



# Tutorial: Practical Introduction to Recommender Systems

Cambridge Spark
Oct 11, 2018 · 10 min read

## Introduction

Recommender systems are a vital tool in a data scientists' toolbox. The aim is simple, given data on customers and items they've bought, automatically make recommendations of other products they'd like. You can see these systems in action on a lot of websites (for example Amazon), and it's not just limited to physical products, they can be used for any customer interaction. Recommender systems were introduced in a previous Cambridge Spark tutorial.

In this tutorial, we want to extend the previous article by showing you how to build recommender systems in python using cutting-edge algorithms. Many traditional

our model will be good at fitting to the data we have, but bad at recommending new products to customers (not ideal given that is their purpose). This is because there is so much missing information. Before getting into the code, we'll give more details about recommender systems, why there is so much missing information, and its solution.

## Recommender Systems and Matrix Factorisation

The data input for a recommender system can be thought of as a large matrix, with the rows indicating an entry for a customer, and the columns indicating an entry for a particular item. Let's call this matrix $R$. Then entry $Rij$ will contain the score that customer $i$ has given to product $j$. For example if it's a review this could be a number from 1–5, or it might just be 0–1 indicating if a user has bought an item or not. This matrix contains a lot of missing information, it's unlikely a customer has bought every item on Amazon! Recommender systems aim to fill in this missing information, by predicting the customer score of items where the score is missing. Then recommender systems will recommend items to the customer that have the highest score. A typical example of the matrix $R$ with entries that are review values from 1–5 is given in the picture below.

| | Item 1 | Item 2 | ... | Item 99 | Item 100 |
|---|---|---|---|---|---|
| **Customer 1** | 5 | NA | ... | NA | 3 |
| **Customer 2** | NA | 2 | ... | 3 | NA |
| **...** | ... | ... | ... | ... | ... |
| **Customer 49** | 2 | 3 | ... | NA | 4 |
| **Customer 50** | 1 | NA | ... | NA | NA |

The Netflix challenge was a competition designed to find the best algorithms for recommender systems. It was run by Netflix using their movie data. During the challenge, one type of algorithm stood out for its excellent performance, and has remained popular ever since. It is known as *matrix factorisation*. This method works by

Suppose that R has dimension $\boldsymbol{d}_1 \times \boldsymbol{d}_2$, then U will have dimension $\boldsymbol{D} \times \boldsymbol{d}_1$ and V will have dimension $\boldsymbol{D} \times \boldsymbol{d}_2$. Here $\boldsymbol{D}$ is chosen by the user, it needs to be large enough to encode the nuances of $R$, but making it too large will make performance slow and could lead to overfitting. A typical size of $\boldsymbol{D}$ is 20.

While at first glance, this factorisation seems quite easy, it is made much more difficult by all the missing data. Imputing the data might work, but it makes the methods very slow. Instead, most popular methods focus only on the matrix entries $Rij$ that are known, and fit the factorisation to minimise the error of these known $Rij$. A problem with doing this though is that predictions will be bad because of overfitting. The methods get around this by using a procedure known as *regularisation*, which is a common way to reduce overfitting. For more details about the workings of the methods, please see the further reading at the end of this post.

## The Package

In this tutorial, we'll use the surprise package, a popular package for building recommendation systems in Python. Mac and Linux users can install this package by opening a terminal and running `pip install surprise`. Windows users can install it using conda `conda install -c conda-forge scikit-surprise`. The package can then be imported in the standard way,

```
In [2]: import surprise
```

## The Dataset

In this tutorial, we'll work with the librec FilmTrust dataset, originally collated for a particular recommender systems paper. The dataset contains 35497 movie ratings from various users of the FilmTrust platform. We chose this dataset as it is relatively small, so examples should run quite quickly. The package `surprise` has a number of datasets built in, but we chose this one as it allows us to demonstrate how to load custom datasets into the package. Normally, recommender systems will use larger datasets than this, so for more challenging datasets we recommend investigating the grouplens website, which has a variety of free datasets available.

In [3]:
```
import numpy as np
import pandas as pd
import urllib
import io
import zipfile

# Download zip file
tmpFile = urllib.request.urlopen('https://www.librec.net/datasets/filmtrust.zip')
# Unzip file
tmpFile = zipfile.ZipFile(io.BytesIO(tmpFile.read()))
# Open desired data file as pandas dataframe, close zipfile
dataset = pd.read_table(io.BytesIO(tmpFile.read('ratings.txt')), sep = ' ', names = ['uid', 'iid', 'rating'])
tmpFile.close()

dataset.head()
```

Out[3]:

| | uid | iid | rating |
|---|---|---|---|
| 0 | 1 | 1 | 2.0 |
| 1 | 1 | 2 | 4.0 |
| 2 | 1 | 3 | 3.5 |
| 3 | 1 | 4 | 3.0 |
| 4 | 1 | 5 | 4.0 |

As you can see this dataset does not really look like the matrix RR. This is because there are so many missing values, so it is much easier to save the file in a *sparse* format. In a sparse format, the first column is the row number of the matrix ii; the second column is the column number of the matrix jj; and the third row is the matrix entry RijRij. For this dataset, the first column is the user ID, the second is the ID of the movie they've reviewed, and the third column is their review score. This sparse format is also the input that matrix factorisation methods require, rather than the full matrix RR, this is because they only use the non-missing matrix entries.

## Fitting the Model

Now it's time to start using the package. First we need to load the dataset into the package `surprise`, this is done using the `Reader` class. The main thing the `Reader` class does is to specify the range of the reviews. Let's first check the range of the reviews for this dataset.

In [4]:
```
lower_rating = dataset['rating'].min()
upper_rating = dataset['rating'].max()
print('Review range: {0} to {1}'.format(lower_rating, upper_rating))
```

```
Review range: 0.5 to 4.0
```

So our review range goes from 0.5 to 4, which is a little non-standard (the default for `surprise` is 1-5). So we will need to change this when we load in our dataset, which is done like this:

We will use the method `SVD++`, one of best performers in the Netflix challenge, which has now become a popular method for fitting recommender systems. As we mentioned earlier, this method extends vanilla SVD algorithms such as the one covered in the previous blog post by only optimising known terms and performing regularisation (note that the method `SVD` in `surprise` is much more sophisticated than vanilla SVD, and much more similar to `SVD++`). More details on method specifics can be found in the further reading. A simple recommender system built with the `SVD++` can be coded as follows:

In [6]:
```
alg = surprise.SVDpp()
output = alg.fit(data.build_full_trainset())
```

For now we've just trained the model on the whole dataset, which is not good practice but we do it just to give you an idea of how the models and predictions work. Later on we'll cover proper testing and evaluation; as well as hyperparameter tuning to maximise performance.

Now we've fitted the model, we can check the predicted score of, for example, user 50 on a music artist 52 using the `predict` method.

In [7]:
```
# The uids and iids should be set as strings
pred = alg.predict(uid='50', iid='52')
score = pred.est
print(score)
```

```
3.0028030537791928
```

So in this case the estimate was a score of 3. But in order to recommend the best products to users, we need to find `n` items that have the highest predicted score. We'll do this in the next section.

## Making Recommendations

Let's make our recommendations to a particular user. Let's focus on uid 50 and find one item to recommend them. First we need to find the movie ids that user 50 didn't rate, since we don't want to recommend them a movie they've already watched!

In [8]:
```
# Get a list of all movie ids
iids = dataset['iid'].unique()
# Get a list of iids that uid 50 has rated
iids50 = dataset.loc[dataset['uid'] == 50, 'iid']
# Remove the iids that uid 50 has rated from the list of all movie ids
iids_to_pred = np.setdiff1d(iids, iids50)
```

find the best one. For this we have to create another dataset with the iids we want to predict in the sparse format as before of: `uid`, `iid`, `rating`. We'll just arbitrarily set all the ratings of this test set to 4, as they are not needed. Let's do this, then output the first prediction.

```
In [9]: testset = [[50, iid, 4.] for iid in iids_to_pred]
        predictions = alg.test(testset)
        predictions[0]

Out[9]: Prediction(uid=50, iid=14, r_ui=4.0, est=3.1692295573778413, details={'was_impossible': False})
```

As you can see from the output, each prediction is a special object. In order to find the best, we'll convert this object into an array of the predicted ratings. We'll then use this to find the iid with the best predicted rating.

```
In [10]: pred_ratings = np.array([pred.est for pred in predictions])
         # Find the index of the maximum predicted rating
         i_max = pred_ratings.argmax()
         # Use this to find the corresponding iid to recommend
         iid = iids_to_pred[i_max]
         print('Top item for user 50 has iid {0} with predicted rating {1}'.format(iid, pred_ratings[i_max]))

         Top item for user 50 has iid 126 with predicted rating 4.0
```

When you implement your own recommender system you will normally have metadata which allows you to get, for example the name of the film from the iid code. Unfortunately, this dataset does not include this information, but many other larger datasets do, such as the movielens dataset.

Similarly you can get the top `n` items for user 50, just replace the `argmax()` method with the `argpartition()` method as per this stackoverflow question.

## Tuning and Evaluating the Model

As you probably already know, it is bad practice to fit a model on the whole dataset without checking its performance and tuning parameters which affect the fit. So for the remainder of the tutorial we'll show you how to tune the parameters of `SVD++` and evaluate the performance of the method. The method `SVD++`, as well as most other matrix factorisation algorithms, will depend on a number of main tuning constants: the dimension DD affecting the size of UU and VV; the *learning rate*, which affects the performance of the optimisation step; the *regularisation term* affecting the overfitting of the model; and the number of epochs, which determines how many iterations of optimisation are used.

for all the learning rate values, and another for all the regularisation terms, so we'll do this for speed. In `surprise`, tuning is performed using a function called `GridSearchCV`, which picks the constants which perform the best at predicting a held out testset. This means constant values to try need to be predefined.

First let's define our list of constant values to check, typically the learning rate is a small value between 0 and 1. In theory, the regularisation parameter can be any positive real value, but in practice it is limited as setting it too small will result in overfitting, while setting it too large will result in poor performance; so trying a list of reasonable values should be fine. The `GridSearchCV` function can then be used to determine the best performing parameter values using cross validation. We've chosen quite a limited list since this code can take a while to run, as it has to fit multiple models with different parameters.

In [11]:
```
param_grid = {'lr_all' : [.001, .01], 'reg_all' : [.1, .5]}
gs = surprise.model_selection.GridSearchCV(surprise.SVDpp, param_grid, measures=['rmse', 'mae'], cv=3)
gs.fit(data)
# Print combination of parameters that gave best RMSE score
print(gs.best_params['rmse'])
```

```
{'lr_all': 0.01, 'reg_all': 0.1}
```

The output prints the combination of parameters that gets the best RMSE on a held out test set, RMSE is a way of measuring the prediction error. In this case, we've only checked a few tuning constant values, because these procedures can take a while to run. But typically you will try out as many values as possible to get the best performance you can.

The performance of a particular model you've chosen can be evaluated using cross validation. This might be used to compare a number of methods for example, or just to check your method is performing reasonably. This can be done by running the following:

In [12]:
```
alg = surprise.SVDpp(lr_all = .001) # parameter choices can be added here.
output = surprise.model_selection.cross_validate(alg, data, verbose = True)
```

```
Evaluating RMSE, MAE of algorithm SVDpp on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8290  0.8358  0.8249  0.8297  0.8210  0.8281  0.0050
MAE (testset)     0.6532  0.6587  0.6551  0.6558  0.6542  0.6554  0.0019
Fit time          16.96   16.16   16.07   16.20   16.03   16.28   0.34
Test time         0.35    0.35    0.35    0.40    0.35    0.36    0.02
```

Now you know how to build high performing recommender systems using the Python package `surprise`, as well as using this to generate predictions for users, and how to tune your system for maximum performance. I recommend checking out the documentation for `surprise` to help really get to grips with the package, maybe play around with the tuning constants to get to know how to maximise performance of your recommendation system, as well as trying the package out with more complicated or larger datasets.

For those that want to know more, outlined here is some recommended reading:

- Get the top 10 scoring items for each user: package FAQ for `surprise`
- Review paper of matrix factorisation methods for recommender systems
- Repository of matrix factorisation datasets at Grouplens
- Another article on matrix factorisation which has more detail on the underlying maths

. . .

# About the Author

Jack Baker is a statistics student, undertaking his PhD at the University of Lancaster, in collaboration with the University of Washington. His research focuses on scalable Bayesian inference.

**Connect with Jack**

. . .

Thanks for reading! If you enjoyed the post, we'd appreciate your support by applauding via the clap (👏) button below or by sharing this article so others can find it.

Keep a look out for our upcoming posts and Data Science tutorials! Busy schedule? Click here to register to receive our bi-weekly newsletter to get our posts direct to your inbox.

. . .

## Further tutorials to practice your skills on:

- Deploying a Machine Learning Model to the web
  - Six Data Science projects to expand your skills and knowledge
  - Unit testing with PySpark
  - Hyperparameter tuning in XGBoost
  - Getting started with XGBoost