Open in app ↗ Resume Membership

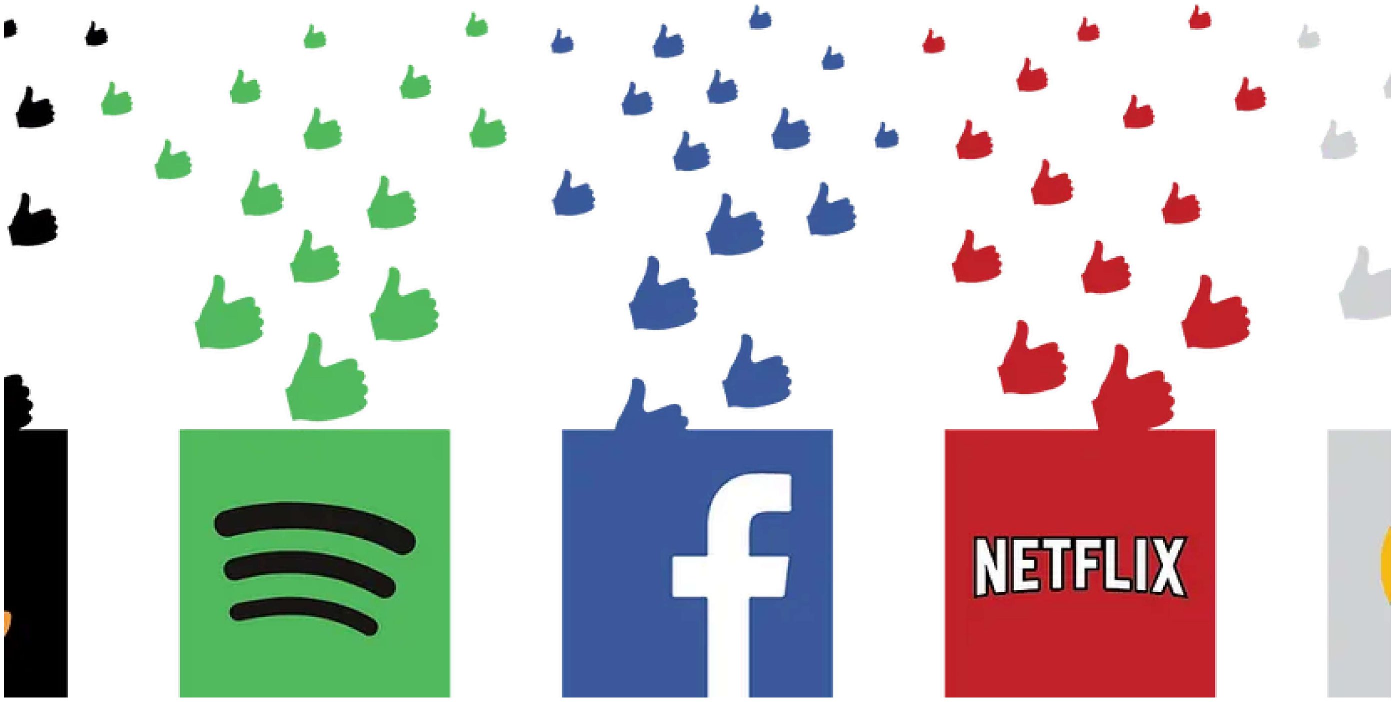
Search Medium

Adina Steinman Follow

Jan 29, 2021 · 5 min read · Listen

Save

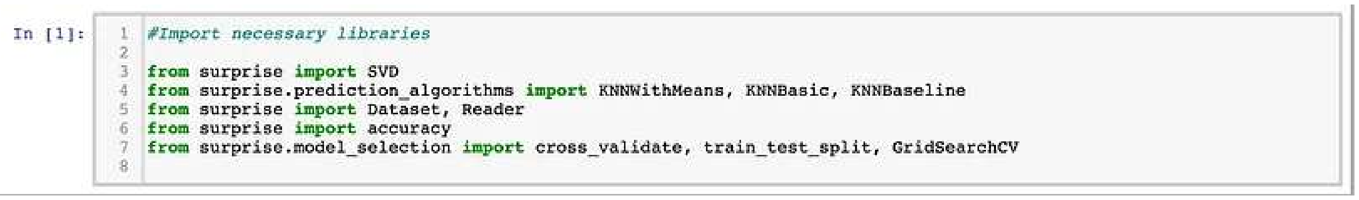
Building A Collaborative-Filtering Recommendation System in Surprise



Many companies use collaborative filtering recommendation systems. For example, Dick's Sporting Goods' advertises the message that "The more you shop, the better our recommendations are!" If more customers shop, then the company will have more data on users and items, and will be able to use this data to recommend other items or features that improve the customer's shopping experience.

Collaborative filtering relies on the fact that the company has *existing* data; otherwise, recommendation systems will s 32 1 t problem" and will need to find another method to generate data in order to provide recommendations. For the context of this post, we will assume that a company already has a sufficient amount of data in order to build a more sophisticated recommendation system, and as a result we will only be explaining the basics of collaborative filtering models. Now, let's get started!

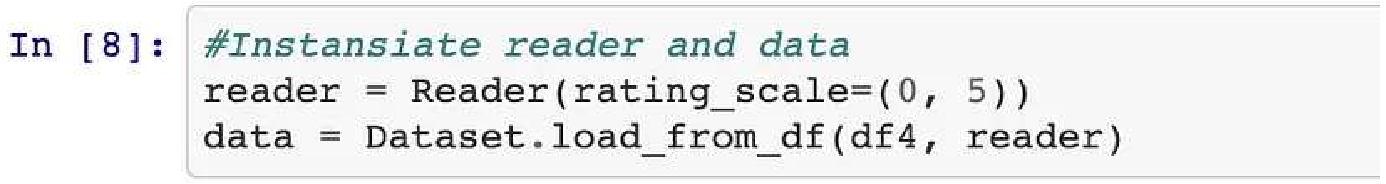
Surprise is a very valuable tool that can be used within Python to build recommendation systems. Its documentation is quite useful and explains its various prediction algorithms' packages. Before we start building a model, it is important to import elements of surprise that are useful for analysis, such as certain model types (SVD, KNNBasic, KNNBaseline, KNNWithMeans, and many more), Dataset and Reader objects (more on this later), accuracy scoring, and built in train-test-split, cross validation and GridSearch. See the import below:



```
In [1]:  #Import necessary libraries

         from surprise import SVD
         from surprise.prediction_algorithms import KNNWithMeans, KNNBasic, KNNBaseline
         from surprise import Dataset, Reader
         from surprise import accuracy
         from surprise.model_selection import cross_validate, train_test_split, GridSearchCV

```

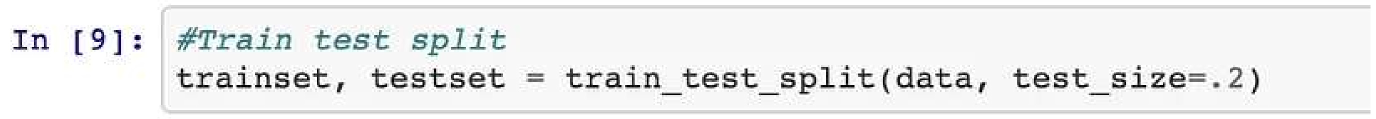
The next step is to instantiate the Reader and Dataset objects. This is where surprise functions differently than other typical scikit learn models. First, you need to instantiate a "reader" object and indicate the rating scale of your ratings data. For example, if you are rating a restaurant on a scale of 1–5, you would include this within your Reader object.

Then, your Dataset object needs to load your data and pass through your reader object, using the "load_from_df" tool. See code below:



```
In [8]: #Instansiate reader and data
        reader = Reader(rating_scale=(0, 5))
        data = Dataset.load_from_df(df4, reader)
```

Great! The data is now ready for the next step of the process; train test split. Surprise has its own built in TTS functionality so you can use that, as shown below:



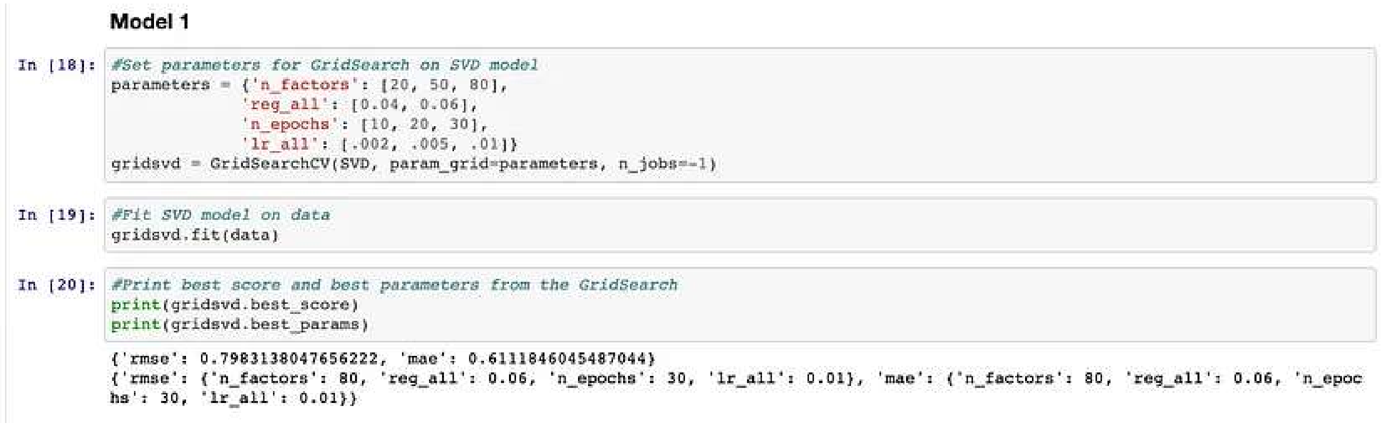
```
In [9]: #Train test split
        trainset, testset = train_test_split(data, test_size=.2)
```

Do note that the datatypes of the trainset and testset differ than the usual scikitlearn types. Not to worry though, as the functions within surprise are built to handle these types.

1. SVD Model

One of the most powerful recommendation systems is the SVD model. SVD is a form of matrix factorization that uses gradient descent to create predictions for a users' ratings, while minimizing the error between the predicted ratings and the actual ratings from our original utility matrix. As a result, gradient descent minimizes RMSE when predicting these new ratings.

The process for SVD is quite similar to other ML models; instantiate the model, fit on the train set, then predict on the test set. You can also incorporate GridSearch to SVD models in order to tune them further. There are several parameters worth noting: number of factors, number of iterations to run (known as "number of epochs), the learning rate, and the regularization term. Below we have chosen several values for each of these parameters and have fit it on a GridSearch model in order to find the best score and the best parameters for our data:



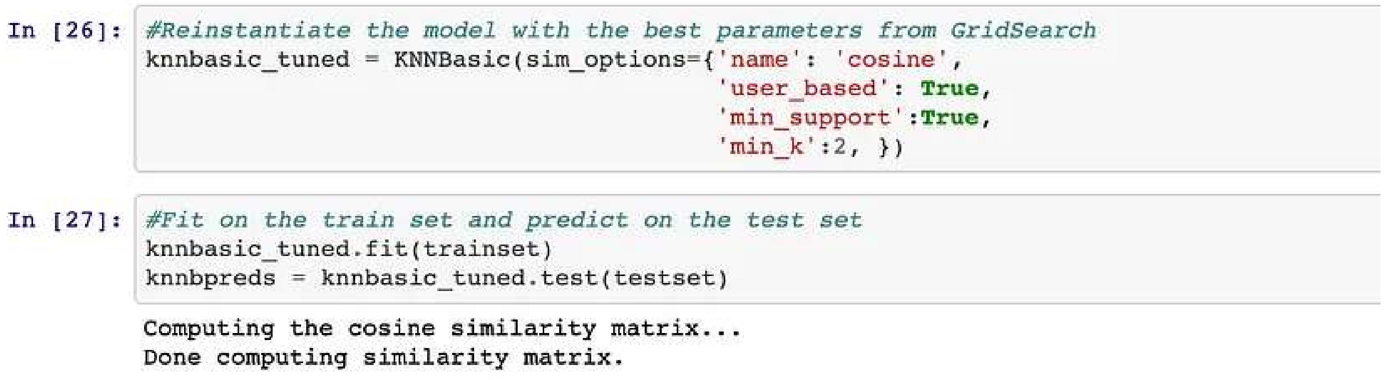
Model 1

```
In [18]: #Set parameters for GridSearch on SVD model
         parameters = {'n_factors': [20, 50, 80],
                       'reg_all': [0.04, 0.06],
                       'n_epochs': [10, 20, 30],
                       'lr_all': [.002, .005, .01]}
         gridsvd = GridSearchCV(SVD, param_grid=parameters, n_jobs=-1)

In [19]: #Fit SVD model on data
         gridsvd.fit(data)

In [20]: #Print best score and best parameters from the GridSearch
         print(gridsvd.best_score)
         print(gridsvd.best_params)

         {'rmse': 0.7983138047656222, 'mae': 0.6111846045487044}
         {'rmse': {'n_factors': 80, 'reg_all': 0.06, 'n_epochs': 30, 'lr_all': 0.01}, 'mae': {'n_factors': 80, 'reg_all': 0.06, 'n_epoc
         hs': 30, 'lr_all': 0.01}}
```

Our result shows us that the RMSE for this model is approximately 0.798 and the best parameters are n_factors=80, reg_all=0.06, n_epochs=30, and lr_all =0.01. This combination of hyperparameters has given us the lowest RMSE possible, given these parameter values. SVD is a very powerful tool that can be expanded further — if you are interested, check back at the documentation.

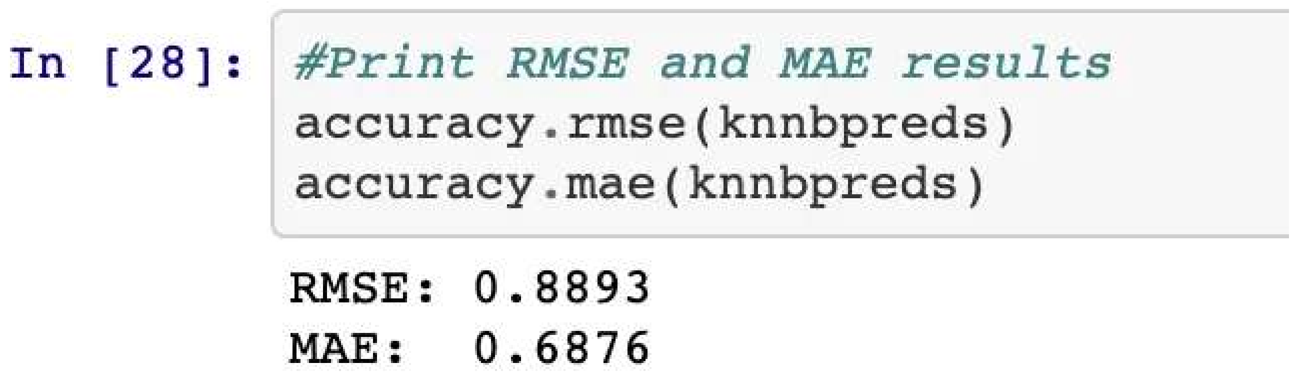
2. KNN Models

Another common model to use is “KNN”. These algorithms look at the nearest neighbors to determine which movie to predict. KNN takes in different hyperparameters than SVD: most notably, the similarity measure (example: cosine vs. pearson), whether it is “user based” or not, with or without “min support”, and the minimum number of neighbors to take into account (denoted by “min k”). Instead of showing GridSearch, below I’ll show how a KNNBasic model can be tuned by including hyperparameters, then fit the model on the train set and predict on the test set.



```
In [26]: #Reinstantiate the model with the best parameters from GridSearch
         knnbasic_tuned = KNNBasic(sim_options={'name': 'cosine',
                                               'user_based': True,
                                               'min_support':True,
                                               'min_k':2, })

In [27]: #Fit on the train set and predict on the test set
         knnbasic_tuned.fit(trainset)
         knnbpreds = knnbasic_tuned.test(testset)

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

Finally, we can display the RMSE and MAE results as follows:



```
In [28]: #Print RMSE and MAE results
         accuracy.rmse(knnbpreds)
         accuracy.mae(knnbpreds)

         RMSE: 0.8893
         MAE:  0.6876
```

In this case, the RMSE is 0.889, meaning that our model's predictions miss the actual ratings by approximately 0.8893 points. On a rating scale of 0–5, which was used for this model, this result is not too bad (a rating of 3.0 vs. 3.8 doesn't drastically change whether or not we can assume that someone liked a movie or not).

Conclusion

This is only a very brief introduction into SVD and KNN models for Recommendation Systems. There are other models that can be explored as well in Recommendation Systems. Additionally, more sophisticated models can be built with packages such as LightFM, and Recommendation Systems can be combined with both Natural Language Processing and Neutral Networks for more advanced Recommendation Systems.

This post has only scratched the surface of the world of Recommendation Systems. I encourage you to explore this topic of data science further, as it is very relevant today, and constantly evolving!

Recommendation System   Surprise   Collaborative Filtering