good correlation to users with few examples in common, we used a correction factor to penalize users with few ratings in common: $\min\{|I_u \cap I_v|/50, 1\}$. This greatly improved results as we will show later.

2.2.2 Raw Cosine Similarity

This metric uses the dot product between the two user rating vectors and divides it by it's norm to show how similar they are.

$$\text{sim}(u,v) = \frac{\mathbf{r}_u \cdot \mathbf{r}_v}{||\mathbf{r}_u|| \cdot ||\mathbf{r}_v||} \tag{2}$$

We used this metric as is.

2.2.3 Jaccard Similarity

This metric is quite simple to compute, it consists of the cardinality of the intersection of items rated over the cardinality of the union of items rated. This metric does not account for rating similarity, hence we do not expect it to yield great results, we mostly used it for contrasting purposes.

$$\text{sim}(u,v) = \frac{|I_u \cap I_v|}{|I_u \cup I_v|} \tag{3}$$

We used this metric as is.

2.2.4 Proximity Impact Popularity Similarity

PIP (JunAhni, 2007) is a similarity metric which we implemented straight from the paper. This similarity measurement is quite computationnally heavy but yield promising results. It is a the sum of the product of three factors for all common ratings.

The first factor is proximity which penalizes rating which are distant (*E.G. 1 and 5*). The second factor is impact which advantages ratings which are far from the median rating in the scale (*2.5 in this case*) since they do provide a lot of information. The last factor is popularity which penalizes ratings that are close to the mean rating of the item for all users.

$$\text{sim}(u,v) = \sum_{k \in P(i)} \text{PIP}(r_{uk}, r_{vk}) \tag{4}$$

$$\text{PIP}(r_{uk}, r_{vk}) = \text{Proximity}(r_{uk}, r_{vk}) \cdot \text{Impact}(r_{uk}, r_{vk}) \cdot \text{Popularity}(r_{uk}, r_{vk}) \tag{5}$$

Since the definitions for the individual components are quite lengthy yet trivial, we invite the reader to consult the appendix for their individual definitions.

We also tried scaling this function by the number of common items rated as defined in the Pearson similarity.

2.3 Algorithm

The UBKNN algorithm itself is the weighted average of the most similar users' ratings. We can choose to normalize these ratings to predict the z score with $r_{ui} = \mu_u + \sigma_u \cdot z_{ui}$. This is the approach which was prefered because it performed much less erratically as will be described later. Thus the algorithm becomes:

$$\hat{r}_{ui} = \mu_u + \sigma_u \cdot \frac{\sum_{v \in I_u \cap I_v} \text{sim}(u, v) \cdot z_{vi}}{\sum_{v \in I_u \cap I_v} |\text{sim}(u, v)|} \tag{6}$$

2.4 Implementation

We used the numpy library for most features and followed the *Scikit-learn* philosophy where you initialize the model with the hyper-parameters you want and then call the `.predict()` function.

Thus we have a `SimilarityMetric` abstract class with child classes for each similarity metric. We instanciate this classes and give them to the `UserBasedKNe arestNeighbour` class in which we define the hyper-parameters. This makes for clean and elegant use of the code such as seen in this example:

```
56
57  metric = PearsonUserSimilarity(is_scaling=True)
58
59  algo = UserBasedKNearestNeighbour(
60      sim_metric=metric,
61      k=10,
62      std='normalization'
63      )
64
65  performance = crossfold_validation(
66      rating_matrix,
67      algo,
68      k_fold=10
69      )
70
```

Figure 1: Code required to define UBKNN

3 Latent Matrix Factorization

3.1 Intuition

The vectors of ratings are highly sparse and of very high dimensionnality. There is most definitely a lower dimension which could represent all the complexity of user tastes or movie types (*taste and movie type being only examples*).

Thus by projecting the users' and items' ratings in a smaller dimension, we catch the same complexity, allow low computationnal cost and avoid overfitting to accurately predict data the model has never seen before. The caveat is how to predict the number of latent features to use which becomes a hyper-parameter optimization problem.

3.2 Algorithm

We whish to decompose the matrix and there are many ways to do so. One frequently used way is the SVD decomposition which famously won the Netflix prize. However, in the scope of this project,

we went for a less computationally intensive and easier method to implement. This method came from the same Netflix competition (Funk, 2006) and is called Funk Matrix Factorization.

First we begin by showing the matrix decomposition technique which we assume the reader is familiar with. We want to decompose the rating matrix such that:

$$\mathbf{R} = \mathbf{P} \cdot \mathbf{Q}^T \tag{7}$$

These matrices which are respectively of dimension $\mathbf{P} \in \mathbb{R}^{u \times k}$ and $\mathbf{Q}^T \in \mathbb{R}^{k \times i}$ where i and u are respectively the number of items and users and k is the hyper-parameter of the latent dimension.

To find the matrices we use this optimization problem which minimizes the error between the matrix multiplication and the ground truth.

$$\min_{\mathbf{p}_u, \mathbf{q}_i} \sum_{r_{ui} \in \mathbf{R}} (r_{ui} - \mathbf{p}_u \cdot \mathbf{q}_i) \tag{8}$$

To solve this problem we use stochastic gradient descent which gives us this formula:

$$\begin{cases} \mathbf{p}_u \leftarrow \mathbf{p}_u + \alpha \cdot \mathbf{q}_i (r_{ui} - \mathbf{p}_u \cdot \mathbf{q}_i) \\ \mathbf{q}_i \ \leftarrow \mathbf{q}_i + \alpha \cdot \mathbf{p}_u (r_{ui} - \mathbf{p}_u \cdot \mathbf{q}_i) \end{cases} \tag{9}$$

These update rules are derived from this blog (Hug, 2017). To avoid overfitting, we chose to add a regularization paramater to the update rule which finally gives us:

$$\begin{cases} \mathbf{p}_u \leftarrow \mathbf{p}_u + \alpha \cdot \mathbf{q}_i (r_{ui} - \mathbf{p}_u \cdot \mathbf{q}_i) - \lambda \cdot \mathbf{p}_u \\ \mathbf{q}_i \ \leftarrow \mathbf{q}_i + \alpha \cdot \mathbf{p}_u (r_{ui} - \mathbf{p}_u \cdot \mathbf{q}_i) - \lambda \cdot \mathbf{q}_i \end{cases} \tag{10}$$

3.3 Implementation

We used a blog post (Valkov, 2019) to inspire our code since this was our first gradient descent implementation and we also implemented the algorithm with the same *Scikit-Learn* mindset to facilitate the training loop. This enables us to quickly scout the hyper-parameter space to find efficient solutions as this loop indicates:

```
113
114   features = [3, 5, 7, 10, 15]
115   lmbda = [0.5, 0.1, 0.05, 0.01]
116   learning_rate = [0.01, 0.001]
117
118 for f in features:
119     for l in lmbda:
120         for lr in learning_rate:
121             algo = LatentMatrixFactorization(
122                 ground_truth=rating_matrix,
123                 n_epochs=200,
124                 n_latent_features=f,
125                 lmbda=l,
126                 learning_rate=lr
127             )
128             prediction = crossfold_validation(rating_matrix, algo, 1)
129
```

Figure 2: Training loop to find the optimal hyper-parameters

To be able to figure out if our model was over-fitting, we also stored the values of the training and validation error on our model to be able to plot as such:

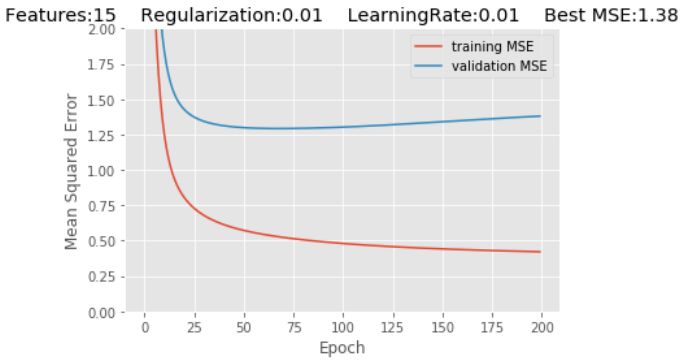

Figure 3: Example MSE tracking for overfitting detection in hyper-parameter search

4 Hyper Parameter Search

4.1 UBKNN

The first step was to find the optimal number of neighbors for each algorithm and see if it was always beneficial to have more neighbors. Based on our sparse dataset which contains on average $100k$ samples $\div$ 1682 items = 59 samples/items we found that after a while, additionnal neighbors are not benefical to the accuracy as figures 4 and 5 show.

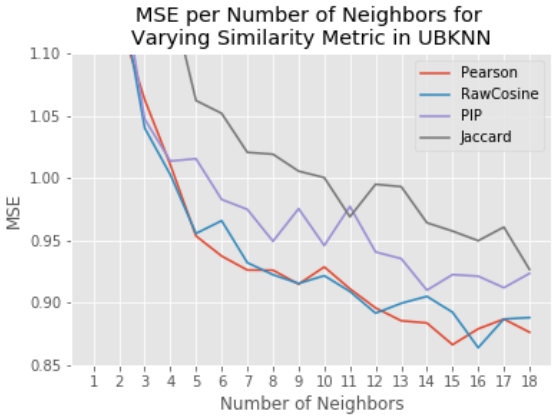

Figure 4: MSE per neighbor

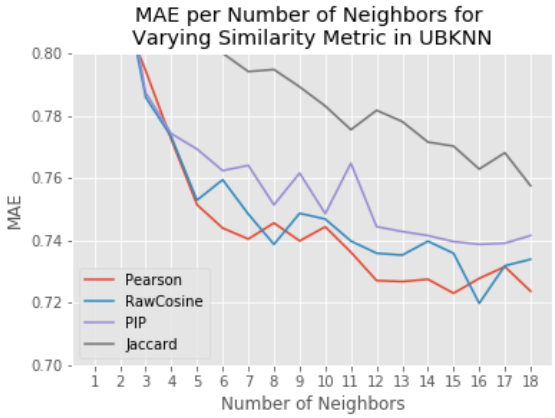

Figure 5: MAE per neighbor

Note that the noise on these graphs is present due to the fact that we used only one validation per data point with a 90%-10% train-validation split to save computation time, thus we should observe the trend and not the individual points. We find that no more improvement is had after the 15 neighbor mark for this dataset.

We also find that the *Jaccard* similarity metric performed poorly as expected. The *PIP* similarity performed underwhelmingly as it is very computationally expensive. Finally, the *Pearson* and *Cosine* measures were the ones which atained optimal performance.

4.2 Latent Matrix Factorization

The stochastic gradient descent is widespread in the machine learning world. It is notoriously difficult to find the right hyper-parameters. Thus, given that we did not have much computationnal ressources, we chose to make a random search with every combinations of the following parameters and choose the best one with a single 90%-10% train-validation split.

$$\text{Latent Matrix dimensions} : \{3, 5, 7, 10, 15\} \tag{11}$$

$$\text{Regularization} : \{0.5, 0.1, 0.05, 0.01\} \tag{12}$$

$$\text{Learning Rate} : \{0.01, 0.001\} \tag{13}$$

$$\text{Epochs} : \{200\} \tag{14}$$

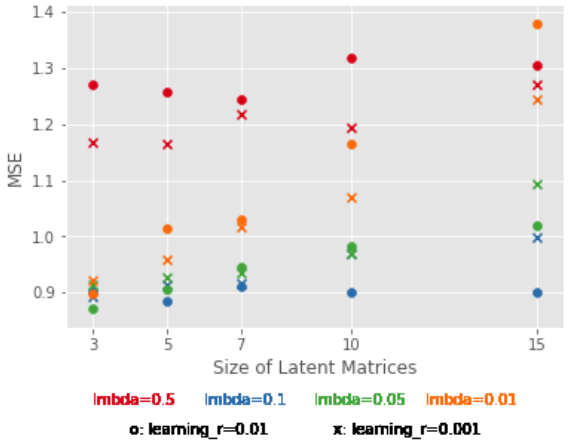

Figure 6: MSE Obtained on the Parameter Space where `lmbda` is Regularization

With this graph we find that the combination which appears to minimize the MSE is regularization: 0.05, learning rate: 0.01, latent matrix dimension: 3 which is at the bottom left of this graph. By looking at the validation accuracy, we find that the validation accuracy stops increasing at around epoch 100 (*This graph is in the appendix*).

4.3 Overall Comparison

Metric	Pearson	Cosine	Matrix Fact.	PIP	Jaccard	Baseline
MSE	.887	.892	.893	.918	.956	1.045
MAE	.728	.732	.740	.740	.770	.816

Table 1: MAE & MSE per Similarity Measure with 10-Fold Cross Validation with Item Mean Baseline

Hence we see that the accuracy is better for the UBKNN with the Pearson model, closely followed by Cosine and Matrix Factorization. However another important factor to look at is the time to predict these values.

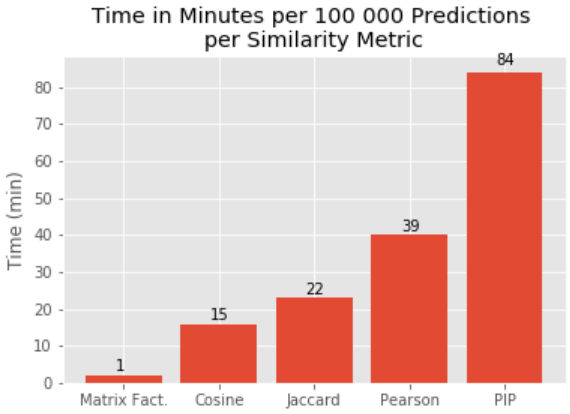

Figure 7: Time (s) required to predict 100 000 samples

We should note that this graph is not fair for metrics other than the matrix factorization technique since the training time for this algorithm is constant and does not vary on the amount of predictions required. Thus if we were to predict 10 000 samples, the Matrix factorization would be about the same time as the cosine metric. However, to predict the whole matrix, the best choice is obviously the Matrix Factorization technique for a good accuracy - computation tradeoff.

Hence we will choose the Matrix factorization technique as the best model for this dataset since it's performance is not far off from UBKNN however it is orders of magnitude more performant.

4.4 Caveats

As we were restricted by our computation power, we could not explore the whole paramter space for the Matrix factorization technique. This means that our results could surely be further optimized.

4.5 How to Use our Model

Inside the `main.py` we made a `predict(R)` function which takes as input a rating matrix `R` and outputs the predictions `R_hat` however this is not the only model we can define as some examples have been provided in the main such as a UBKNN definition.

5 Conclusion

We have implemented many different algorithms to make a reccomender system. With more time and computation power, we would've liked to implement even more algorithms, especially matrix factorization techniques which seem very promising.

We respected our goal to focus on code clarity and implement by ourselves every algorithm and test them thouroughly. We would like to thank the organizers of this exercise since it was very instructive.

References

Agarwal, A., & Chauhan, M. (2017). Similarity measuresused in recommender systems: A study. *International Journal of EngineeringT echnology Science and Research*, *4*(6).

Funk, S. (2006). Netflix update: Try this at home. Retrieved from `https://sifter.org/simon/journal/20061211.html`

Hug, N. (2017). Understanding matrix factorization for recommendation (part 3) - svd for recommendation. Retrieved from `http://nicolas-hug.com/blog/matrix_facto_3`

JunAhni, H. (2007). A new similarity measure for collaborative filtering to alleviate the new user cold-starting problem. *Information Sciences*, *178*(1).

Valkov, V. (2019). Music artist recommender system using stochastic gradient descent — machine learning from scratch (part vii). Retrieved from `https://towardsdatascience.com/music-artist-recommender-system-using-stochastic-gradient-descent-machine-learning-from-scratch-5f2f1aae972c`

A PIP Similarity

Exceprt from the (JunAhni, 2007) Paper

Agreement

For any two ratings r_1 and r_2, let R_{max} be the maximum rating and R_{min} the minimum in the rating scale, and let $R_{med} = \frac{R_{max}+R_{min}}{2}$

A Boolean function *Agreement*(r_1, r_2) is defined as follows:

Agreement(r_1, r_2) = **false** if ($r_1 > R_{med}$ and $r_2 < R_{med}$) or ($r_1 < R_{med}$ and $r_2 > R_{med}$), and

Agreement(r_1, r_2) = **true** otherwise

Proximity

A simple absolute distance between the two ratings is defined as:

$D(r_1, r_2) = |r_1 - r_2|$ if *Agreement*(r_1, r_2) is **true**, and

$D(r_1, r_2) = 2 \cdot |r_1 - r_2|$ if *Agreement*(r_1, r_2) is **false**

Then the *Proximity*(r_1, r_2) is defined as:

$$Proximity(r_1, r_2) = \{\{2 \cdot (R_{max} - R_{min}) + 1\} - D(r_1, r_2)\}^2$$

Impact

Impact *Impact*(r_1, r_2) is defined as:

Impact(r_1, r_2) = ($|r_1 - R_{med}| + 1$)($|r_2 - R_{med}| + 1$) if *Agreement*(r_1, r_2) is **true**, and

$$Impact(r_1, r_2) = \frac{1}{(|r_1 - R_{med}|+1)(|r_2 - R_{med}|+1)}$$ if *Agreement*(r_1, r_2) is **false**

Popularity

Let μ_k be the average rating of item k by all users

Then *Popularity*(r_1, r_2) is defined as:

$$Popularity(r_1, r_2) = 1 + \left(\frac{r_1+r_2}{2} - \mu_k\right)^2$$ if ($r_1 > \mu_k$ and $r_2 > \mu_k$) or ($r_1 < \mu_k$ and $r_2 < \mu_k$), and

Popularity(r_1, r_2) = 1 otherwise

Figure 8: PIP definition