Published in Towards Data Science

Susan Li Follow

Dec 26, 2018 · 6 min read · Listen

Photo credit: Pixabay

Building and Testing Recommender Systems With Surprise, Step-By-Step

Learn how to build your own recommendation engine with the help of Python and Surprise Library, Collaborative Filtering

Recommender systems are one of the most common used and easily understandable applications of data science. Lots of work has been done on this topic, the interest and demand in this area remains very high because of the rapid growth of the internet and the information overload problem. It has become necessary for online businesses to help users to deal with information overload and provide personalized recommendations, content and services to them.

Two of the most popular ways to approach recommender systems are collaborative filtering and content-based recommendations. In this post, we will focus on the collaborative filtering approach, that is: the user is recommended items that people with similar tastes and preferences liked in the past. In another word, this method predicts unknown ratings by using the similarities between users.

We'll be working with the Book-Crossing, a book ratings data set to develop recommendation system algorithms, with the Surprise library, which was built by Nicolas Hug. Let's get started!

The Data

The Book-Crossing data comprises three tables, we will use two of them: The users table and the book ratings table.

```
user = pd.read_csv('BX-Users.csv', sep=';', error_bad_lines=False,
encoding="latin-1")
user.columns = ['userID', 'Location', 'Age']
rating = pd.read_csv('BX-Book-Ratings.csv', sep=';',
error_bad_lines=False, encoding="latin-1")
rating.columns = ['userID', 'ISBN', 'bookRating']
df = pd.merge(user, rating, on='userID', how='inner')
df.drop(['Location', 'Age'], axis=1, inplace=True)
df.head()
```

	userID	ISBN	bookRating
0	2	0195153448	0
1	7	034542252	0
2	8	0002005018	5
3	8	0060973129	0
4	8	0374157065	0

Figure 1

EDA

Ratings Distribution

```python
from plotly.offline import init_notebook_mode, plot, iplot
import plotly.graph_objs as go
init_notebook_mode(connected=True)

data = df['bookRating'].value_counts().sort_index(ascending=False)
trace = go.Bar(x = data.index,
               text = ['{:.1f} %'.format(val) for val in (data.values / df.shape[0] * 100)],
               textposition = 'auto',
               textfont = dict(color = '#000000'),
               y = data.values,
               )
# Create layout
layout = dict(title = 'Distribution Of {} book-ratings'.format(df.shape[0]),
              xaxis = dict(title = 'Rating'),
              yaxis = dict(title = 'Count'))
# Create plot
fig = go.Figure(data=[trace], layout=layout)
iplot(fig)
```


ratings_distribution.py

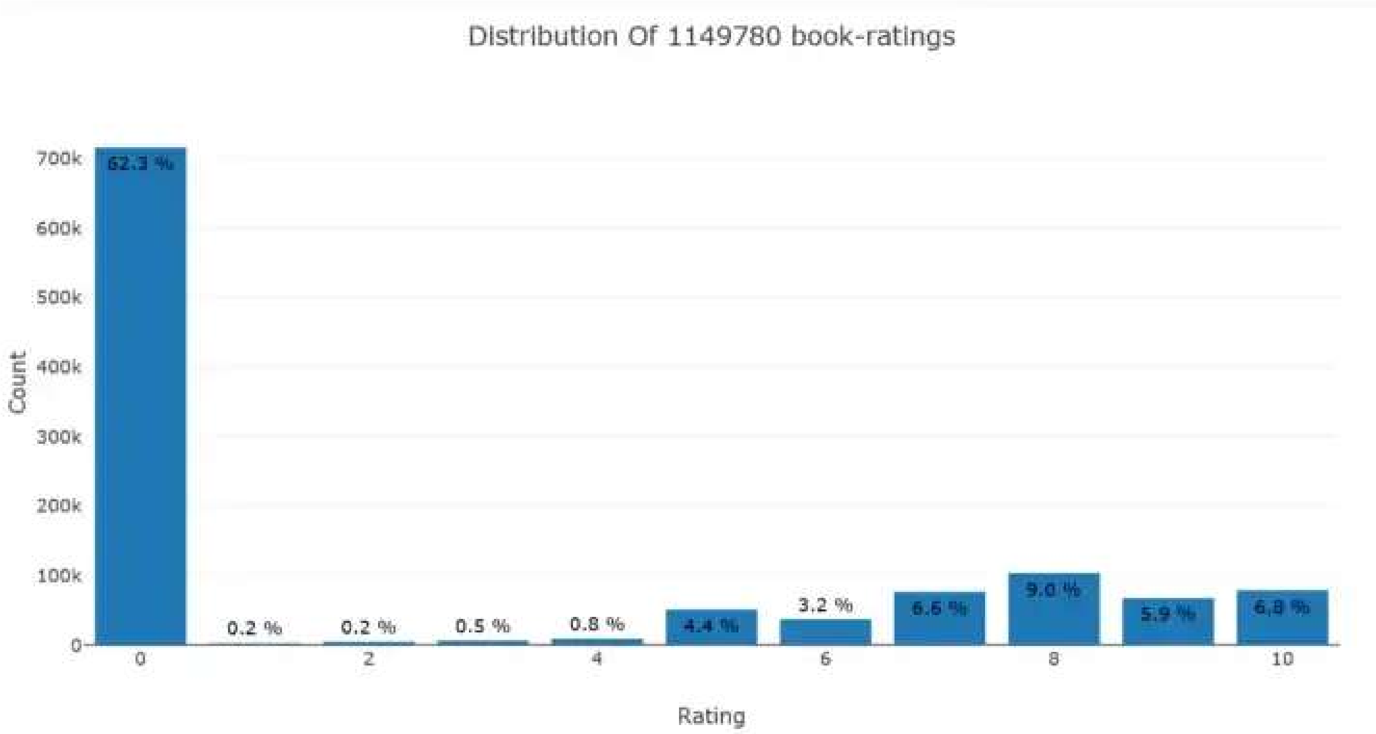

Figure 2

We can see that over 62% of all ratings in the data are 0, and very few ratings are 1 or 2, or 3, low rating books mean they are generally really bad.

Ratings Distribution By Book

```
# Number of ratings per book
data = df.groupby('ISBN')['bookRating'].count().clip(upper=50)

# Create trace
trace = go.Histogram(x = data.values,
                     name = 'Ratings',
                     xbins = dict(start = 0,
                                  end = 50,
                                  size = 2))
# Create layout
layout = go.Layout(title = 'Distribution Of Number of Ratings Per Book (Clipped at 100)',
                   xaxis = dict(title = 'Number of Ratings Per Book'),
                   yaxis = dict(title = 'Count'),
                   bargap = 0.2)

# Create plot
fig = go.Figure(data=[trace], layout=layout)
iplot(fig)
```

ratings_distribution_by_book.py hosted with ❤ by GitHub view raw

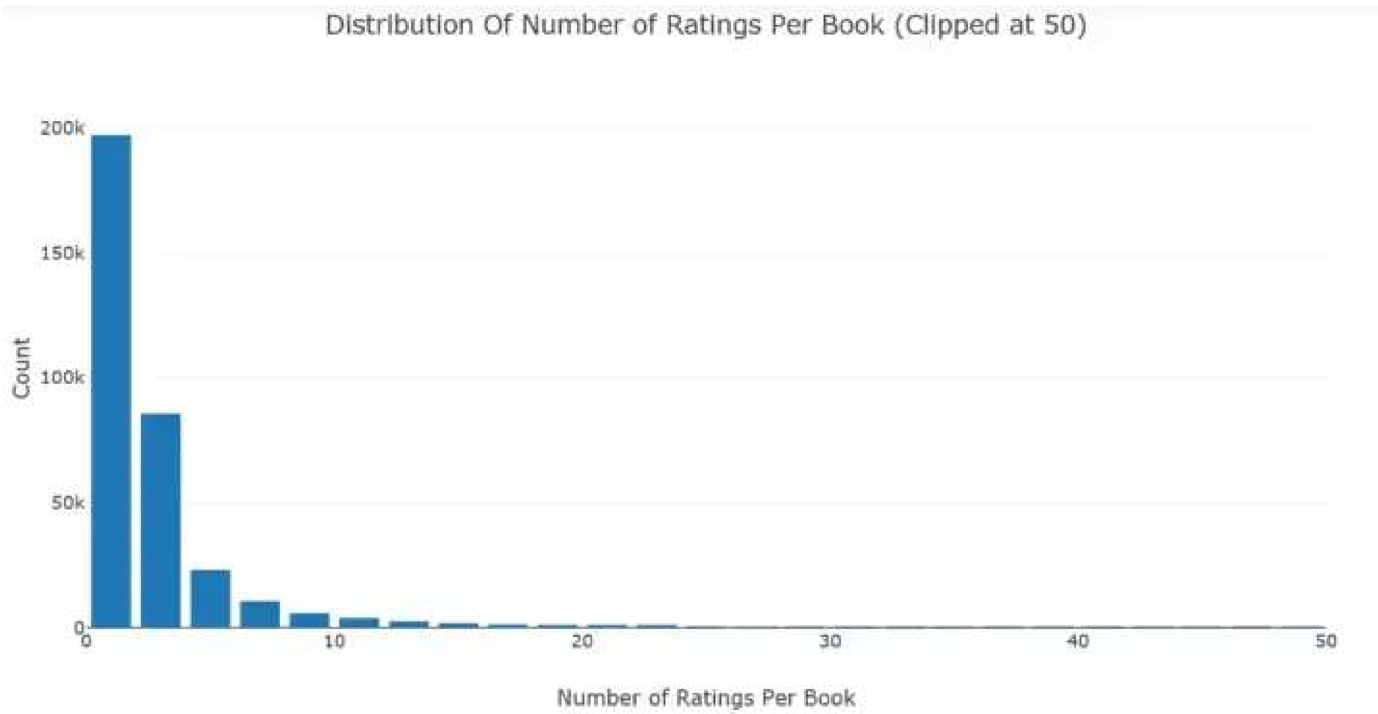

Figure 3

```
df.groupby('ISBN')
['bookRating'].count().reset_index().sort_values('bookRating',
ascending=False)[:10]
```

	ISBN	bookRating
247408	0971880107	2502
47371	0316666343	1295
83359	0385504209	883
9637	0060928336	732
41007	0312195516	723
101670	044023722X	647
166705	0679781587	639
28153	0142001740	615
166434	067976402X	614
153620	0671027360	586

Figure 4

Most of the books in the data received less than 5 ratings, and very few books have many ratings, although the most rated book has received 2,502 ratings.

Ratings Distribution By User

```
# Number of ratings per user
data = df.groupby('userID')['bookRating'].count().clip(upper=50)

# Create trace
trace = go.Histogram(x = data.values,
                     name = 'Ratings',
                     xbins = dict(start = 0,
                                  end = 50,
                                  size = 2))
# Create layout
layout = go.Layout(title = 'Distribution Of Number of Ratings Per User (Clipped at 50)',
                   xaxis = dict(title = 'Ratings Per User'),
                   yaxis = dict(title = 'Count'),
                   bargap = 0.2)

```

```
# Create plot
fig = go.Figure(data=[trace], layout=layout)
```

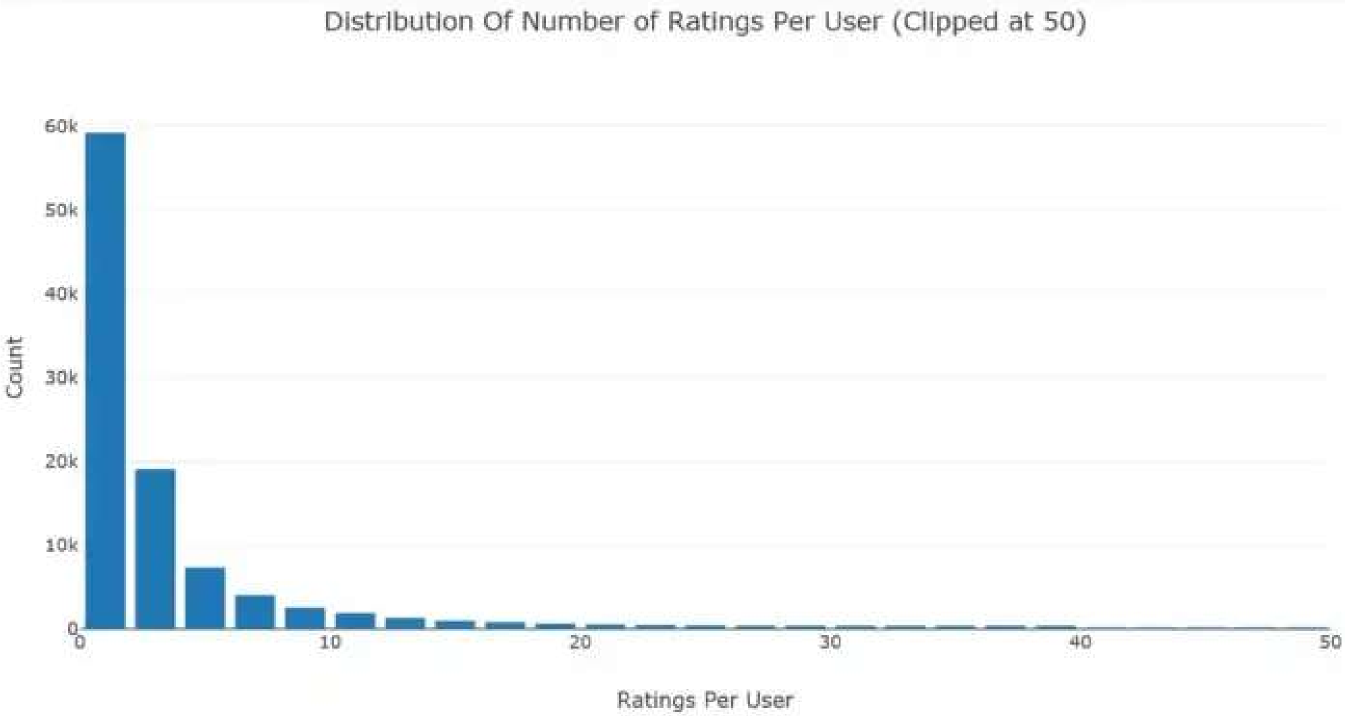

Figure 5

```
df.groupby('userID')['bookRating'].count().reset_index().sort_values('bookRating', ascending=False)[:10]
```

	userID	bookRating
4213	11676	13602
74815	198711	7550
58113	153662	6109
37356	98391	5891
13576	35859	5850
80185	212898	4785
105111	278418	4533
28884	76352	3367
42037	110973	3100
88584	235105	3067

Figure 6

Most of the users in the data gave less than 5 ratings, and not many users gave many ratings, although the most productive user have given 13,602 ratings.

I'm sure you have noticed that the above two plots share the same distribution. The number of ratings per book and the number of ratings per user decay exponentially.

To reduce the dimensionality of the data set, and avoid running into "memory error", we will filter out rarely rated books and rarely rating users.

```
min_book_ratings = 50
filter_books = df['ISBN'].value_counts() > min_book_ratings
filter_books = filter_books[filter_books].index.tolist()

min_user_ratings = 50
filter_users = df['userID'].value_counts() > min_user_ratings
filter_users = filter_users[filter_users].index.tolist()

df_new = df[(df['ISBN'].isin(filter_books)) & (df['userID'].isin(filter_users))]
print('The original data frame shape:\t{}'.format(df.shape))
```

```
print('The new data frame shape:\t{}'.format(df_new.shape))
```

filter_dataframe.py hosted with ❤ by GitHub view raw

```
The original data frame shape:  (1149780, 3)
The new data frame shape:       (140516, 3)
```

Figure 7

Surprise

To load a data set from the above pandas data frame, we will use the `load_from_df()` method, we will also need a `Reader` object, and the `rating_scale` parameter must be specified. The data frame must have three columns, corresponding to the user ids, the item ids, and the ratings in this order. Each row thus corresponds to a given rating.

```
reader = Reader(rating_scale=(0, 9))
data = Dataset.load_from_df(df_new[['userID', 'ISBN', 'bookRating']],
reader)
```

With the Surprise library, we will benchmark the following algorithms:

Basic algorithms

NormalPredictor

- `NormalPredictor` algorithm predicts a random rating based on the distribution of the training set, which is assumed to be normal. This is one of the most basic algorithms that do not do much work.

BaselineOnly

- `BaselineOnly` algorithm predicts the baseline estimate for given user and item.

k-NN algorithms

KNNBasic

- `KNNBasic` is a basic collaborative filtering algorithm.

KNNWithMeans

- `KNNWithMeans` is basic collaborative filtering algorithm, taking into account the mean ratings of each user.

KNNWithZScore

- `KNNWithZScore` is a basic collaborative filtering algorithm, taking into account the z-score normalization of each user.

KNNBaseline

- `KNNBaseline` is a basic collaborative filtering algorithm taking into account a baseline rating.

Matrix Factorization-based algorithms

SVD

- `SVD` algorithm is equivalent to Probabilistic Matrix Factorization

SVDpp

- The `SVDpp` algorithm is an extension of SVD that takes into account implicit ratings.

NMF

- `NMF` is a collaborative filtering algorithm based on Non-negative Matrix Factorization. It is very similar with SVD.

Slope One

- `SlopeOne` is a straightforward implementation of the SlopeOne algorithm.

Co-clustering

- `Coclustering` is a collaborative filtering algorithm based on co-clustering.

We use “rmse” as our accuracy metric for the predictions.

```
benchmark = []
# Iterate over all algorithms
for algorithm in [SVD(), SVDpp(), SlopeOne(), NMF(), NormalPredictor(), KNNBaseline(), KNNBasic(), KNNW
    # Perform cross validation
    results = cross_validate(algorithm, data, measures=['RMSE'], cv=3, verbose=False)

    # Get results & append algorithm name
    tmp = pd.DataFrame.from_dict(results).mean(axis=0)
    tmp = tmp.append(pd.Series([str(algorithm).split(' ')[0].split('.')[-1]], index=['Algorithm']))
    benchmark.append(tmp)

pd.DataFrame(benchmark).set_index('Algorithm').sort_values('test_rmse')
```

benchmark.py hosted with ❤ by GitHub view raw

KNNBaseline	0.880004	3.490825	5.455574
KNNWithZScore	0.862389	3.508873	4.919546
SVD	5.547440	3.541042	0.306871
KNNBasic	0.667804	3.725668	4.032472
SVDpp	136.691374	3.790323	4.714200
NMF	6.082658	3.843718	0.309214
NormalPredictor	0.114894	4.665311	0.308486

Figure 8

Train and Predict

`BaselineOnly` algorithm gave us the best rmse, therefore, we will train and predict with `BaselineOnly` and use Alternating Least Squares (ALS).

```
print('Using ALS')
bsl_options = {'method': 'als',
               'n_epochs': 5,
               'reg_u': 12,
               'reg_i': 5
               }
```

```
algo = BaselineOnly(bsl_options=bsl_options)
cross_validate(algo, data, measures=['RMSE'], cv=3, verbose=False)
```

```
Using ALS
Estimating biases using als...
Estimating biases using als...
Estimating biases using als...

{'fit_time': (0.13807177543640137, 0.12630414962768555, 0.1693267822265625),
 'test_rmse': array([ 3.37381566,  3.36756676,  3.37800743]),
 'test_time': (0.2851989269256592, 0.322648286819458, 0.3984529972076416)}
```

Figure 9

algorithm on the trainset, and the `test()` method which will return the predictions made from the testset.

```
trainset, testset = train_test_split(data, test_size=0.25)
algo = BaselineOnly(bsl_options=bsl_options)
predictions = algo.fit(trainset).test(testset)
accuracy.rmse(predictions)
```

```
Estimating biases using als...
RMSE: 3.3581
```

Figure 10

To inspect our predictions in details, we are going to build a pandas data frame with all the predictions. The following code were largely taken from this notebook.

predictions_details.py

Best Predictions:

	uid	iid	rui	est	details	Iu	Ui	err
13857	269566	0061098795	0.0	0.0	{'was_impossible': False}	276	30	0.0
14688	102967	051512317X	0.0	0.0	{'was_impossible': False}	384	59	0.0
14689	238781	0451203895	0.0	0.0	{'was_impossible': False}	178	76	0.0
26302	63938	0380817446	[illegible]	[illegible]	[illegible] False}	71	26	0.0
14712	244736	0061098795	[illegible]	[illegible]	[illegible] False}	77	30	0.0
14720	278418	0743460529	0.0	0.0	{'was_impossible': False}	174	51	0.0
2771	170518	080411868X	0.0	0.0	{'was_impossible': False}	155	105	0.0
14737	238545	0440241073	0.0	0.0	{'was_impossible': False}	41	146	0.0
26275	238120	0553297260	0.0	0.0	{'was_impossible': False}	314	34	0.0
26273	36836	0394742117	0.0	0.0	{'was_impossible': False}	158	25	0.0

Figure 11

The above are the best predictions, and they are not lucky guesses. Because Ui is anywhere between 25 to 146, they are not really small, meaning that significant number of users have rated the target book.

Worst predictions:

	uid	iid	rui	est	details	Iu	Ui	err
4430	263460	0061097101	10.0	0.317065	{'was_impossible': False}	61	88	9.682935
12250	129358	0515128546	10.0	0.314570	{'was_impossible': False}	97	80	9.685430
33088	35857	0380710722	10.0	0.285230	{'was_impossible': False}	191	59	9.714770
1934	78834	0399145990	10.0	0.279658	{'was_impossible': False}	154	17	9.720342
2419	226006	0425100650	10.0	0.260445	{'was_impossible': False}	14	42	9.739555
29657	14521	0553275976	10.0	0.169291	{'was_impossible': False}	156	84	9.830709
2794	14521	0553269631	10.0	0.070703	{'was_impossible': False}	156	27	9.929297
25532	115490	081297106X	10.0	0.028978	{'was_impossible': False}	159	41	9.971022
30944	182442	0679433740	10.0	0.000000	{'was_impossible': False}	36	33	10.000000
5395	26544	055358264X	10.0	0.000000	{'was_impossible': False}	191	47	10.000000

Figure 12

The worst predictions look pretty surprise. Let's look in more details of the last one ISBN "055358264X". The book was rated by 47 users, user "26544" rated 10, our BaselineOnly algorithm predicts this user would rate 0.

```
import matplotlib.pyplot as plt
%matplotlib notebook

df_new.loc[df_new['ISBN'] == '055358264X']['bookRating'].hist()
plt.xlabel('rating')
plt.ylabel('Number of ratings')
plt.title('Number of ratings book ISBN 055358264X has received')
plt.show();
```

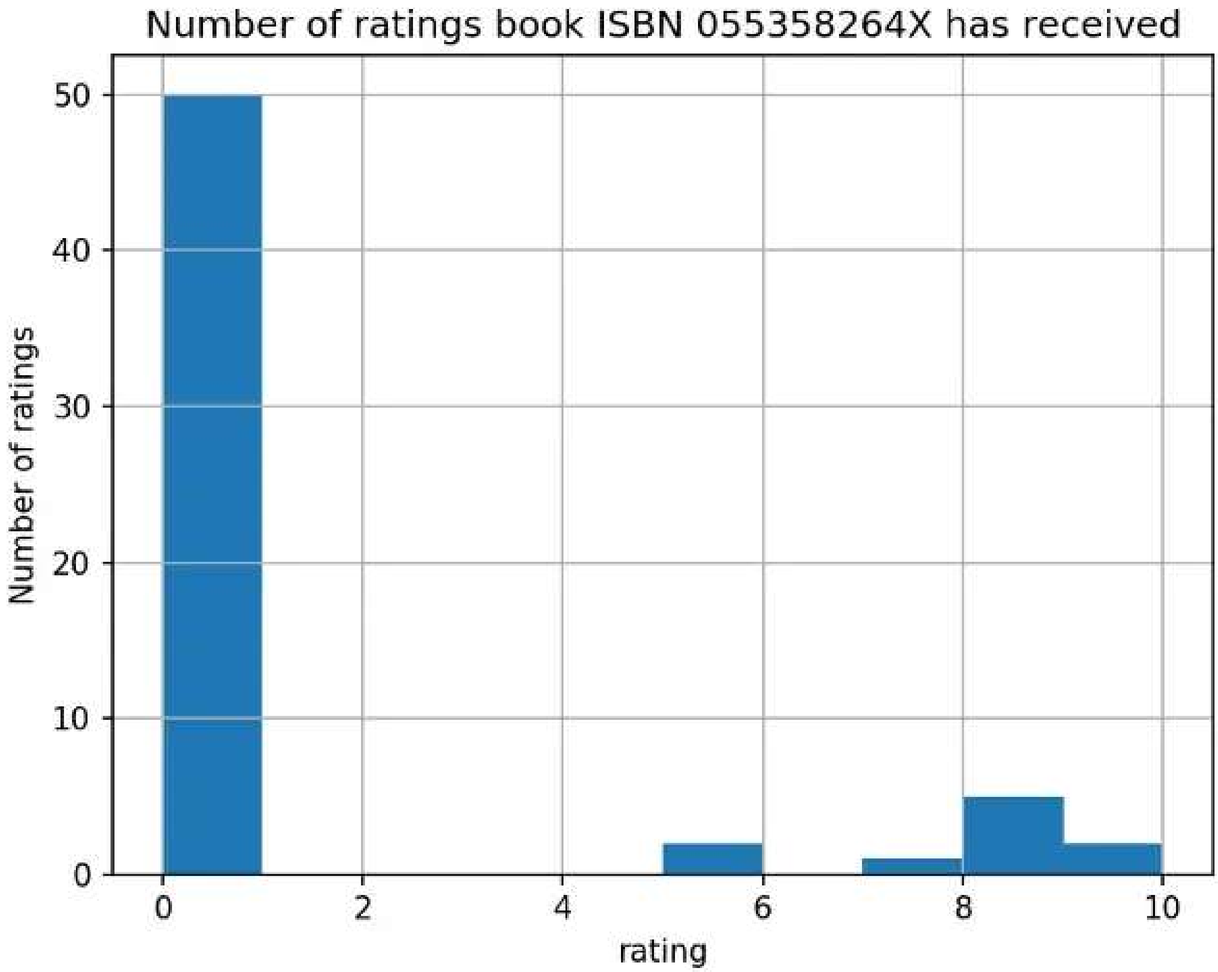

Figure 13

It turns out, most of the ratings this book received was 0, in another word, most of the users in the data rated this book 0, only very few users rated 10. Same with the other predictions in "worst predictions" list. It seems that for each prediction, the users are some kind of outsiders.

That was it! I hope you enjoyed the recommendation (or rather, a rating prediction) journey with Surprise. Jupyter notebook can be found on Github. Happy Holidays!

Reference: Surprise' documentation

Data Science Machine Learning Recommendation System Python

Collaborative Filtering