# Implementing your own recommender systems in Python



## Implementing your own recommender systems in Python

Nowadays, *recommender systems* are used to personalize your experience on the web, telling you what to buy, where to eat or even who you should be friends with. People's tastes vary, but generally follow patterns. People tend to like things that are similar to other things they like, and they tend to have similar taste as other people they are close with. Recommender systems try to capture these patterns to help predict what else you might like. E-commerce, social media, video and online news platforms have been actively deploying their own recommender systems to help their customers to choose products more efficiently, which serves win-win strategy.

Two most ubiquitous types of recommender systems are *Content-Based* and *Collaborative Filtering (CF)*. Collaborative filtering produces recommendations based on the knowledge of users' attitude to items, that is it uses the "wisdom of the crowd" to recommend items. In contrast, content-based recommender systems focus on the attributes of the items and give you recommendations based on the similarity between them.

In general, Collaborative filtering (CF) is the workhorse of recommender engines. The algorithm has a very interesting property of being able to do feature learning on its own, which means that it can start to learn for itself what features to use. CF can be divided into *Memory-Based Collaborative Filtering* and *Model-Based Collaborative filtering*. In this tutorial, you will implement Model-Based CF by using singular value decomposition (SVD) and Memory-Based CF by computing cosine similarity.

You will use MovieLens dataset, which is one of the most common datasets used when implementing and testing recommender engines. It contains 100k movie ratings from 943 users and a selection of 1682 movies. You should add unzipped movielens dataset folder to your notebook directory. You can download the dataset here (http://files.grouplens.org/datasets/movielens/ml-100k.zip).

```
import numpy as np
import pandas as pd
```

You read in the `u.data` file, which contains the full dataset. You can read a brief description of the dataset here (http://files.grouplens.org/datasets/movielens/ml-100k-README.txt).

```
header = ['user_id', 'item_id', 'rating', 'timestamp']
df = pd.read_csv('ml-100k/u.data', sep='\t', names=header)
```

Get a sneak peek of the first two rows in the dataset. Next, let's count the number of unique users and movies.

```
n_users = df.user_id.unique().shape[0]
n_items = df.item_id.unique().shape[0]
print 'Number of users = ' + str(n_users) + ' | Number of movies = ' + str(n_items)
```

```
Number of users = 943 | Number of movies = 1682
```

You can use the `scikit-learn` (http://scikit-learn.org/stable/) library to split the dataset into testing and training. `Cross_validation.train_test_split` (http://scikit-learn.org/stable/modules/generated/sklearn.cross_validation.train_test_split.html) shuffles and splits the data into two datasets according to the percentage of test examples (`test_size`), which in this case is 0.25.

```
from sklearn import cross_validation as cv
train_data, test_data = cv.train_test_split(df, test_size=0.25)
```

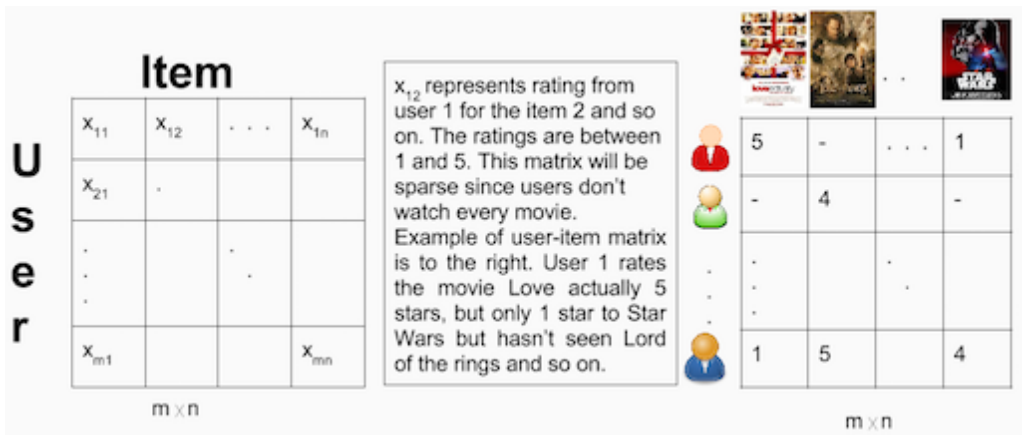
## Memory-Based Collaborative Filtering

Memory-Based Collaborative Filtering approaches can be divided into two main sections: *user-item filtering* and *item-item filtering*. A *user-item filtering* takes a particular user, find users that are similar to that user based on similarity of ratings, and recommend items that those similar users liked. In contrast, *item-item filtering* will take an item, find users who liked that item, and find other items that those users or similar users also liked. It takes items and outputs other items as recommendations.

- *Item-Item Collaborative Filtering*: “Users who liked this item also liked …”
- *User-Item Collaborative Filtering*: “Users who are similar to you also liked …”

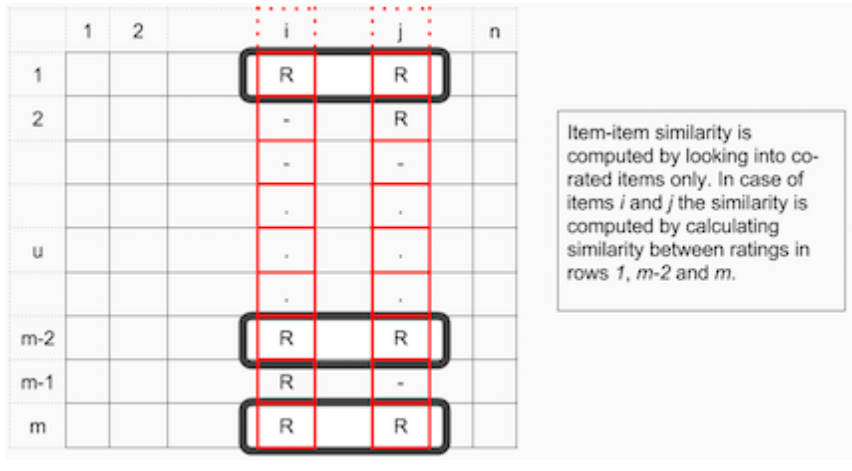
In both cases, you create a user-item matrix which you build from the entire dataset. Since you have split the data into testing and training you will need to create two 943 × 1682 matrices. The training matrix contains 75% of the ratings and the testing matrix contains 25% of the ratings.

Example of user-item matrix:

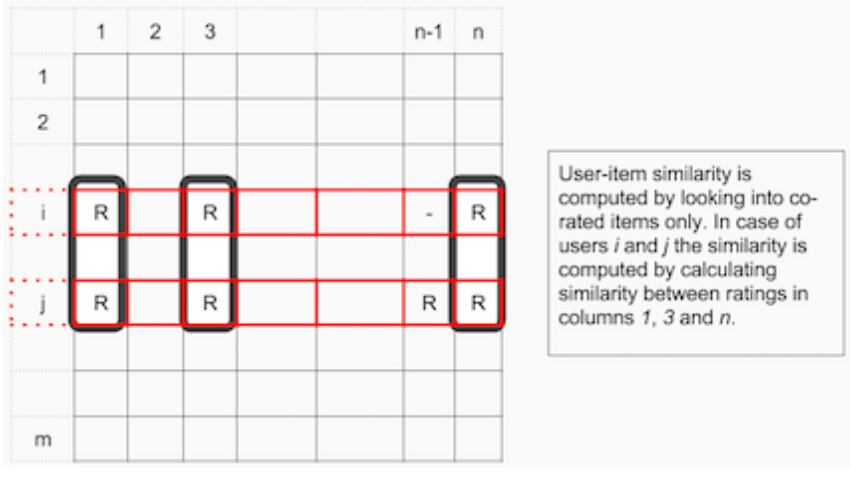



After you have built the user-item matrix you calculate the similarity and create a similarity matrix.

The similarity values between items in *Item-Item Collaborative Filtering* are measured by observing all the users who have rated both items.



For *User-Item Collaborative Filtering* the similarity values between users are measured by observing all the items that are rated by both users.




A distance metric commonly used in recommender systems is *cosine similarity*, where the ratings are seen as vectors in $n$-dimensional space and the similarity is calculated based on the angle between these vectors. Cosine similiarity for users $a$ and $m$ can be calculated using the formula below, where you take dot product of the user vector $u_k$ and the user vector $u_a$ and divide it by multiplication of the Euclidean lengths of the vectors.

$$s_u^{cos}(u_k, u_a) = \frac{u_k \cdot u_a}{\|u_k\| \, \|u_a\|} = \frac{\sum x_{k,m} x_{a,m}}{\sqrt{\sum x_{k,m}^2 \sum x_{a,m}^2}}$$

To calculate similarity between items $m$ and $b$ you use the formula:

$$s_u^{cos}(i_m, i_b) = \frac{i_m \cdot i_b}{\|i_m\| \, \|i_b\|} = \frac{\sum x_{a,m} x_{a,b}}{\sqrt{\sum x_{a,m}^2 \sum x_{a,b}^2}}$$

Your first step will be to create the user-item matrix. Since you have both testing and training data you need to create two matrices.

```
#Create two user-item matrices, one for training and another for testing
train_data_matrix = np.zeros((n_users, n_items))
for line in train_data.itertuples():
    train_data_matrix[line[1]-1, line[2]-1] = line[3]

test_data_matrix = np.zeros((n_users, n_items))
for line in test_data.itertuples():
    test_data_matrix[line[1]-1, line[2]-1] = line[3]
```

You can use the `pairwise distances` (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.pairwise_distances.html) function from `sklearn` to calculate the cosine similarity. Note, the output will range from 0 to 1 since the ratings are all positive.

```
from sklearn.metrics.pairwise import pairwise_distances
user_similarity = pairwise_distances(train_data_matrix, metric='cosine')
item_similarity = pairwise_distances(train_data_matrix.T, metric='cosine')
```

Next step is to make predictions. You have already created similarity matrices: `user_similarity` and `item_similarity` and therefore you can make a prediction by applying following formula for user-based CF:

$$\hat{x}_{k,m} = \bar{x}_k + \frac{\sum\limits_{u_a} sim_u(u_k, u_a)(x_{a,m} - \bar{x}_{u_a})}{\sum\limits_{u_a} |sim_u(u_k, u_a)|}$$

You can look at the similarity between users $k$ and $a$ as weights that are multiplied by the ratings of a

similar user $a$ (corrected for the average rating of that user). You will need to normalize it so that the ratings stay between 1 and 5 and, as a final step, sum the average ratings for the user that you are trying to predict.

The idea here is that some users may tend always to give high or low ratings to all movies. The relative difference in the ratings that these users give is more important than the absolute values. To give an example: suppose, user $k$ gives 4 stars to his favourite movies and 3 stars to all other good movies. Suppose now that another user $t$ rates movies that he/she likes with 5 stars, and the movies he/she fell asleep over with 3 stars. These two users could have a very similar taste but treat the rating system differently.

When making a prediction for item-based CF you don't need to correct for users average rating since query user itself is used to do predictions.

$$\hat{x}_{k,m} = \frac{\sum_{i_b} sim_i(i_m, i_b)(x_{k,b})}{\sum_{i_b} |sim_i(i_m, i_b)|}$$

```
def predict(ratings, similarity, type='user'):
    if type == 'user':
        mean_user_rating = ratings.mean(axis=1)
        #You use np.newaxis so that mean_user_rating has same format as ratings
        ratings_diff = (ratings - mean_user_rating[:, np.newaxis])
        pred = mean_user_rating[:, np.newaxis] + similarity.dot(ratings_diff) /
np.array([np.abs(similarity).sum(axis=1)]).T
    elif type == 'item':
        pred = ratings.dot(similarity) / np.array([np.abs(similarity).sum(axis=1)])
    return pred
```

```
item_prediction = predict(train_data_matrix, item_similarity, type='item')
user_prediction = predict(train_data_matrix, user_similarity, type='user')
```

## Evaluation

There are many evaluation metrics but one of the most popular metric used to evaluate accuracy of predicted ratings is *Root Mean Squared Error (RMSE)*.

$$RMSE = \sqrt{\frac{1}{N}\sum(x_i - \hat{x}_i)^2}$$

You can use the `mean_square_error` (http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html) (MSE) function from `sklearn`, where the RMSE is just the square root of MSE. To read more about different evaluation metrics you can take a look at this article (http://research.microsoft.com/pubs/115396/EvaluationMetrics.TR.pdf).

Since you only want to consider predicted ratings that are in the test dataset, you filter out all other elements in the prediction matrix with `prediction[ground_truth.nonzero()]`.

```
from sklearn.metrics import mean_squared_error
from math import sqrt
def rmse(prediction, ground_truth):
    prediction = prediction[ground_truth.nonzero()].flatten()
    ground_truth = ground_truth[ground_truth.nonzero()].flatten()
    return sqrt(mean_squared_error(prediction, ground_truth))
```

```
print 'User-based CF RMSE: ' + str(rmse(user_prediction, test_data_matrix))
print 'Item-based CF RMSE: ' + str(rmse(item_prediction, test_data_matrix))
```

```
User-based CF RMSE: 3.1236202241
Item-based CF RMSE: 3.44983070639
```

Memory-based algorithms are easy to implement and produce reasonable prediction quality. The drawback of memory-based CF is that it doesn't scale to real-world scenarios and doesn't address the well-known cold-start problem, that is when new user or new item enters the system. Model-based CF methods are scalable and can deal with higher sparsity level than memory-based models, but also suffer when new users or items that don't have any ratings enter the system. I would like to thank Ethan Rosenthal for his post (http://blog.ethanrosenthal.com/2015/11/02/intro-to-collaborative-filtering/) about Memory-Based Collaborative Filtering.

## Model-based Collaborative Filtering

Model-based Collaborative Filtering is based on *matrix factorization (MF)* which has received greater exposure, mainly as an unsupervised learning method for latent variable decomposition and dimensionality reduction. Matrix factorization is widely used for recommender systems where it can deal better with scalability and sparsity than Memory-based CF. The goal of MF is to learn the latent preferences of users and the latent attributes of items from known ratings (learn features that describe the characteristics of ratings) to then predict the unknown ratings through the dot product of the latent features of users and items. When you have a very sparse matrix, with a lot of dimensions, by doing matrix factorization you can restructure the user-item matrix into low-rank structure, and you can represent the matrix by the multiplication of two low-rank matrices, where the rows contain the latent vector. You fit this matrix to approximate your original matrix, as closely as possible, by multiplying the low-rank matrices together, which fills in the entries missing in the original matrix.

Let's calculate the sparsity level of MovieLens dataset:

```
sparsity=round(1.0-len(df)/float(n_users*n_items),3)
print 'The sparsity level of MovieLens100K is ' +  str(sparsity*100) + '%'
```

```
The sparsity level of MovieLens100K is 93.7%
```

To give an example of the learned latent preferences of the users and items: let's say for the MovieLens dataset you have the following information: *(user id, age, location, gender, movie id, director, actor, language, year, rating)*. By applying matrix factorization the model learns that important user features are *age group (under 10, 10-18, 18-30, 30-90)*, *location* and *gender*, and for movie features it learns that *decade*, *director* and *actor* are most important. Now if you look into the information you have stored, there is no such feature as the *decade*, but the model can learn on its own. The important aspect is that the CF model only uses data (user_id, movie_id, rating) to learn the latent features. If there is little data available model-based CF model will predict poorly, since it will be more difficult to learn the latent features.

Models that use both ratings and content features are called *Hybrid Recommender Systems* where both Collaborative Filtering and Content-based Models are combined. Hybrid recommender systems usually show higher accuracy than Collaborative Filtering or Content-based Models on their own: they are capable to address the cold-start problem better since if you don't have any ratings for a user or an item you could use the metadata from the user or item to make a prediction. Hybrid recommender systems will be covered in the next tutorials.

### SVD

A well-known matrix factorization method is *Singular value decomposition (SVD)*. Collaborative Filtering can be formulated by approximating a matrix $\boldsymbol{X}$ by using singular value decomposition. The

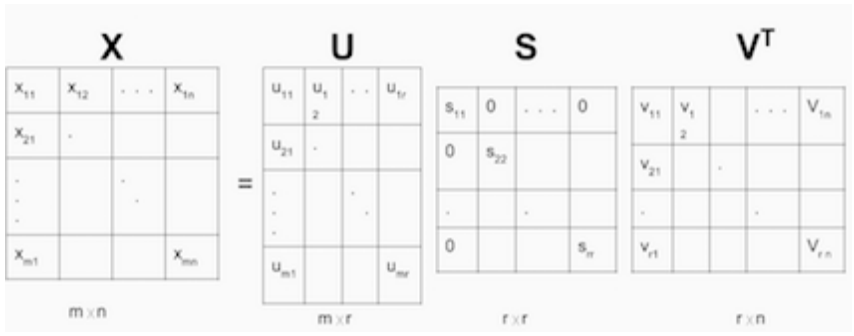
winning team at the Netflix Prize competition used SVD matrix factorization models to produce product recommendations, for more information I recommend to read articles: Netflix Recommendations: Beyond the 5 stars (http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html) and Netflix Prize and SVD (http://buzzard.ups.edu/courses/2014spring/420projects/math420-UPS-spring-2014-gower-netflix-SVD.pdf). The general equation can be expressed as follows: $X = U \times S \times V^T$

Given an $m \times n$ matrix $X$:

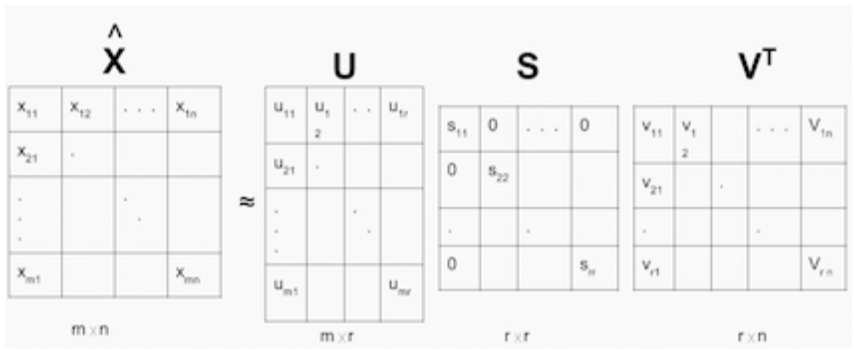
- $U$ is an $m \times r$ orthogonal matrix
- $S$ is an $r \times r$ diagonal matrix with non-negative real numbers on the diagonal
- $V^T$ is an $r \times n$ orthogonal matrix

Elements on the diagnoal in $S$ are known as *singular values of* $X$.

Matrix $X$ can be factorized to $U$, $S$ and $V$. The $U$ matrix represents the feature vectors corresponding to the users in the hidden feature space and the $V$ matrix represents the feature vectors corresponding to the items in the hidden feature space.




Now you can make a prediction by taking dot product of $U$, $S$ and $V^T$.




```
import scipy.sparse as sp
from scipy.sparse.linalg import svds
```

```
#get SVD components from train matrix. Choose k.
u, s, vt = svds(train_data_matrix, k = 20)
s_diag_matrix=np.diag(s)
X_pred = np.dot(np.dot(u, s_diag_matrix), vt)
print 'User-based CF MSE: ' + str(rmse(X_pred, test_data_matrix))
```

```
User-based CF MSE: 2.71348045599
```

Carelessly addressing only the relatively few known entries is highly prone to overfitting. SVD can be very slow and computationally expensive. More recent work minimizes the squared error by applying alternating least square or stochastic gradient descent and uses regularization terms to prevent overfitting.

To wrap it up:

- In this post we have covered how to implement simple *Collaborative Filtering* methods, both memory-based CF and model-based CF.
- *Memory-based models* are based on similarity between items or users, where we use cosine-

similarity.

- *Model-based CF* is based on matrix factorization where we use SVD to factorize the matrix.
- Building recommender systems that perform well in cold-start scenarios (where litle data is availabe on new users and items) remains a challenge. The standard collaborative filtering method performs poorly is such settings. In the next tutorials you will have the change to dig deeper into this problem.

---

## ABOUT THE AUTHOR

Agnes Johannsdottir

Agnes is a master student in Business Analytics at University College London. She studied Management Engineering in Iceland and worked for 2 years as an IT consultant in supply chain. Her main interests lie in using data science methods (especially machine learning) to apply in Retail and Supply Chain businesses.

---