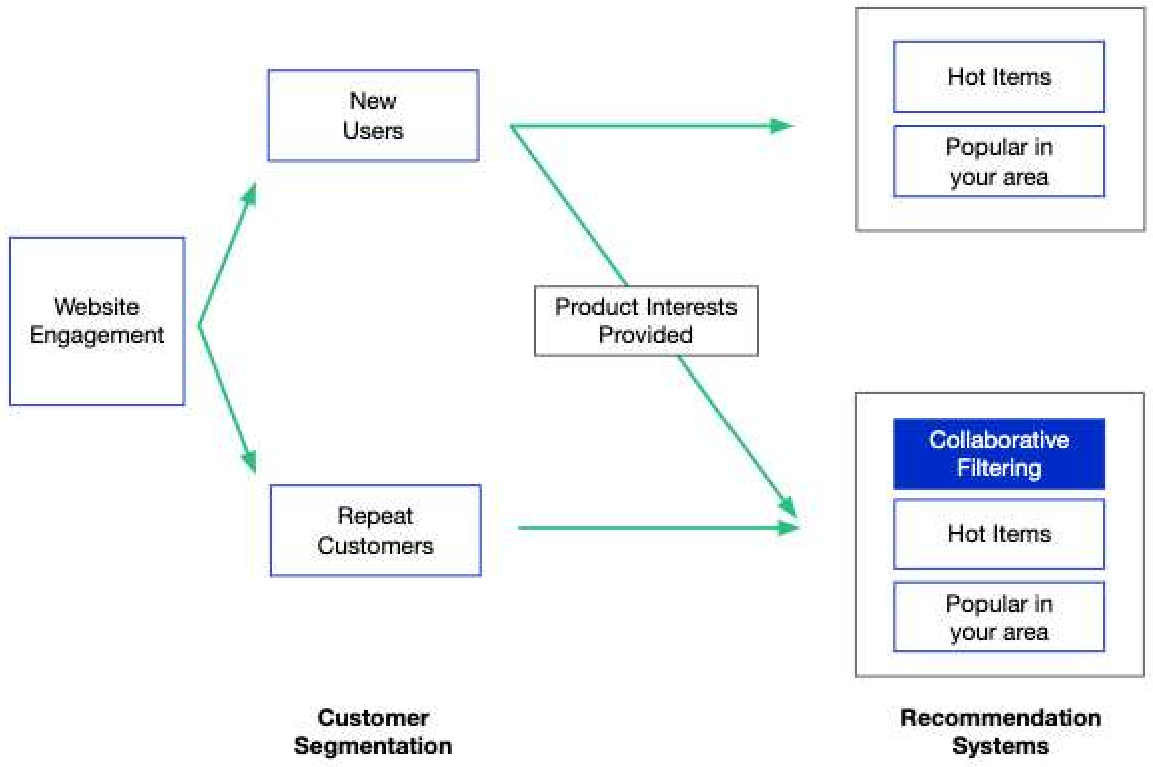
A Simple Approach To Building a Recommendation System

Leveraging the Surprise package to build a collaborative filtering recommender in Python

Molly Liebeskind Follow
Mar 26 · 6 min read ★



Example recommendation system with collaborative filtering. Image by Molly Liebeskind.

To understand the power of recommendation systems, it is easiest to focus on Netflix, whose state of the art recommendation system keeps us in front of our TVs for hours. However, recommenders are extremely diverse, playing a role in cross-selling products, identifying employee candidates who have similar skill sets, and finding customers who will respond to promotional messaging. And these examples only just scratch the surface of how recommendations systems can be used.

Although recommenders can be very complex, there are two simple approaches that act as a good starting point.

- Content based filtering: uses item features to recommend similar items to the ones that a user has previously liked or interacted with. Pandora's music genome project identifies musical attributes for each song and uses that information to find similar songs and make recommendations.
- Collaborative filtering: identifies items that a user will like based on how similar users rated each item. Netflix identifies shows and movies users will enjoy by determining which content similar users watched.

This post will focus on developing a collaborative filtering recommendation system using sales transaction data released by the Brazillian e-commerce company, Olist.

Getting Started

To build the recommender, we will use Surprise, a Python scikit package built for collaborative filtering.

The first step is to load in the packages we'll need and the dataset. The dataset is made up of 8 tables but for the purposes of this demonstration, I have already joined tables and isolated the columns we need. The full code is here.

```
#Import packages
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from surprise import NormalPredictor, Reader, Dataset, accuracy,
SVD, SVDpp, KNNBasic, CoClustering, SlopeOne

from surprise.model_selection import cross_validate, KFold,
GridSearchCV, train_test_split
```

```
#import dataset
olist_data = pd.read_csv('olist_data.csv')
```

As often happens with real-world data, this dataset is not perfectly constructed for creating a collaborative recommendation system. A big challenge here is that almost 90% of the users are first-time customers which means that we don't have previous ratings from them to identify products that like. Instead, we will separate the dataset into repeat and first time users and feed only the repeat customers into the collaborative filtering model. For first-time customers, we can still provide recommendations but they will be more generic, focused on item popularity and the user's location.

```
def repeat_and_first_time(data):

    repeaters = data.groupby('customer_unique_id').filter(lambda x:
len(x) > 1)
    first_timers = data.groupby('customer_unique_id').filter(lambda
x: len(x) == 1)

    return repeaters, first_timers
```

Using Surprise

In order to leverage surprise's built in user-ratings matrix conversion, we will need to supply a dataframe that contains a user id column, an item id column, and a rating column.

```
def create_user_ratings_df(data):
    df = data.groupby(['customer_unique_id','product_id'])
['review_score'].agg(['mean']).reset_index()

    df = df.rename({'mean':'estimator', 'product_id':'productId'},
axis=1)
    return df

user_ratings_df = create_user_ratings_df(repeater_data)

user_ratings_df.head()
```

Example user-ratings matrix. Image by Molly Liebeskind.

From here, Surprise will help us generate a user-ratings matrix where each user id is a row and each product the company offers is a column. This will have the same impact as creating a Pandas pivot table. We will divide that dataframe into a train and test set with an 80/20 partition.

```
def surprise_df(data):

    scale = (data.estimator.min(), data.estimator.max())
    reader = Reader(rating_scale=scale)

    df = Dataset.load_from_df(data[['customer_unique_id',
                                    'productId',
                                    'estimator']], reader)

    return df

user_ratings_matrix = surprise_df(user_ratings_df)

train_set, test_set = train_test_split(user_ratings_matrix,
test_size=0.2, random_state=19)
```

Surprise offers 11 different prediction algorithms including variations of KNN and dimensionality reduction techniques such as SVD and NMF. For this demonstration, we will test out a few of the most common techniques.

Using a 5-fold validation, we will compare the results of the following models.

- NormalPredictor: a baseline model that predicts a random rating based on the distribution of the training set, which is assumed to be normal.

- SVD: A matrix factorization technique popularized by Simon Funk as part of the Netflix prize.
- KNNBasic: leverages cosine similarity (or user-determined distance metric) to perform KNN.
- CoClustering: an algorithm that assigns points to clusters in a similar method to k-means.

There are two ways to assess model performance. Qualitatively, you can look at a given user and determine if the recommendation makes sense given other products they like. For example, if someone likes horror movies and doesn't like romantic comedies, *The Shining* would be a good recommendation relative to *Love Actually*. For this dataset, we do not have information about each product, only a product id so we will use a quantitative measure, root mean squared error. A combination of the two methods is ideal, though a quantitive measure is much more realistic in production.

```
kf = KFold(n_splits=5, shuffle=True, random_state=19)

def model_framework(train_data):

    #store the rmse values for each fold in the k-fold loop
    normp_rmse, svd_rmse, knn_rmse, co_rmse, slope_rmse = [],[],[],
[],[]

    for trainset, testset in kf.split(train_data):

        #baseline
        normp = NormalPredictor()
        normp.fit(trainset)
        normp_pred = normp.test(testset)
        normp_rmse.append(accuracy.rmse(normp_pred,verbose=False))

        #svd
        svd = SVD(n_factors=30, n_epochs=50,biased=True,
lr_all=0.005, reg_all=0.4, verbose=False)
        svd.fit(trainset)
        svd_pred = svd.test(testset)
        svd_rmse.append(accuracy.rmse(svd_pred,verbose=False))

        #knn
        knn = KNNBasic(k=40,sim_options={'name': 'cosine',
'user_based': False}, verbose=False)
        knn.fit(trainset)
        knn_pred = knn.test(testset)
        knn_rmse.append(accuracy.rmse(knn_pred,verbose=False))

        #co_clustering
        co = CoClustering(n_cltr_u=3,n_cltr_i=3,n_epochs=20)
        co.fit(trainset)
        co_pred = co.test(testset)
        co_rmse.append(accuracy.rmse(co_pred,verbose=False))


    mean_rmses = [np.mean(normp_rmse),
                  np.mean(svd_rmse),
                  np.mean(knn_rmse),
                  np.mean(co_rmse),
                  np.mean(slope_rmse)]

    model_names = ['baseline','svd','knn','coclustering','slopeone']
    compare_df = pd.DataFrame(mean_rmses, columns=['RMSE'],
index=model_names)

    return compare_df

comparison_df = model_framework(train_set)
comparison_df.head()
```

Image by Molly Liebeskind.

Based on the above, we determine that SVD has the lowest rmse and is the model we will move forward with tuning.

Model Tuning: Surprise's Grid Search

The surprise package offers an option to tune parameters using GridSearchCV. We will provide GridSearchCV with a dictionary of parameters and the rmse will be calculated and compared for every combination of the parameters.

```
def gridsearch(data, model, param_grid):
    param_grid = param_grid
    gs = GridSearchCV(model, param_grid, measures=['rmse'], cv=5)
    gs.fit(data)

    new_params = gs.best_params['rmse']
    best_score = gs.best_score['rmse']

    print("Best score:", best_score)
    print("Best params:", new_params)

    return new_params, best_score

svd_param_grid = {'n_factors': [25, 50,100],
                  'n_epochs': [20,30,50],
                  'lr_all': [0.002,0.005,0.01],
                  'reg_all':[0.02,0.1, 0.4]}

svd_params, svd_score = gridsearch(train_set, SVD, svd_param_grid)
```

From that search, we receive an output that tells us the best score (lowest rmse) obtained was 1.27 which was produced using the parameters {'n_factors': 25, 'n_epochs': 50, 'lr_all': 0.01, 'reg_all': 0.1}.

Final Model and Metrics

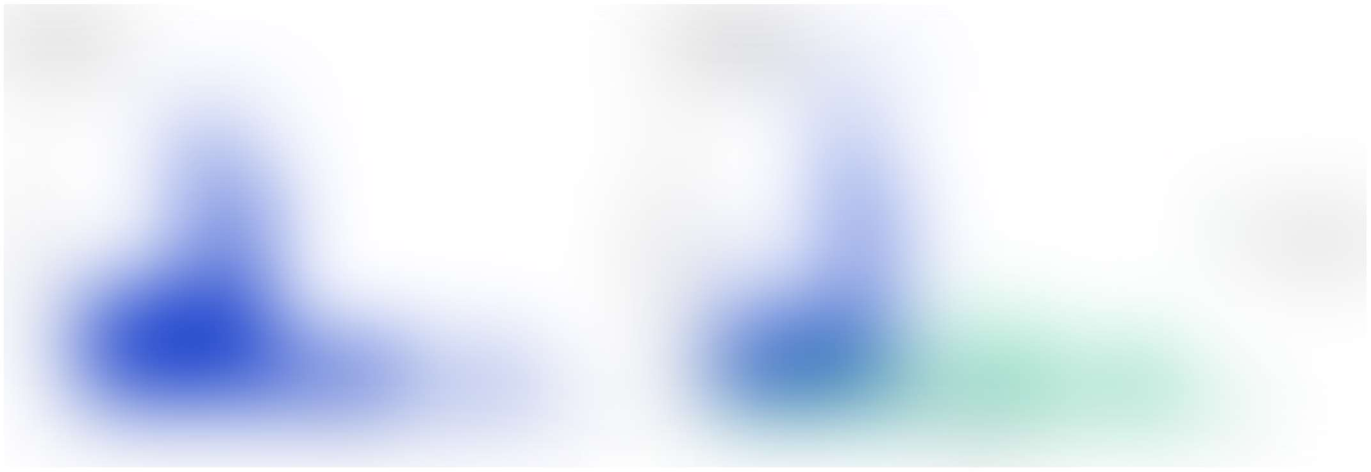
Leveraging the parameters above, we then run the model on the full train set without cross validation and obtain accuracy scores against the test set.

```
def final_model(train_set, test_set):
    params = {'n_factors': 10, 'n_epochs': 50, 'lr_all': 0.01,
              'reg_all': 0.1}

    svdpp = SVDpp(n_factors=params['n_factors'],
                  n_epochs=params['n_epochs'],
                  lr_all=params['lr_all'],
                  reg_all=params['reg_all'])
    svdpp.fit(train_set)

    predictions = svdpp.test(test_set)
    rmse = accuracy.rmse(predictions,verbose=False)

    return predictions, rmse

final_predictions, model_rmse = final_model(train_set, test_set)
```

The .test attribute outputs predictions that contain the user id, item id, the users actual rating, the models estimated rating, and indicator that the prediction was possible to make. In addition to looking at the model_rmse output from our final, trained model, we will look at the distribution of the absolute errors across all predictions. To do this, we will package the predictions output into a dataframe and add on a column that indicates error for each prediction. We will then visualize the results by plotting a histogram of the errors.

```
results = pd.DataFrame(final_predictions, columns=['userid',
'item_id', 'user_rating', 'model_est', 'details'])
results['err'] = abs(results.model_est - results.user_rating)
```

In the plots below, we see that although the error on the full dataset was 1.0, when a user gave a product a rating higher than 3, the model made much better predictions and the error was only 0.8. In contrast, when a user gave a product a rating of 3 or below, the error was significantly higher at 1.5. This is a good result because to provide good recommendations, we care much more about accurately predicting the products that a user likes and rates above 3.



Plots comparing overall model error with error for ratings > 3 and ratings ≤3. Image by Molly Liebeskind.

The model can then be packaged using the dump module, which is a wrapper around Pickle. For this example, the collaborative filtering recommendation is just a part of the overall system. It pairs with a recommendation based on overall top selling products and top performers in each geographic region.

Additional Notes

To become familiar with the process, Surprise has some great built-in datasets and the documentation thoroughly explains different cross-validation approaches, similarity metrics, and prediction algorithms.

At the moment, Surprise is not capable of handling implicit ratings, performing content filtering, or generating a hybrid recommender. However, for a beginner, Surprise is a simple and straightforward package for collaborative filtering.

You can find all code from this example on github.

Data Science Recommendation System Collaborative Filtering Machine Learning Python