index

May 20, 2022

1 Implementing Recommendation Engines with `surprise`

1.1 Introduction

This lesson will give you a brief introduction to implementing recommendation engines with a Python library called `surprise`. You'll get a chance to try out multiple different types of collaborative filtering engines, ranging from both basic neighborhood-based methods to matrix factorization methods.

1.2 Objectives

You will be able to:

- Use `surprise`'s built-in reader class to process data to work with recommender algorithms
- Use `surprise` to create and cross-validate different recommender algorithms
- Obtain a prediction for a specific user for a particular item

In this lesson, we'll be working with a dataset built-in to `surprise` called the `jester` dataset. This dataset contains jokes rated by users on a scale of -10 to 10 based off a user's perceived humor level for a given joke. Let's get recommending some jokes!

First, you'll have to load the jokes dataset. You might have to download it first if prompted. Let's investigate the dataset after we load that. In this folder, you'll find the file for the text of jokes if you want to investigate what caliber of human you're dealing with here.

[1]:
```
from surprise import Dataset
from surprise.model_selection import train_test_split
jokes = Dataset.load_builtin(name='jester')
```

```
Dataset jester could not be found. Do you want to download it? [Y/n] Y
Trying to download dataset from
http://eigentaste.berkeley.edu/dataset/archive/jester_dataset_2.zip…
Done! Dataset jester has been saved to /Users/miladshirani/.surprise_data/jester
```

[2]:
```
type(jokes)
```

[2]:
```
surprise.dataset.DatasetAutoFolds
```

[3]:
```
# Split into train and test set
trainset, testset = train_test_split(jokes, test_size=0.2)
```

Notice how there is no `X_train` or `y_train` in our values here. Our only features here are the ratings of other users and items, so we need to keep everything together. What is happening in the train-test split here is that `surprise` is randomly selecting certain r_{ij} for users u_i and items $i_{j} $. 80% of the ratings are in the training set and 20% in the test set. Let's investigate `trainset` and `testset` further.

[4]:
```
print('Type trainset :',type(trainset),'\n')
print('Type testset :',type(testset))
```

```
Type trainset : <class 'surprise.trainset.Trainset'>


Type testset : <class 'list'>
```

Interestingly enough, the values here are of different data types! The `trainset` is still a `surprise` specific data type that is optimized for computational efficiency and the `testset` is a standard Python list - you'll see why when we start making predictions. Let's take a look at how large our `testset` is as well as what's contained in an individual element. A sacrifice of `surprise`'s implementation is that we lose a lot of the exploratory methods that are present with Pandas.

[5]:
```
print(len(testset))
print(testset[0])
```

```
352288
('60701', '8', -5.375)
```

1.3 Memory-Based Methods (Neighborhood-Based)

To begin with, we can calculate the more simple neighborhood-based approaches. Some things to keep in mind are what type of similarities you should use. These can all have fairly substantial effects on the overall performance of the model. You'll notice that the API of surprise is very similar to scikit-learn when it comes to model fitting and testing. To begin with, we'll import the modules we'll be using for the neighborhood-based methods.

[6]:
```
from surprise.prediction_algorithms import knns
from surprise.similarities import cosine, msd, pearson
from surprise import accuracy
```

One of our first decisions is item-item similarity versus user-user similarity. For the sake of computation time, it's best to calculate the similarity between whichever number is fewer, users or items. Let's see what the case is for our training set.

[7]:
```
print('Number of users: ', trainset.n_users, '\n')
print('Number of items: ', trainset.n_items, '\n')
```

```
Number of users:  58798


Number of items:  140
```

There are clearly way more users than items! We'll take that into account when inputting the

specifications to our similarity metrics. Because we have fewer items than users, it will be more efficient to calculate item-item similarity rather than user-user similarity.

[8]:
```
sim_cos = {'name':'cosine', 'user_based':False}
```

Now it's time to train our model. Note that if you decide to train this model with `user_based=True`, it will take quite some time!

[9]:
```
basic = knns.KNNBasic(sim_options=sim_cos)
basic.fit(trainset)
```

```
Computing the cosine similarity matrix…
Done computing similarity matrix.
```

[9]:
```
<surprise.prediction_algorithms.knns.KNNBasic at 0x7fbaeff97d30>
```

And now our model is fit! Let's take a look at the similarity metrics of each of the items to one another by using the `sim` attribute of our fitted model.

[10]:
```
basic.sim
```

[10]:
```
array([[1.        , 0.46531749, 0.16642227, …, 0.47965091, 0.38701276,
        0.32375144],
       [0.46531749, 1.        , 0.20890969, …, 0.52057486, 0.63129855,
        0.56011353],
       [0.16642227, 0.20890969, 1.        , …, 0.11791301, 0.13302966,
        0.25306592],
       …,
       [0.47965091, 0.52057486, 0.11791301, …, 1.        , 0.62050709,
        0.31432924],
       [0.38701276, 0.63129855, 0.13302966, …, 0.62050709, 1.        ,
        0.40435633],
       [0.32375144, 0.56011353, 0.25306592, …, 0.31432924, 0.40435633,
        1.        ]])
```

Now it's time to test the model to determine how well it performed.

[11]:
```
predictions = basic.test(testset)
```

[12]:
```
print(accuracy.rmse(predictions))
```

```
RMSE: 4.2162
4.216159622998984
```

Not a particularly amazing model…. As you can see, the model had an RMSE of about 4.5, meaning that it was off by roughly 4 points for each guess it made for ratings. Not horrendous when you consider we're working on a range of 20 points, but let's see if we can improve it. To begin with, let's try with a different similarity metric (pearson correlation) and evaluate our RMSE.

[13]:
```
sim_pearson = {'name':'pearson', 'user_based':False}
basic_pearson = knns.KNNBasic(sim_options=sim_pearson)
basic_pearson.fit(trainset)
predictions = basic_pearson.test(testset)
print(accuracy.rmse(predictions))
```

```
Computing the pearson similarity matrix…
Done computing similarity matrix.
RMSE: 4.2789
4.2788659590605915
```

Pearson correlation seems to have performed better than cosine similarity in the basic KNN model, we can go ahead and use pearson correlation as our similarity metric of choice for future models. The next model we're going to try is KNN with Means. This is the same thing as the basic KNN model, except it takes into account the mean rating of each user or item depending on whether you are performing user-user or item-item similarities, respectively.

[14]:
```
sim_pearson = {'name':'pearson', 'user_based':False}
knn_means = knns.KNNWithMeans(sim_options=sim_pearson)
knn_means.fit(trainset)
predictions = knn_means.test(testset)
print(accuracy.rmse(predictions))
```

```
Computing the pearson similarity matrix…
Done computing similarity matrix.
RMSE: 4.1376
4.137603498545671
```

A little better… let's try one more neighborhood-based method before moving into model-based methods. Let's try the KNNBaseline method. This is a more advanced method because it adds in a bias term that is calculated by way of minimizing a cost function represented by:

$$\sum_{r_{ui} \in R_{\text{train}}} (\hat{r}_{ui} - (\mu + b_i + b_u))^2 + \lambda(b_u^2 + b_i^2)$$

With b_i and b_u being biases for items and users respectively and μ referring to the global mean.

[15]:
```
sim_pearson = {'name':'pearson', 'user_based':False}
knn_baseline = knns.KNNBaseline(sim_options=sim_pearson)
knn_baseline.fit(trainset)
predictions = knn_baseline.test(testset)
print(accuracy.rmse(predictions))
```

```
Estimating biases using als…
Computing the pearson similarity matrix…
Done computing similarity matrix.
RMSE: 4.1333
4.133261062634373
```

Even better! Now let's see if we can get some insight by applying some matrix factorization techniques!

1.4 Model-Based Methods (Matrix Factorization)

It's worth pointing out that when SVD is calculated for recommendation systems, it is preferred to be done with a modified version called "Funk's SVD" that only takes into account the rated values, ignoring whatever items have not been rated by users. The algorithm is named after Simon Funk, who was part of the team who placed 3rd in the Netflix challenge with this innovative way of performing matrix decomposition. Read more about Funk's SVD implementation at his original blog post. There is no simple way to include for this fact with SciPy's implementation of `svd()`, but luckily the `surprise` library has Funk's version of SVD implemented to make our lives easier!

Similar to other `sklearn` features, we can expedite the process of trying out different parameters by using an implementation of grid search. Let's make use of the grid search here to account for some different configurations of parameters within the SVD pipeline. This might take some time! You'll notice that the `n_jobs` is parameter set to -1, which ensures that all of the cores on your computer will be used to process fitting and evaluating all of these models. To help keep track of what is occurring here, take note of the different values. This code ended up taking over 16 minutes to complete even with parallelization in effect, so the optimal parameters are given to you for the SVD model below. Use them to train a model and let's see how well it performs. If you want the full grid search experience, feel free to uncomment the code and give it a go!

The optimal parameters are :

```
{'n_factors': 100, 'n_epochs': 10, 'lr_all': 0.005, 'reg_all': 0.4}
```

[16]:
```
from surprise.prediction_algorithms import SVD
from surprise.model_selection import GridSearchCV

# param_grid = {'n_factors':[20, 100],'n_epochs': [5, 10], 'lr_all': [0.002, 0.
 ↪005],
#               'reg_all': [0.4, 0.6]}
# gs_model = GridSearchCV(SVD,param_grid=param_grid,n_jobs =␣
 ↪-1,joblib_verbose=5)
# gs_model.fit(jokes)
```

[17]:
```
svd = SVD(n_factors=100, n_epochs=10, lr_all=0.005, reg_all=0.4)
svd.fit(trainset)
predictions = svd.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 4.2620
4.2619629532526
```

Interestingly, this model performed worse than the others! In general, the advantages of matrix factorization starts to show itself when the size of the dataset becomes massive. At that point, the storage challenges increase for the memory-based models, and there is enough data for latent factors to become extremely apparent.

1.5 Making Predictions

Now that we've explored some models, we can think about how we might fit the models into the context of an application. To begin with, let's access some basic functionality of `surprise` models to get predicted ratings for a given user and item. All that's needed are the `user_id` and `item_id` for which you want to make a prediction. Here we're making a prediction for user 34 and item 25 using the SVD model we just fit.

[18]:
```
user_34_prediction = svd.predict('34', '25')
user_34_prediction
```

[18]:
```
Prediction(uid='34', iid='25', r_ui=None, est=2.6218240217111184,
details={'was_impossible': False})
```

The output of the prediction is a tuple. Here, we're going to access the estimated rating.

[19]:
```
user_34_prediction[3]
```

[19]:
```
2.6218240217111184
```

You might be wondering, "OK I'm making predictions about certain items rated by certain users, but how can I actually give certain N recommendations to an individual user?" Although `surprise` is a great library, it does not have this recommendation functionality built into it, but in the next lab, you will get some experience not only fitting recommendation system models, but also programmatically retrieving recommended items for each user.

1.5.1 Sources

Jester dataset originally obtained from:

Eigentaste: A Constant Time Collaborative Filtering Algorithm. Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Information Retrieval, 4(2), 133-151. July 2001.

1.5.2 Additional Resources

- Surprise Documentation
- Surprise Tutorial

1.6 Summary

You now should have an understanding of the basic considerations one should take note of when coding a recommendation system as well as how to implement them in different ways using `surprise`. In the upcoming lab, you will be tasked with fitting models using `surprise` and then retrieving those predicted values in a meaningful way to give recommendations to people. Let's see how well it works in action.