Matrix Factorization

SUS-Labs - Eyad Kannout

Matrix Factorization:

Matrix factorization is a simple embedding model. Given the feedback matrix $A \in R^{m \times n}$, where m is the number of users (or queries) and n is the number of items, the model learns:

1. A user embedding matrix $U \in \mathbb{R}^{m \times d}$, where row i is the embedding for user i
2. An item embedding matrix $V \in \mathbb{R}^{n \times d}$, where row j is the embedding for item j

The embeddings are learned such that the product UV^T is a good approximation of the feedback matrix A. Observe that the (i, j) entry of $U.V^T$ is simply the dot product $\langle U_i, V_i \rangle$ of the embeddings of user i and items j, which you want to be close to $A_{i,j}$

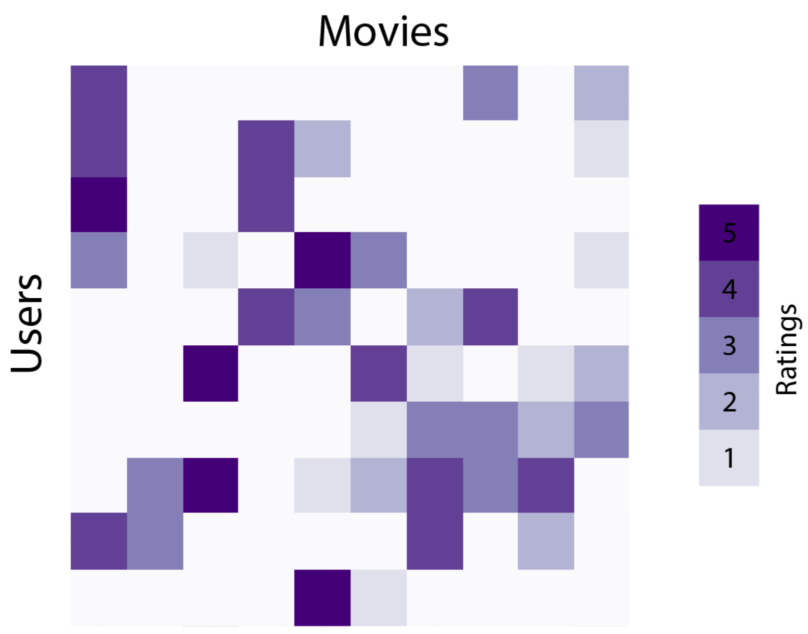
The main idea in matrix factorization is to break down the main large complicated matrix into product of small matrices. In order to break down the main matrix we need find the dependencies between rows and columns in that matrix in order to predict the missing values (usually the ratings) in this matrix.

In order to find this factorization we need to figure out:

1. What are the features (factors)?
2. What are the features (factors) users like?
3. What are the scores that movies gives to features (factors)?

For all previous findings we need machine learning

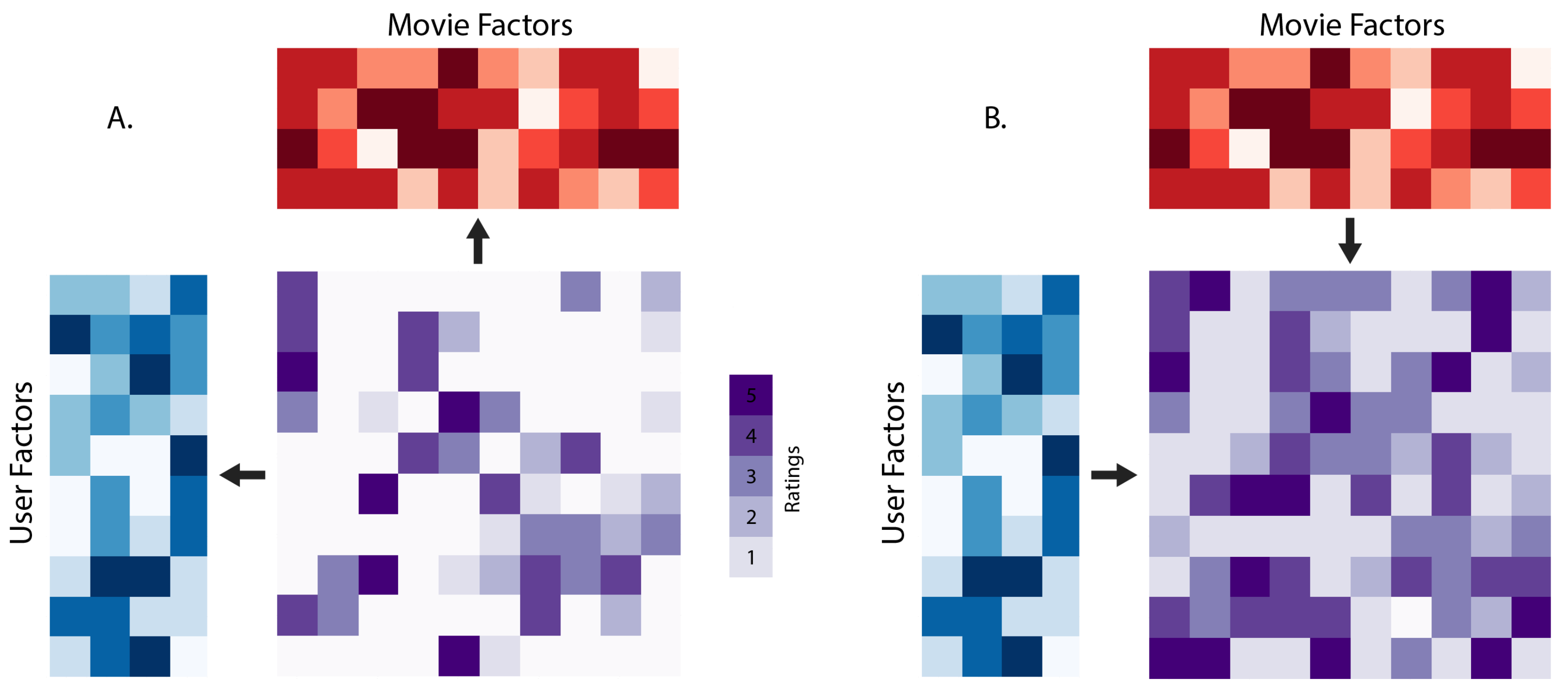
The Netflix Prize (https://en.wikipedia.org/wiki/Netflix_Prize) is likely the most famous example many data scientists explored in detail for matrix factorization. Simply put, the challenge was to predict a viewer's rating of shows that they hadn't yet watched based on their ratings of shows that they had watched in the past, as well as the ratings that other viewers had given a show. This takes the form of a matrix where the rows are users, the columns are shows, and the values in the matrix correspond to the rating of the show. Most of the values in this matrix are missing, as users have not rated (or seen) the majority of shows. Below is an illustrative example of what this data tends to look like, where ratings are shown in purple and missing values are denoted in white.



Matrix factorization is a simple idea that tries to learn connections between the known values in order to impute the missing values in a smart fashion. Simply put, for each row and for each column it learns k numeric "factors" that represent the row or column. In the example of the Netflix challenge, the factors for movies could correspond to "is this a comedy?" or "is a big name actor/actress in this movie?" The factors for users might correspond to "does this user like comedies?" and "does this user like this actor/actress?" Ratings are then determined by a dot product between these two smaller matrices (the user factors and the movie factors) such that an individual prediction for how user i will rate movie j is calculated by

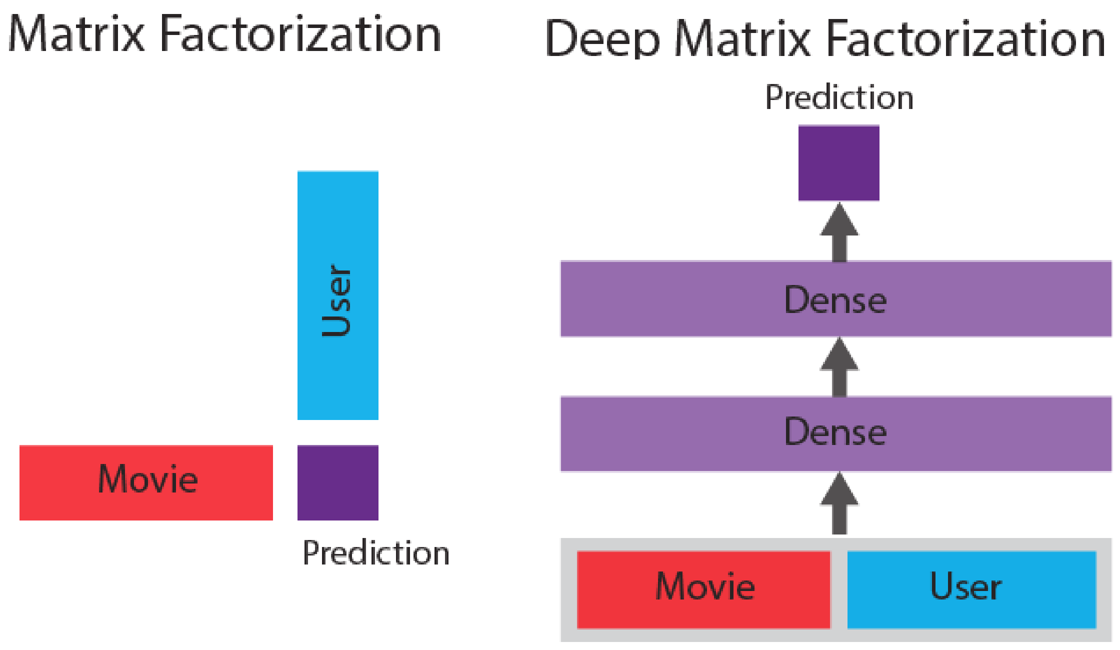
$$rating_{i,j} = \sum_{l=1}^{k} user_{i,l} \cdot movie_{j,l}$$

It is impractical to pre-assign meaning to these factors at large scale; rather, we need to learn the values directly from data. Once these factors are learned using the observational data, a simple dot product between the matrices can allow one to fill in all of the missing values. If two movies are similar to each other, they will have likely have similar factors, and their corresponding matrix columns will have similar values. The upside to this approach is that an informative set of factors can be learned automatically from data, but the downside is that the meaning of these factors might not be easily interpretable. This may or may not be an issue for your application. Below shows the two steps in this approach. On the left is the training step, where the factors are learned using the observed data, and on the right is the prediction step, where the missing values are imputed using the dot product between the two factor matrices.



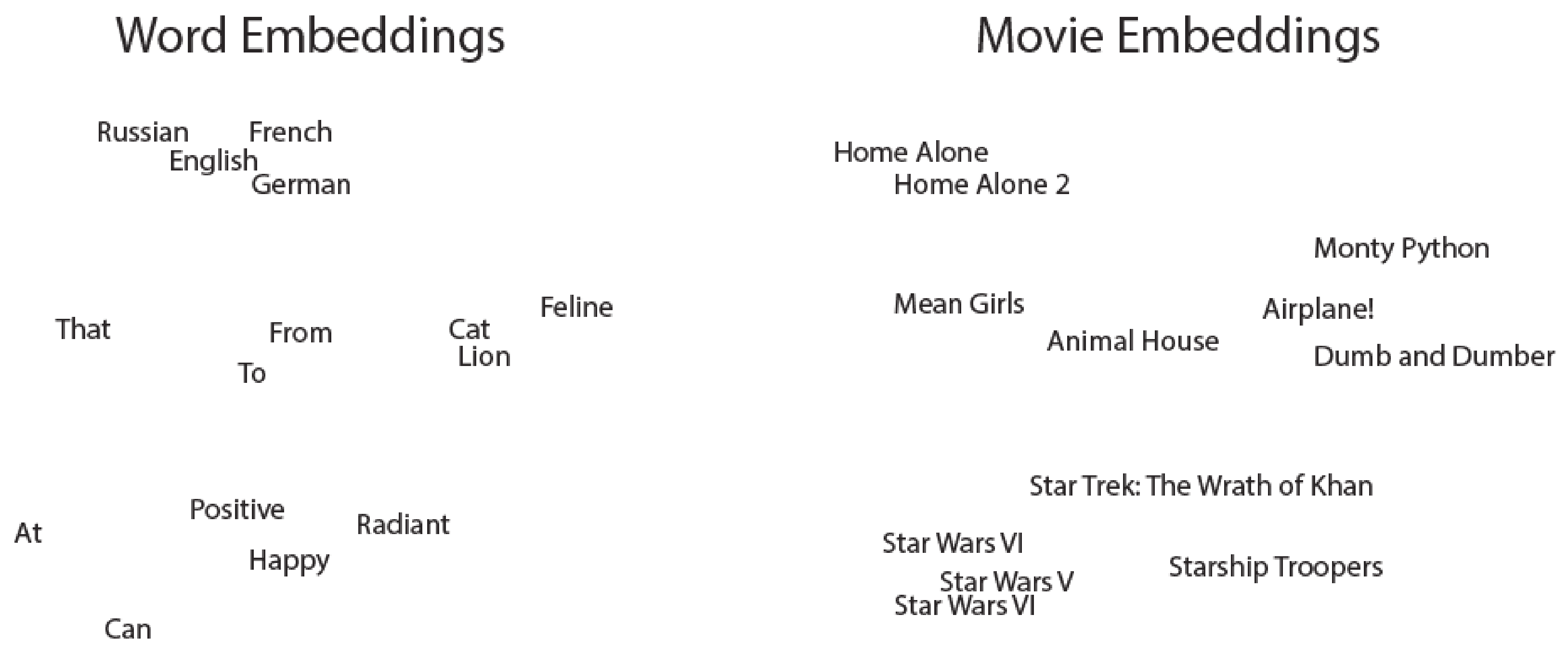
Matrix factorization is a linear method, meaning that if there are complicated non-linear interactions going on in the dataset, a simple dot product may not be able to handle it well. Given the recent success of deep learning in complicated non-linear computer vision and natural language processing tasks, it is natural to want to find a way to incorporate it into matrix factorization as well. A way to do this is called "deep matrix factorization" and involves the replacement of the dot product with a neural network that is trained jointly with the factors. This makes the model more powerful because a neural network can model important non-linear combinations of factors to make better predictions.

Below is a comparison of the two methods.



In traditional matrix factorization the prediction is the simple dot product between the factors for each of the dimensions. In contrast, in deep matrix factorization the factors for both are concatenated together and used as the input to a neural network whose output is the prediction. The parameters in the neural network are then trained jointly with the factors to produce a sophisticated non-linear model for matrix factorization.

A core component of the matrix factorization model will be the embedding layer. This layer takes in an integer and outputs a dense array of learned features. This is widely used in natural language processing, where the input would be words, and the output might be the features related to that words' sentiment. The strength of this layer is the ability for the network to learn what useful features are instead of needing them to be predefined. Below is an illustrative example of the output of a two dimensional embedding layer. We can see that on the left words that mean similar things cluster together, such as languages towards the top and positive words near the bottom. One can do a similar thing for movies (on the right) and observe that comedies seem to cluster towards the center and sci-fi clusters closer to the bottom. This is useful for our model because if we learn that a user likes one comedy, we can infer that they will have a higher ranking for another comedy based on their co-localization in this embedding space.



Movie Recommendations Using Matrix Factorization

Recommendation Systems are one of the most popular applications that use Matrix Factorization in order to predict useful recommendations for the users. In this lecture, we will focus on model-based collaborative filtering algorithm that utilizes Matrix Factorization to predict a users ratings for movies based on only a dataset users and their ratings of movies.

The original user-movie matrix is very sparse as it contains mostly 0s of unknown ratings. MF turns this sparse matrix into low-rank structure by compressing the sparse information into a k-dimenional space, where k represents the number of latent factors. It creates a smaller U matrix ("row factor") and smaller V matrix ("column factor"). Multiplying these creates the approximation of the orginial, sparse matrix.

In next experiment, we will use the ratings file of the [1M MovieLens (http://files.grouplens.org/datasets/movielens/ml-1m.zip)](http://files.grouplens.org/datasets/movielens/ml-1m.zip) dataset to produce recommendations for users based on previous ratings.

Example1: Matrix Factorization via Singular Value Decomposition

Matrix factorization is the breaking down of one matrix in a product of multiple matrices. There are many different ways to factor matrices, but singular value decomposition is particularly useful for making recommendations.

So what is singular value decomposition (SVD)?

At a high level, SVD is an algorithm that decomposes a matrix R into the best lower rank (i.e. smaller/simpler) approximation of the original matrix R. Mathematically, it decomposes R into a two unitary matrices and a diagonal matrix:

$$R = U\Sigma V^T$$

where R is users's ratings matrix, U is the user "features" matrix, Σ is the diagonal matrix of singular values (essentially weights), and V^T is the movie "features" matrix. U and V^T are orthogonal, and represent different things. U represents how much users "like" each feature and V^T represents how relevant each feature is to each movie.

To get the lower rank approximation, we take these matrices and keep only the top k features, which we think of as the underlying tastes and preferences vectors.

Setting Up the Ratings Data

In [2]:
```
import pandas as pd
import numpy as np
import os
import urllib.request
import zipfile

# If we don't have the data yet, download it from the appropriate source
if not os.path.exists('ml-1m.zip'):
    urllib.request.urlretrieve('http://files.grouplens.org/datasets/movielens/ml-1m.zip', 'ml-1m.zip')

# Now extract the data since we know we have it at this point
with zipfile.ZipFile("ml-1m.zip", "r") as f:
    f.extractall("./")

# Now load it up using a pandas dataframe
ratings = './ml-1m/ratings.dat'
headers = ['UserID', 'MovieID', 'Rating', 'Timestamp']
header_row = None
ratings_df = pd.read_csv(ratings,
                         sep='::',
                         names=headers,
                         header=header_row,
                         engine='python',
                         dtype={
                           'userID': np.int32,
                           'movieID': np.int32,
                           'rating': np.float32,
                           'timestamp': np.int32})
ratings_df.head()
```

Out[2]:

	UserID	MovieID	Rating	Timestamp
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291

In [3]:
```
movies = './ml-1m/movies.dat'
headers = ['MovieID', 'Title', 'Genres']
header_row = None
movies_df = pd.read_csv(movies,
                         sep='::',
                         names=headers,
                         header=header_row,
                         engine='python',
                         dtype={
                           'userID': np.int32,
                           'movieID': np.int32,
                           'rating': np.float32,
                           'timestamp': np.int32,
                         })
movies_df['MovieID'] = movies_df['MovieID'].apply(pd.to_numeric)
movies_df.head()
```

Out[3]:

	MovieID	Title	Genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy

We want the format of ratings matrix to be one row per user and one column per movie. we'll `pivot` `ratings_df` to get that and call the new variable `R`.

In [4]:
```python
R_df = ratings_df.pivot(index = 'UserID', columns ='MovieID', values = 'Rating').fillna(0)
R_df.head()
```

Out[4]:

MovieID	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**...**	**3943**	**3944**	**3945**	**3946**	**3947**	**3948**	**3949**	**3950**	**3951**	**3952**
UserID																					
1	5.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 3706 columns

The last thing is to de-mean the data (normalize by each users mean) and convert it from a dataframe to a numpy array.

In [5]:
```python
R = R_df.values
print(R)
user_ratings_mean = np.mean(R, axis = 1)
R_demeaned = R - user_ratings_mean.reshape(-1, 1)
print(R_demeaned)
```

```
[[5. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [3. 0. 0. ... 0. 0. 0.]]
[[ 4.94009714 -0.05990286 -0.05990286 ... -0.05990286 -0.05990286
  -0.05990286]
 [-0.12924987 -0.12924987 -0.12924987 ... -0.12924987 -0.12924987
  -0.12924987]
 [-0.05369671 -0.05369671 -0.05369671 ... -0.05369671 -0.05369671
  -0.05369671]
 ...
 [-0.02050729 -0.02050729 -0.02050729 ... -0.02050729 -0.02050729
  -0.02050729]
 [-0.1287102  -0.1287102  -0.1287102  ... -0.1287102  -0.1287102
  -0.1287102 ]
 [ 2.6708041  -0.3291959  -0.3291959  ... -0.3291959  -0.3291959
  -0.3291959 ]]
```

Singular Value Decomposition

Scipy and Numpy both have functions to do the singular value decomposition. I'm going to use the Scipy function `svds` because it let's me choose how many latent factors I want to use to approximate the original ratings matrix (instead of having to truncate it after).

In [6]:
```python
from scipy.sparse.linalg import svds
U, sigma, Vt = svds(R_demeaned, k = 50)
```

The function returns exactly what matrices mentioned before, except that the Σ returned is just the values instead of a diagonal matrix. Let's convert it to the diagonal matrix form.

In [7]:
```python
sigma = np.diag(sigma)
print(sigma)
```

```
[[ 147.18581225    0.            0.         ...    0.
     0.            0.        ]
 [   0.          147.62154312    0.         ...    0.
     0.            0.        ]
 [   0.            0.          148.58855276 ...    0.
     0.            0.        ]
 ...
 [   0.            0.            0.         ...  574.46932602
     0.            0.        ]
 [   0.            0.            0.         ...    0.
   670.41536276    0.        ]
 [   0.            0.            0.         ...    0.
     0.         1544.10679346]]
```

Making Predictions from the Decomposed Matrices

Now, we have everything we need to make movie ratings predictions for every user. I can do it all at once by following the math and matrix multiply U, Σ, and V^T back to get the rank $k = 50$ approximation of R.

We also need to add the user means back to get the actual star ratings prediction.

In [8]:
```python
all_user_predicted_ratings = np.dot(np.dot(U, sigma), Vt) + user_ratings_mean.reshape(-1, 1)
```

If I wanted to put this kind of system into production, I'd want to create a training and validation set and optimize the number of latent features (k) by minimizing the Root Mean Square Error. Intuitively, the Root Mean Square Error will decrease on the training set as k increases (because I'm approximating the original ratings matrix with a higher rank matrix).

However, for movies, between around 20 and 100 feature "preferences" vectors have been found to be optimal for generalizing to unseen data.

I could create a training and validation set and optimize k by minimizing RMSE, but since I'm just going through proof of concept I'll leave that for another post. I just want to see some movie recommendations.

Task1:

Write a function to recommend movies for any user. This functions should return the movies with the highest predicted rating that the specified user hasn't already rated?

for the sake of comparison: return the list of movies the user has already rated as well.

.....

Example2: Matrix Factorization via Singular Value Decomposition using Surprise Library

SVD is implemented in the Surprise (https://surprise.readthedocs.io/en/stable/) library as a recommender module.

- Detailed documentations of the SVD module in Surprise can be found here (https://surprise.readthedocs.io/en/stable/matrix_factorization.html#surprise.prediction_algorithms.matrix_factorization.SVD).
- Source codes of the SVD implementation is available on the Surprise Github repository, which can be found here (https://github.com/NicolasHug/Surprise/blob/master/surprise/prediction_algorithms/matrix_factorization.pyx).

Load data

In [9]:
```
from surprise import SVD
from surprise import Dataset
from surprise.model_selection import cross_validate
from surprise import accuracy
from surprise.model_selection import train_test_split
from surprise.model_selection import KFold
from surprise.model_selection import cross_validate


# Load the movielens-100k or movielens-1m dataset (download it if needed),
data = Dataset.load_builtin('ml-100k')
# data = Dataset.load_builtin('ml-1m')
```

The SVD (https://surprise.readthedocs.io/en/stable/matrix_factorization.html#surprise.prediction_algorithms.matrix_factorization.SVD) has a lot of parameters. The most important ones are:

- `n_factors` , which controls the dimension of the latent space (i.e. the size of the vectors p_u and q_i). Usually, the quality of the training set predictions grows with as `n_factors` gets higher.
- `n_epochs` , which defines the number of iteration of the SGD procedure.

Note that both parameter also affect the training time.

We will here set `n_factors` to `200` and `n_epochs` to `30` . To train the model, we simply need to call the `fit()` method.

In [10]:
```
algo = SVD(random_state=0, n_factors=200, n_epochs=30, verbose=False)
```

Running classical K-fold cross-validation procedure with 3 splits to evalaute the recommender system

In [11]:
```
# define a cross-validation iterator
kf = KFold(n_splits=3)

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

    # train and test algorithm.
    algo.fit(trainset)
    predictions = algo.test(testset)

    # Compute and print Root Mean Squared Error
    accuracy.rmse(predictions, verbose=True)
```

```
RMSE: 0.9563
RMSE: 0.9531
RMSE: 0.9570
```

Task 2

Read about input parameters of [SVD algotirhm (https://surprise.readthedocs.io/en/stable/matrix_factorization.html#surprise.prediction_algorithms.matrix_factorization.SVD)](https://surprise.readthedocs.io/en/stable/matrix_factorization.html#surprise.prediction_algorithms.matrix_factorization.SVD):

Then, use GridSearchCV tool (https://surprise.readthedocs.io/en/stable/getting_started.html#tune-algorithm-parameters-with-gridsearchcv) to find parameter combination that yields the best results?

Task 3

Using entire dataset, build a recommeder system using Surprise.SVD algorithm with best set of imput paarammeters which you found in previous task, and then Write function that generates the top k recommendations for specific user?

Note: the generated recommendation should exclude the movies that have been already rated the user.

Conclusion

We've seen that we can make good recommendations with raw data based collaborative filtering methods (neighborhood models) and latent features from low-rank matrix factorization methods (factorization models).

Low-dimensional matrix recommenders try to capture the underlying features driving the raw data (which we understand as tastes and preferences). From a theoretical perspective, if we want to make recommendations based on people's tastes, this seems like the better approach. This technique also scales **significantly** better to larger datasets.