University of Liège

Machine Learning

Movie rating predictor

Master 1 in Data Science & Engineering

Authors:
Tom Crasset
Antoine Louis
Cyril Geortay

Professors :
L. Wehenkel
P. Geurts

Academic year 2018-2019

1 Introduction

The aim of this project is to predict the rating given by a user for a movie. This is similar to the famous problem where Netflix challenged the machine learning community to find the model that could best suggest movies for a user, given his previously acquired tastes in movies. This challenge is hosted as a Kaggle challenge where every team of 2 or 3 students compete with each other to try to have the model with the lowest mean squared error. To achieve this, a data set of 69,453 ratings given by 911 users to 1,541 movies ($\approx 5\%$ of the ratings) is provided, with some extra information about the movies and the users.

In the following sections, we present the different steps of our strategy, with the data we decided to consider, the models we decided to try out, the choice of their parameters and our results.

2 Problem approach

2.1 Basic idea

Our first idea was to start from the *toy_example* model where the approach ignores the user features (age, gender, etc.) as well as the movie's (release date, category, etc.). We tried to adjust the parameters of the model, which was using a *DecisionTree* estimator with no constraint on the depth, on a ($n\_movies \times n\_users$, $n\_movies+n\_users$) learning matrix. We chose to adapt the tree's `max_depth` value. However, we quickly found out that a bigger depth doesn't mean a bigger score. In fact, it was quite the opposite. Using a `max_depth` of 1 gave us the best results. This model was quite simple and we didn't expect much of it. We also tried a *KneighborsRegressor* approach, but the results were pretty bad, even when varying the number of neighbors.

2.2 First approach

For the following models, we decided to take the user features and the movie features into account, by adding them to the *data_test.csv*. We implemented a simple python script using the pandas library to join the different files *data_user.csv*, *data_movie.csv* and *data_train.csv* together, dropping a few unnecessary columns like `IMBD-Title` or `video_release_date`, resulting in a dataset with around 50 features due to the one-hot encoding of the gender and occupation of the user.

We first trained this composed dataset with the *DecisionTree* and *KneighborsRegressor* estimators by doing cross-validation over 10 samples and varying their complexity. But the mean squared error still wasn't in the range we wanted it to be, as can be seen in Figure 1.

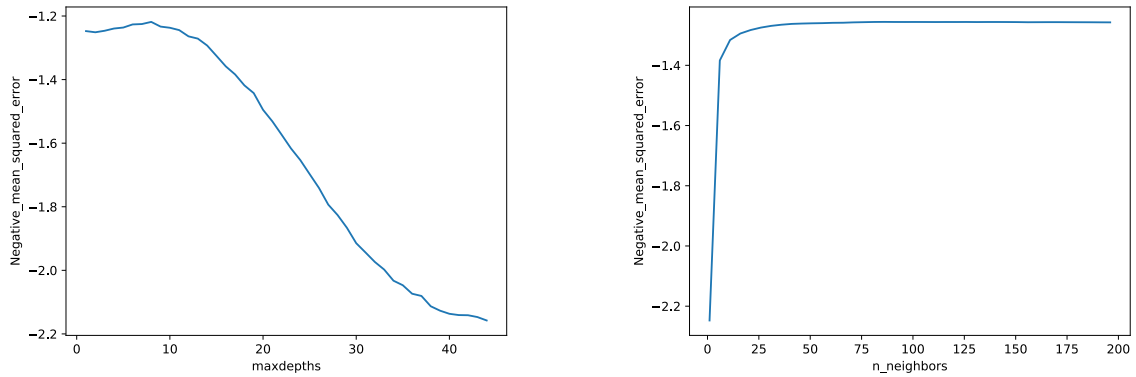

Figure 1: (left) NMSE using *DecisionTreeRegressor* with a varying complexity
(right) NMSE using *KNeighborsRegerssor* with a varying complexity

2.3 Ensemble methods

As the results weren't conclusive at all, we thought about using ensemble methods like bagging or boosting. Coming from the *DecisionTrees* estimator, the next logical step was a *RandomForest* model. We chose to use bootstrap samples and to keep all the features in our data. Here again, we had the choice to adjust the `max_depth` of the trees. We ran multiple simulations with different depths and computed the mean squared error (MSE) for each one of them, as can be seen in Figure 2.

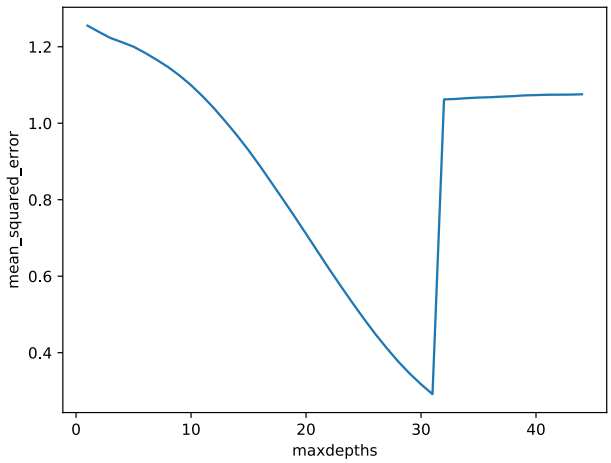

Figure 2: Mean squared error using a *RandomForest* model with a varying complexity max_depth

In addition to tuning the depth, it became apparent that the number of estimators for the bagging approach could be tuned too. Using a simple iteration of various numbers of estimators and computing the MSE for each iteration, we simultaneously predicted on the test and training sets taking as inputs the data containing relevant user and movie features. The results are presented in Figure 3. As the number of estimators increases, it seems that we face overfitting where the MSE on the training set nearly immediately drops way lower than the error on the testing set. To oppose to that, we tried to reduce the number of features from the dataset, but this only hurt the general accuracy while still overfitting at the same threshold.

From this graph, it can be deduced that there is no difference in accuracy when building a *RandomForest* with 20 or 100 estimators, so it is better to use the lower bound to slightly reduce the chance to overfit the training data while reducing the computation time.

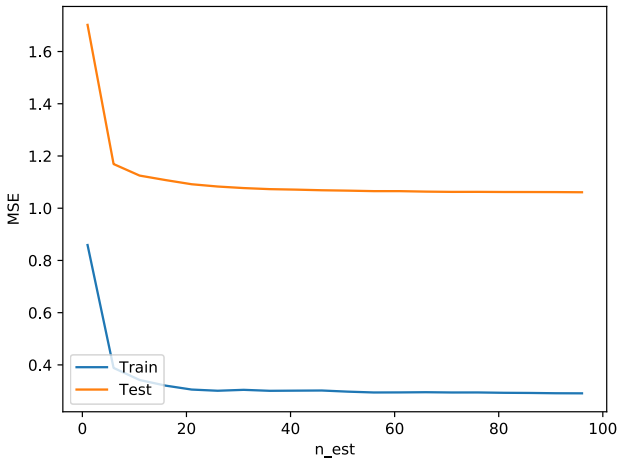

Figure 3: Mean squared error on a *RandomForest* model for multiple numbers of estimators

After hyper-parameters tuning, we noticed that choosing the best parameters only resulted in a very slight improvement of 0.8% in contrast to the *RandomForest* model we had after tuning only `max_depth`, which seems to be the most important parameter compared to the others.

Bagging was a small advance but still a disappointment overall because we expected a much higher drop in the mean squared error. Besides bagging, there is also the boosting approach. Using boosting alone or in combination with a bagging approach such as *RandomForest*, like *GradienBoostingRegressor* or *AdaboostRegressor* respectively, didn't yield better results than the previous models, even after careful tuning of the parameters.

2.4 Neural network

As it has proved its efficiency in complex machine learning problems such as image recognition, we thought that testing a neural network on our data sets would be a promising move. Thus, we used a multi-layer perceptron model (*MLPRegressor*) to try and approximate a function predicting a rating given the features of a user and a movie.

We played with the structure of the network, assuring that each model that we were testing had converged (negligible variation of the loss function from one iteration to the next). We first chose to use a structure of one hidden layer, playing only with the number of neurons in this layer. Then, secondly, we made a structure composed of layers of 50 neurons each, changing only the number of layers. The results of our tests can be observed in Figure 4.

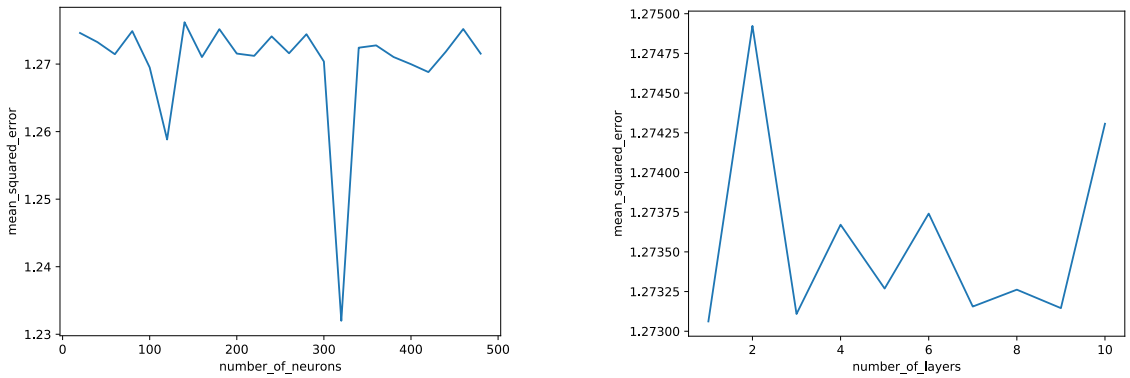

Figure 4: (left) MSE using *MLPRegressor* with a varying number of neurons in its unique layer (right) MSE using *MLPRegressor* with a varying number of layers

It turned out that the structure of the neural network wasn't quite involved in the accuracy of the model and that the mean squared error (MSE) produced wasn't even close to the one produced by *RandomForest*.

2.5 Matrix factorization

It became clear that there was no much bigger improvement to be made with the standard models, so we began reading the literature around the topic of movie ratings and recommendations. Various sources ([1], [2]) stated matrix factorization as one efficient approach for rating prediction, and so we decided to use it as our base model.

Matrix factorization is breaking the huge sparse rating matrix up into a product of multiple dense matrices. The intuition behind matrix factorization and movie rating is that there should be hidden features from which a given user rates a movie. For example, a user might like movies that have the same director or famous actor in it. Matrix factorization can pick up on that trend even though that information is not given as input data. By discovering these hidden features, the model could be able to predict the rating that a certain user gives to a certain movie, based on whether the features that the user likes would be found in that given movie or not. If we denote R the rating matrix, U the set of

users, M the set of movies, and K the number of hidden features, we have to find a matrix P (of size $|U| \times |K|$) and a matrix Q (of size $|M| \times |K|$) such that

$$R = P \times Q^T, \tag{1}$$

In this way, P represents the association between a user and the features and Q represents the association between a movie and the features.

Then, to get the predicted rating that a given user makes for a movie, we just need to compute the dot product of P and Q.

$$\hat{r}_{ij} = p_i^T q_j = \sum_{k=1}^{k} p_{ik} q_{kj} \tag{2}$$

However, we need to compute the matrices P and Q. This is done by minimising the squared error between the predicted rating $\hat{r}$ and the real rating r using gradient descent. The error is given by:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2 \tag{3}$$

and the modification of P and Q from one iteration to the next results in:

$$\frac{\partial}{\partial p_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(q_{kj}) = -2e_{ij} q_{kj} \tag{4}$$

$$\frac{\partial}{\partial q_{ik}} e_{ij}^2 = -2(r_{ij} - \hat{r}_{ij})(p_{ik}) = -2e_{ij} p_{ik} \tag{5}$$

which yields the following new values of P and Q:

$$p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + 2\alpha e_{ij} q_{kj} \tag{6}$$

$$q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + 2\alpha e_{ij} p_{ik} \tag{7}$$

where α is the learning rate with a value in the vicinity of 10^{-2}.

However, to avoid overfitting over the training data, we need to introduce a regularization parameter β, with a similar magnitude as α:

$$e_{ij}^2 = (r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj})^2 + \frac{\beta}{2} \sum_{k=1}^{K} (||P||^2 + ||Q||^2) \tag{8}$$

The update of the predictions becomes:

$$p'_{ik} = p_{ik} + \alpha \frac{\partial}{\partial p_{ik}} e_{ij}^2 = p_{ik} + \alpha(2e_{ij} q_{kj} - \beta p_{ik}) \tag{9}$$

$$q'_{kj} = q_{kj} + \alpha \frac{\partial}{\partial q_{kj}} e_{ij}^2 = q_{kj} + \alpha(2e_{ij} p_{ik} - \beta q_{kj}) \tag{10}$$

To aid the factorization method to converge faster, one can introduce the notion of bias. Intuitively, this is easy to understand if we assume that every user has certain preferences that might bias him to a certain movie, for example an avid movie enthusiast might, in general, rate movies more critically than an occasional user that enjoys watching a movie once in a while. This is denoted formally as:

$$\hat{r}_{ij} = b + \sum_{k=1}^{k} p_{ik} q_{kj} \tag{11}$$

where b is a sum of the general bias (a mean over the non-zero elements of the rating matrix), the user biases and the movie biases.

In practice, we computed our matrix factorization model thanks to an implementation found on the web ([3]).

2.6 Improvement of matrix factorization

Using the matrix factorization method alone wasn't sufficient to lower the MSE on the platform. So, we decided to apply another machine learning algorithm on top of that. As inputs, we created a learning matrix from the rating matrix predicted by the matrix factorization model, and trained it on various estimators, by running a lot of simulations to tune the parameters.

After having tested distinct models, it results that *GradientBoostingRegressor* estimator with a `max_depth` of 7 gives us the lowest error, with a MSE of 0.87931 as private score on Kaggle.

2.7 Tuning and validation techniques

For each and every model we implemented, the general procedure for creating it was pretty much the same. First we select a wide range of parameters sequences related to the model we were working on. We then use the *RandomizedSearchCV* class, giving it these sequences, the training data, and the model to train, to find the sequence of parameters that is best suited to the model in the scope of our training data. This method is called hyper-parameters tuning.

Then, we choose one or two parameters that are determinant in the accuracy of our model. Then we make a loop were the value of this (or these) parameter(s) is changed at each iteration. At each step, we create the model and we use cross validation on the whole training data set. From cross validation, we extract the mean MSE of the splits to use it as the MSE of the model. Finally, we select the model whose determinant parameter(s) value(s) deliver(s) the best MSE to test it on the testing data set.

2.8 Results

In Table 1, we listed the MSE's for each model we tested in this project.

	Validation score	Public score	Private score
MF with Gradient Boosting	0.90192	0.88591	0.87931
MF with Linear Regression	0.90981	0.90236	0.89092
MF with Multi-layer Perceptron	0.91734	0.90319	0.89500
MF with Random Forest	0.91346	0.90676	0.89656
MF with Decision Tree	1.05769	1.00460	0.99397
Random Forest	1.04311	1.02807	1.03229
Adaboost with Random Forrest	1.12528	1.12078	1.11978
Decision Tree	1.19520	1.18696	1.17497
Multi-layer Perceptron	1.23194	1.22668	1.21609
Matrix Factorization	1.41218	1.40569	1.37725
K-nearest Neighbors	2.53147	2.52036	2.54654

Table 1: MSE's (scores) for different models

3 Conclusion

In this project, our thinking process went in five distinct steps. First we thought about a few simple models to use on our data sets (Decision Trees and K-nearest Neighbors). We knew and rapidly figured out that these methods were far to be optimal, but at least it gave us a landmark to base on.

Secondly, we tried ensemble methods by using Random Forest. This gave us more satisfying results but still not quite what we expected.

Thirdly, we considered the idea to use a neural network (Multi-layer Perceptron), as it has proven to be very efficient in certain situations. This was a disappointment since, despite the time spent trying to tune the parameters and to find a good structure, the results were even worse than what could be delivered by a simple model such as Decision Trees.

Then, seeing that we would not be able to get a sufficiently good score with the use of classical methods, we looked in literature for other ways to do. We found the Matrix Factorization method, but its results were even worse than the ones delivered by the neural network.

Finally, we used classical machine learning methods on top of Matrix Factorization. This brought a tremendous improvement to our results, leading us to a satisfying accuracy in predictions.

References

[1] Matrix Factorization and Neighbor Based Algorithms for the Netflix Prize Problem, Gábor Takács et al.

[2] `http://www.albertauyeung.com/post/python-matrix-factorization/`

[3] `https://github.com/albertauyeung/matrix-factorization-in-python`