How to Build a Model-Based Recommendation System using Python Surprise

A step-by-step guide on implementing a latent factor recommendation engine using the Surprise library in Python.

Mate Pocs Follow
May 18 · 9 min read ★

Photo by Deva Williamson on Unsplash

This post is the last piece of my Python Surprise recommendation series in which I present the techniques I used in my boardgame recommendation engine project. (See my GitHub repo for the whole project.)

Previous posts in the series:

Part 1: How to Build a Memory-Based Recommendation System using Python Surprise: I recommend reading this post first if you are not familiar with the topic, especially the Data Import and Data Preparation steps since they are identical for the memory-based and model-based approach.

Part 2: My Python Code for Flexible Recommendations: This post contains the additional code I wrote for the recommendation framework that enables one to create predictions without re-training the whole model. However, the approach only works for the KNN-style, simpler models.

I also foolishly started an analogy on cupcakes, backed by the cover pictures. In the first post, we had many cupcakes, didn’t know how to choose, in the second, we had an odd newcomer cupcake to the party, struggled with integrating it, and now… well I guess now the cupcakes are in a row, representing matrix factorisation. That’s the best I can come up with.

In this post, we are going to discuss how latent factor models work, how to train such a model in Surprise with hyperparameter tuning, and what other conclusions we can draw from the results.

Model-Based Recommendation Systems

A quick recap on where we are. Within recommendation systems, there is a group of models called collaborative-filtering, which tries to find similarities between users or between items based on recorded user-item preferences or ratings. In my previous posts, we discussed a subgroup of collaborative systems called memory-based models. They are called memory-based because the algorithm is not complicated, but requires a lot of memory to keep track of the results.

In this post, we are discussing another subgroup of collaborative-filtering models: model-based models (which is a rather silly name). As opposed to the memory-based approaches, this uses some sort of machine learning algorithm. There are many different variations within this group, what we are going to concentrate on is the singular value decomposition methods.

In Surprise, there are three such models: `SVD` , `SVDpp` , and `NMF` , out of which I am only going to discuss `SVD` . `NMF` is a simplified version, ignoring user and item biases. `SVDpp` adds a very cool feature where you also separately keep track of whether the user rated the item or not, which should also be relevant information of course, but I found that it does not improve my efficiency while adding a lot of computation time.

Math Formula

In the SVD model, an estimated rating of user *u* on item *i* is calculated as:

$$\hat{r}_{ui} = \mu + b_u + b_i + q_i^T p_u$$

where μ is the overall average rating, and every other parameter is calculated from the model with a gradient descent method. So the model will try to fit this estimated rating on all the known ratings, minimise the MSE, and return the closest fit.

b_u and b_i are scalars, they represent the biases of the user *u* or item *i*. For example, user *u* tends to be b_u off from from the grand average rating. These biases can be switched off when fitting the model, that is basically what the `NMF` model is.

p_u and q_i are vectors, and their length is a hyperparameter of the model, *n*. They are the actual matrix-factorisation part of the model, that is where the magic happens. Each user and item will be represented by their vector, that tries to capture their essence in *n* numbers. And we get the rating by multiplying the item — user pairs (and adding averages and biases of course).

It might be tempting to think of these n dimensions as something humanly comprehensible. For example, if we deal with boardgames, the first dimension could measure how complex the rulebook is. Now a user that places a high emphasis on complexity (which means they have a large number at q_i [*1*]) will give a high rating to a game that has a high complexity (which means a high p_u[*1*]). However, in my experience, this is rarely the case, it is quite difficult to match meaning with the individual coordinates that come out of the model.

Training the Model in Surprise

Assuming you already have the database imported and set up (once again, if you are not sure how to do that, please refer to my previous post), working with `SVD` is similar to how you work with other models in Surprise. First, you need to import the model:

```
from surprise import SVD
```

Then you can fit the model on the trainset and test the model performance using the `RMSE` score (which stands for Root Mean Squared Error, the lower the better):

```
SVD_model = SVD()
SVD_model.fit(trainset)
predictions = SVD_model.test(testset)
accuracy.rmse(predictions)
```

Similar to the memory-based models, in order to predict a rating for a specific user, you can use the predict method, but the user needs to be in your database:

```
SVD_model.predict(uid = 'TestUser1', iid = '161936')
```

Analysing the Model

As you saw, fitting an SVD model is simple, but analysing the result is a bit more complicated.

Using the `pu` , `qi` , `bu` and `bi` methods of an `SVD` object, you can get the corresponding values from the math formula. In my project, I found `qi` to be the most interesting: calling the `qi` method on the already fit `SVD` model will return a 2-dimensional array, where height is the number of items, width is the `n_factor` parameter of the model (we will discuss the parameters in the next section). Each row represents an item with `n_factor` number of factors, these are the so-called latent factors that the model found, and they represent the item in the rating calculations.

As I mentioned before, it is tempting to try to find easily comprehensible meaning in these coordinates, but in my experience, it doesn't really happen. However, I think one really interesting way to take this analysis one step further would be to use these latent factors as a basis for a cluster analysis, and see if anything interesting comes up from that. You can use the `n_factor` latent factors as features, and calculate the item's distances based on them, like you would do in any regular cluster analysis.

What are our Hyperparameters?

An important part of any machine learning process is tuning the hyperparameters. In this section, we are going to have a look at the `SVD` parameters in Surprise.

Before we start, please note that the RMSE score with the default parameters was 1.332 for my project, and after my GCP Virtual Machine ran for hours, I managed to bring RMSE down to 1.3210. Which is not a lot of improvement… Might just be my database, might be because the default parameters in Surprise were set up efficiently. Regardless of that, I think it's still important to consider the hyperparameters.

There are four hyperparameters I tuned:

- `n_factors` : We have briefly touched upon this in the previous section, this parameter determines the size of your p_u and q_i vectors. This determines how many latent factors the model will try to find. The higher the number is, the more power the model has, but it also comes with a higher chance of overfitting.

- `n_epochs` : This factor determines how many times the gradient descent calculations are repeated. Increasing this number makes predictions more accurate, but requires longer to calculate.

- `lr_all` : Learning rate factor for all of the parameters. These are the step sizes the model will use to minimise the cost function, see more on this in the Surprise documentation.

- `reg_all` : Regularisation factor for all of the parameters. Surprise uses an L2 regularisation, which roughly means that it will try to minimise the differences between the squared value of the parameters. (The parameters being all the b_u, b_i, p_u and q_i.)

There are numerous other parameters you can play with, most of them are different setups of learning rate or and regularisation parameters. You can technically set a different learning rate or regularisation for each four types of the model parameters, and the `…_all` parameters cover them all. I did not find it necessary to go into such details.

Hyperparameter Tuning

We are going to use `GridSearchCV` to tune the hyperparameters in Surprise. It works mostly like its counterpart in scikit-learn, as the name suggests, it will search all the possible combinations on the hyperparameter grid, using Cross-Validation.

First, we need a dictionary in which the keys are hyperparameter names, and the values are lists of different items you want to check:

```
param_grid = {
    'n_factors':[5, 10,20],
    'n_epochs': [5, 10, 20],
    'lr_all': [0.002, 0.005],
    'reg_all': [0.4, 0.6]}
```

You have to be careful about how you set up these parameters, as each of the possible combinations will be checked. In our cases, that is 3 * 3 * 2 * 2 = 36 different combinations, and for every one of these, the model will run multiple times, depending on the Cross-Validation you choose. I left the `cv` parameter at 5-fold, which means 36 * 5 = 180 model runs in total.

That can take a long time, and this might be the time to start thinking about using cloud computing. I wrote a post recently about how to quickly set up a free Virtual Machine on the Google Cloud Platform.

Once you have your parameter grid, you can set up the `GridSearchCV` object like so:

```
gs_model = GridSearchCV(
    algo_class = SVD,
    param_grid = param_grid,
    n_jobs = -1,
    joblib_verbose = 5)
```

The first parameter, `algo_class`, is the type of model you want to use. `n_jobs` = -1 simply tells the model that it can use all available processors, which is highly desirable when you have a parallelisable operation. One other parameter that you might want to change is `cv`, I just left it at the default, which does a 5-Fold Cross-Validation.

Then you can simply fit on the data:

```
gs_model.fit(data)
```

Please note that once again, surprise handles databases a bit differently, you can only fit your `GridSearchCV` on the whole dataset, can't split it into train and test.

Finally, you can get a list of the parameters from the parameter grid that resulted in the best `RMSE` score:

```
gs_model.best_params
```

My Process

My process when working with `GridSearchCV` was the following:

- Fit a `GridSearchCV` model on the whole data, with a parameter grid that covered a wide range
- Calculated the Cross-Validated `RMSE` score
- Repeated the process for a different parameter grid that either drilled down to a lower level if the optimal parameter seemed to be in the middle, or explored higher / lower parameters if the optimal parameter seemed to be at a boundary of the previous parameter grid

- Once the decrease in `RMSE` score was minimal, saved the best hyperparameters and used them in the `SVD` models in the next steps
- Because I wanted to compare the results with my KNN-type models from earlier, I ran the `SVD` model on the `trainset`, and calculated the test `RMSE` score on the `testset`

An alternative approach would have been to simply rely on the GSCV scores, and fit the final model only once on the full trainset.

Conclusion

This concludes my series on recommendation system project.

It was interesting to work with all these different approaches, especially how close their performance turned out to be. I spent a lot of time identifying the best model, but the truth is, even the simplest KNN models performed relatively well.

I think that is partly because my data was not nearly as sparse as it usually is with recommendation systems. 10% of the possible user - item ratings were populated, which is considered to be an extremely high ratio. Imagine if people on average bought 10% of all the products on Amazon! The high ratio is because I limited my focus on the top 100 boardgames of all time, which are naturally popular titles. As an additional bonus, average ratings were pretty close and high, since these are all universally considered to be good games.

References

Surprise documentation:

Welcome to Surprise' documentation! - Surprise 1 documentation

If you're new to Surprise, we invite you to take a look at the Getting Started guide, where you'll find a series of...

surprise.readthedocs.io

My previous posts in the series:

How to Build a Memory-Based Recommendation System using Python Surprise

A step-by-step guide on implementing a kNN-style recommendation engine using the Surprise library in Python, from data…

towardsdatascience.com

My Python Code for Flexible Recommendations

My additional custom Python code that enables you to run more flexible recommendations based on Surprise library's…

towardsdatascience.com

Python Data Science Machine Learning Recommendation System Scikit Surprise