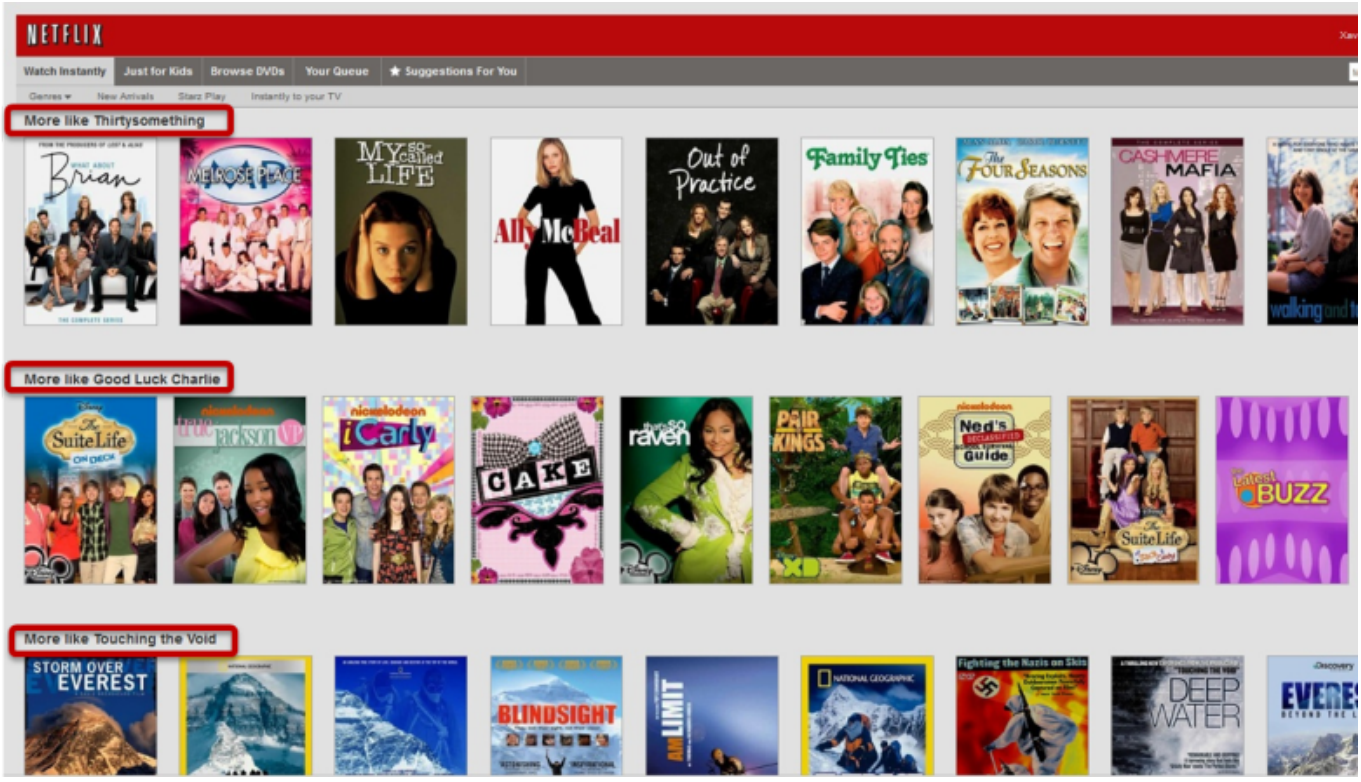
Prototyping a Recommender System Step by Step Part 2: Alternating Least Square (ALS) Matrix Factorization in Collaborative Filtering

Kevin Liao Follow
Nov 17, 2018 · 9 min read



Item Based Collaborative Filtering Movie Recommender

Part 1 of *recommender systems* can be found *here*

In the *last post*, we covered a lot of ground in how to build our own recommender systems and got our hand dirty with **Pandas** and **Scikit-learn** to implement a **KNN item-**

based collaborative filtering movie recommender. The source code of the KNN recommender system can be found in my *Github repo*.

In this post, we will talk about how to improve our movie recommender system with a more sophisticated machine learning technique: **Matrix Factorization**. Later in this post, we will discuss why we want to use matrix factorization in collaborative filtering and what is matrix factorization and how is it implemented in **Spark**.

Shortcomings in Item Based Collaborative Filtering

```
You have input movie: Iron Man
Found possible matches in our database: ['Iron Man (2008)', 'Iron Man 3 (2013)', 'Iron Man 2 (2010)']

Recommendation system start to make inference
......

It took my system 1.39s to make inference

Recommendations for Iron Man:
1: Bourne Ultimatum, The (2007), with distance of 0.42213231325149536
2: Sherlock Holmes (2009), with distance of 0.4194795489311218
3: Inception (2010), with distance of 0.3934664726257324
4: Avatar (2009), with distance of 0.3836246728897095
5: WALL·E (2008), with distance of 0.3835691213607788
6: Star Trek (2009), with distance of 0.37533360719680786
7: Batman Begins (2005), with distance of 0.37033611536026
8: Iron Man 2 (2010), with distance of 0.3703038692474365
9: Avengers, The (2012), with distance of 0.35819315910339355
10: Dark Knight, The (2008), with distance of 0.3013307452201843
```

Output of KNN Item Based Collaborative Filtering Recommender From Previous Post

During the last section in *previous post*, we asked our model for some movie recommendations. After we evaluated the list of recommended movies, we quickly identified two obvious limitations in our KNN approach. One is the “popularity bias”, the other is “item cold-start problem”. There will be another limitation, “scalability issue”, if the underlying training data is too big to fit in one machine

- **popularity bias:** refers to system recommends the movies with the most interactions without any personalization
- **item cold-start problem**: refers to when movies added to the catalogue have either none or very little interactions while recommender rely on the movie’s interactions to make recommendations
- **scalability issue**: refers to lack of the ability to scale to much larger sets of data when more and more users and movies added into our database

All three above are very typical challenges for collaborative filtering recommender. They arrive naturally along with the user-movie (or movie-user) interaction matrix where each entry records an interaction of a user `i` and a movie `j`. In a real world setting, the vast majority of movies receive very few or even no ratings at all by users. We are looking at an extremely sparse matrix with more than 99% of entries are missing values.

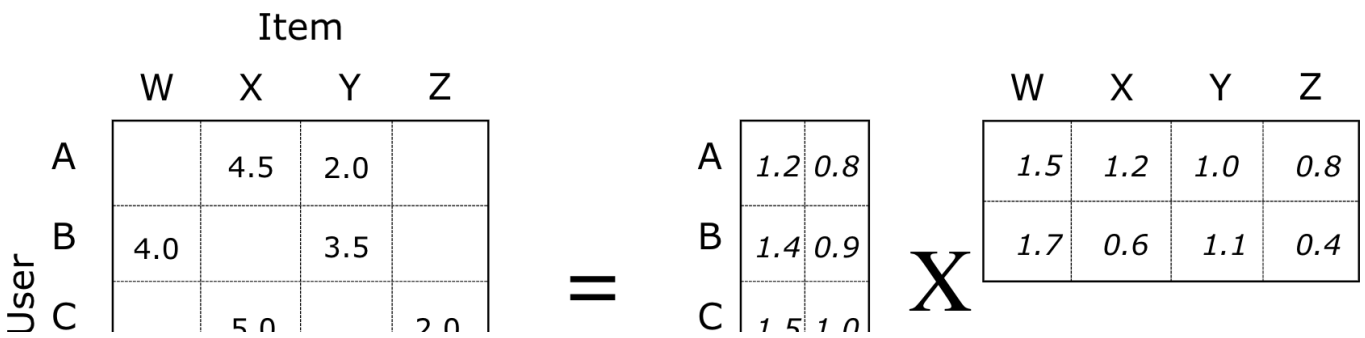
userId	4	5	10	14	15	18	19	26	31	34	...	283199	283204	283206	283208	283210	283215	283219	283222	283224
movieId																				
1	4.0	NaN	5.0	4.5	4.0	NaN	NaN	NaN	5.0	NaN	...	5.0	NaN	NaN	4.5	NaN	4.0	4.0	NaN	NaN
2	4.0	NaN	NaN	4.0	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	5.0	NaN	NaN	4.0
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	2.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	5.0	NaN	NaN	NaN
6	4.5	NaN	NaN	NaN	NaN	3.0	4.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	5.0	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
10	4.0	NaN	NaN	NaN	NaN	3.0	4.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	3.0
11	3.5	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	3.0	NaN	NaN	NaN
12	NaN	NaN	NaN	NaN	NaN	NaN	3.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
13	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
14	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	3.0
15	NaN	NaN	NaN	NaN	NaN	NaN	3.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

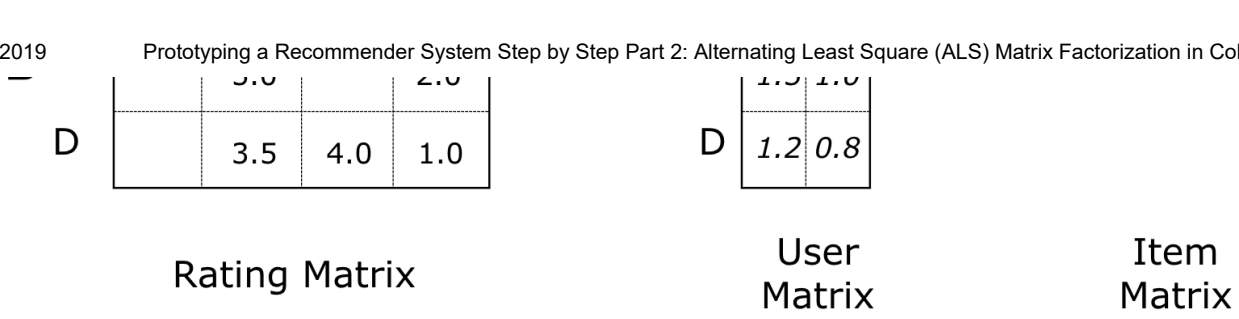
Sparse Rating Data

With such a sparse matrix, what ML algorithms can be trained and reliable to make inference? To find solutions to the question, we are effectively solving a data sparsity problem.

Matrix Factorization

In collaborative filtering, **matrix factorization** is the state-of-the-art solution for sparse data problem, although it has become widely known since *Netflix Prize Challenge*.





Matrix Factorization of Movie Ratings Data

What is matrix factorization? Matrix factorization is simply a family of mathematical operations for matrices in linear algebra. To be specific, a matrix factorization is a factorization of a matrix into a product of matrices. In the case of collaborative filtering, **matrix factorization** algorithms work by **decomposing** the user-item interaction matrix into the product of two **lower dimensionality rectangular matrices**. One matrix can be seen as the user matrix where rows represent users and columns are latent factors. The other matrix is the item matrix where rows are latent factors and columns represent items.

How does matrix factorization solve our problems?

1. Model learns to factorize rating matrix into user and movie representations, which allows model to predict better personalized movie ratings for users
2. With matrix factorization, less-known movies can have rich latent representations as much as popular movies have, which improves recommender's ability to recommend less-known movies

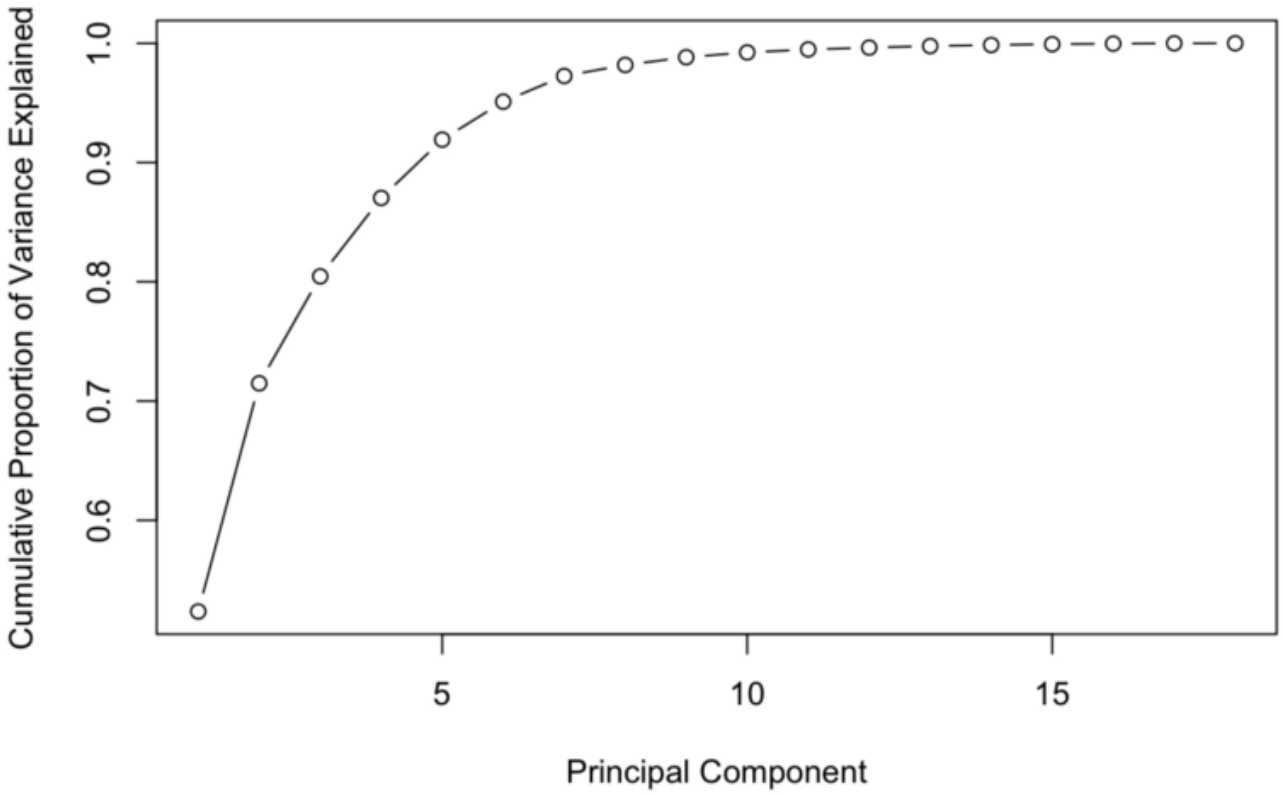
In the sparse user-item interaction matrix, the predicted rating user `u` will give item `i` is computed as:

$$\tilde{r}_{ui} = \sum_{f=0}^{nfactors} H_{u,f} W_{f,i}$$

where H is user matrix, W is item matrix

Rating of item `i` given by user `u` can be expressed as a dot product of the user latent vector and the item latent vector.

Notice in above formula, the number of **latent factors** can be tuned via cross-validation. **Latent factors** are the features in the lower dimension latent space projected from user-item interaction matrix. The idea behind matrix factorization is to use latent factors to represent user preferences or movie topics in a much lower dimension space. Matrix factorization is one of very effective **dimension reduction** techniques in machine learning.



Variance Explained By Components In PCA

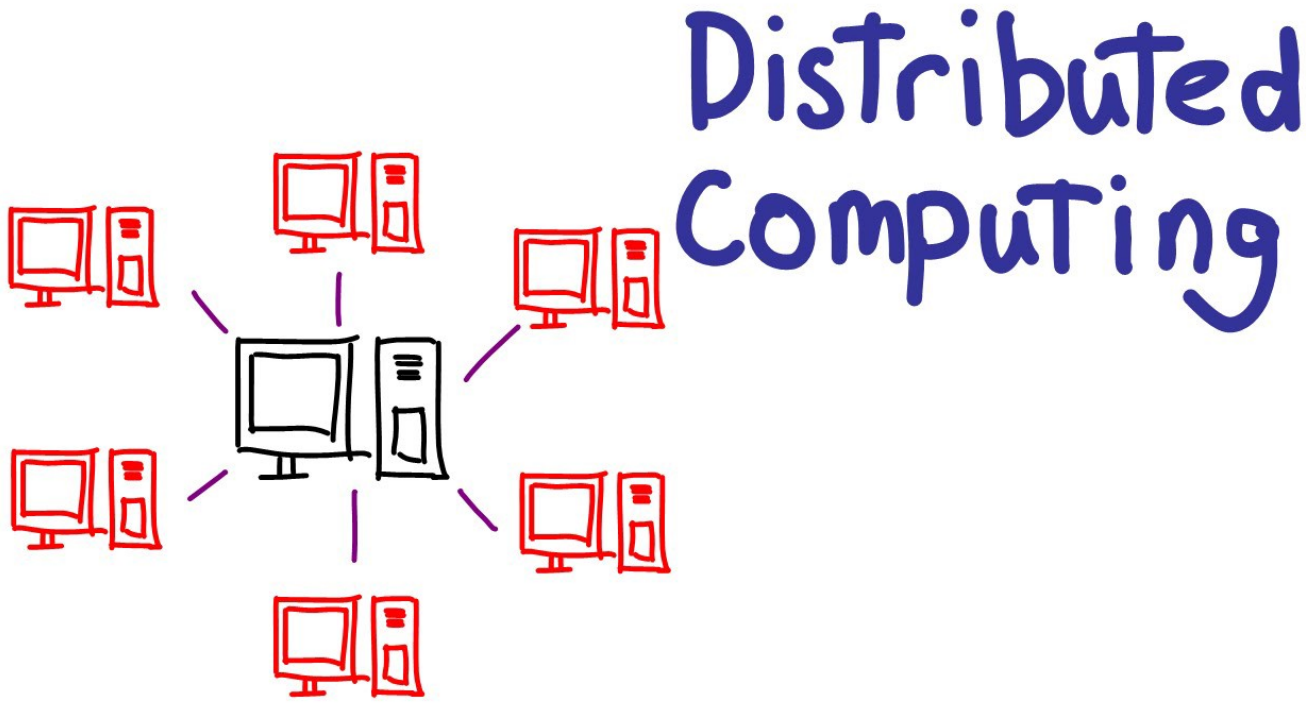
Very much like the concept of **components** in **PCA**, the number of latent factors determines the amount of abstract information that we want to store in a lower dimension space. A matrix factorization with one latent factor is equivalent to a *most popular* or *top popular* recommender (e.g. recommends the items with the most interactions without any personalization). Increasing the number of latent factors will improve personalization, until the number of factors becomes too high, at which point the model starts to overfit. A common strategy to avoid overfitting is to add **regularization terms** to the objective function.

The objective of matrix factorization is to minimize the error between true rating and predicted rating:

$$\arg\min_{H,W} \|R - \tilde{R}\|_{\mathrm{F}} + \alpha\|H\| + \beta\|W\|$$

where H is user matrix, W is item matrix

Once we have an objective function, we just need a training routine (eg, gradient descent) to complete the implementation of a matrix factorization algorithm. This implementation is actually called **Funk SVD**. It is named after Simon Funk, who he shared his findings with the research community during Netflix prize challenge in 2006.



Scaling Machine Learning Applications With Distributed Computing

Although Funk SVD was very effective in matrix factorization with single machine during that time, it's not **scalable** as the amount of data grows today. With terabytes or even petabytes of data, it's impossible to load data with such size into a single machine. So we need a machine learning model (or framework) that can train on dataset spreading across from cluster of machines.

Alternating Least Square (ALS) with Spark ML

Alternating Least Square (ALS) is also a matrix factorization algorithm and it runs itself in a parallel fashion. ALS is implemented in Apache Spark ML and built for a larges-scale collaborative filtering problems. ALS is doing a pretty good job at solving scalability and sparseness of the Ratings data, and it's simple and scales well to very large datasets.

Some high-level ideas behind ALS are:

- Its objective function is slightly different than Funk SVD: ALS uses **L2 regularization** while Funk uses **L1 regularization**
- Its training routine is different: ALS minimizes **two loss functions alternatively**; It first holds user matrix fixed and runs gradient descent with item matrix; then it holds item matrix fixed and runs gradient descent with user matrix
- Its scalability: ALS runs its gradient descent in **parallel** across multiple partitions of the underlying training data from a cluster of machines

SGD Algorithm for MF

Input: training matrix V, the number of features K, regularization parameter λ, learning rate ϵ
Output: row related model matrix W and column related model matrix H

1: Initialize W, H to $UniformReal(0, \frac{1}{\sqrt{K}})$
2: **repeat**
3: **for random** $V_{ij} \in V$ **do**
4: $error = W_{i*}H_{*j} - V_{ij}$
5: $W_{i*} = W_{i*} - \epsilon(error \cdot H_{*j}^{\intercal} + \lambda W_{i*})$
6: $H_{*j} = H_{*j} - \epsilon(error \cdot W_{i*}^{\intercal} + \lambda H_{*j})$
7: **end for**
8: **until** convergence

Pseudocode For SGD In Matrix Factorization

If you are interested in learning more about ALS, you can find more details in this paper: *Large-scale Parallel Collaborative Filtering for the Netflix Prize*

Just like other machine learning algorithms, ALS has its own set of hyper-parameters. We probably want to tune its hyper-parameters via **hold-out validation** or **cross-validation**.

Most important hyper-params in Alternating Least Square (ALS):

- maxIter: the maximum number of iterations to run (defaults to 10)
- rank: the number of latent factors in the model (defaults to 10)
- regParam: the regularization parameter in ALS (defaults to 1.0)

Hyper-parameter tuning is a highly recurring task in many machine learning projects. We can code it up in a function to speed up the tuning iterations.

```
from pyspark.ml.recommendation import ALS


def tune_ALS(train_data, validation_data, maxIter, regParams, ranks):
    """
    grid search function to select the best model based on RMSE of
    validation data

    Parameters
    ----------
    train_data: spark DF with columns ['userId', 'movieId', 'rating']

    validation_data: spark DF with columns ['userId', 'movieId', 'rating']

    maxIter: int, max number of learning iterations

    regParams: list of float, one dimension of hyper-param tuning grid

    ranks: list of float, one dimension of hyper-param tuning grid

    Return
    ------
    The best fitted ALS model with lowest RMSE score on validation data
    """
    # initial
    min_error = float('inf')
    best_rank = -1
    best_regularization = 0
    best_model = None
    for rank in ranks:
        for reg in regParams:
            # get ALS model
```

```
            als = ALS().setMaxIter(maxIter).setRank(rank).setRegParam(reg)
            # train ALS model
            model = als.fit(train_data)
            # evaluate the model by computing the RMSE on the validation data
            predictions = model.transform(validation_data)
            evaluator = RegressionEvaluator(metricName="rmse",
                                            labelCol="rating",
                                            predictionCol="prediction")
            rmse = evaluator.evaluate(predictions)
            print('{} latent factors and regularization = {}: '
                  'validation RMSE is {}'.format(rank, reg, rmse))
            if rmse < min_error:
                min_error = rmse
                best_rank = rank
                best_regularization = reg
                best_model = model
    print('\nThe best model has {} latent factors and '
          'regularization = {}'.format(best_rank, best_regularization))
    return best_model
```

After tuning, we found the best choice of hyper-parameters: `maxIter=10`, `regParam=0.05`, `rank=20`

Implementing ALS Recommender System

Now that we know we have a wonderful model for movie recommendation, the next question is: how do we take our wonderful model and productize it into a recommender system? Machine learning model productization is another big topic and I won't get into details about it. In this post, I will show how to build a MVP (minimum viable product) version for ALS recommender.

To productize a model, we need to build a work flow around the model. Typical ML work flow roughly starts with data preparation via pre-defined set of ETL jobs, offline/online model training, then ingesting trained models to web services for production. In our case, we are going to build a very minimum version of movie recommender that just does the job. Our work flow is following:

1. A new user inputs his/her favorite movies, then system create new user-movie interaction samples for the model

2. System retrains ALS model on data with the new inputs
3. System creates movie data for inference (in my case, I sample all movies from the data)
4. System make rating predictions on all movies for that user
5. System outputs top N movie recommendations for that user based on the ranking of movie rating predictions

Here is a small snippet of the source code for our MVP recommender system:

```
def make_recommendations(self, fav_movie, n_recommendations):
    """
    make top n movie recommendations
    Parameters
    ----------
    fav_movie: str, name of user input movie
    n_recommendations: int, top n recommendations
    """
    # get data
    movie_user_mat_sparse, hashmap = self._prep_data()
    # get recommendations
    raw_recommends = self._inference(
        self.model, movie_user_mat_sparse, hashmap,
        fav_movie, n_recommendations)
    # print results
    reverse_hashmap = {v: k for k, v in hashmap.items()}
    print('Recommendations for {}:'.format(fav_movie))
    for i, (idx, dist) in enumerate(raw_recommends):
        print('{0}: {1}, with distance '
              'of {2}'.format(i+1, reverse_hashmap[idx], dist))
```

make recommendations als.py hosted with ♡ by GitHub view raw

snippet of our final step in ALS recommender's implementation

This snippet demos our `make_recommendations` method in our recommender's implementation. Please find the detailed source code for recommender application in my *GitHub Repo*.

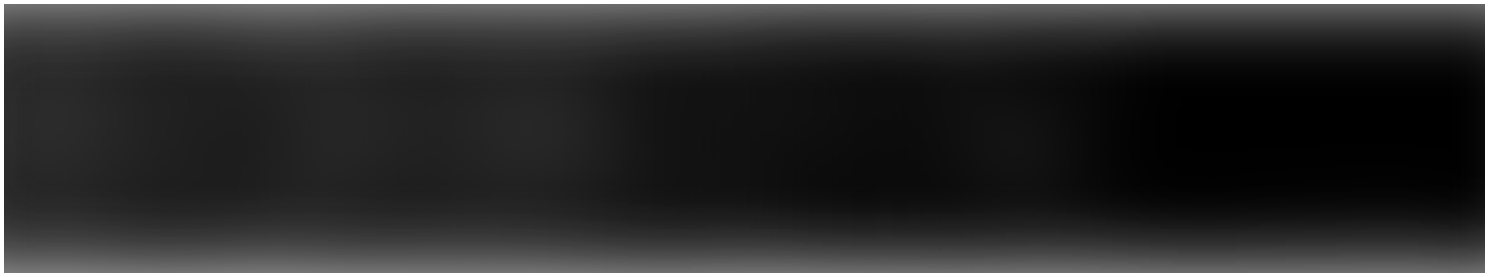
Let's Make Some Recommendations

Once we implemented the ALS recommender system in a python script as a small **Pyspark** program, we can submit our spark application to a cluster with Client Deploy Mode or Cluster Deploy Mode and enjoy the power of distributed computing.

Finally, we are done with the technical details and implementations. Now let's ask our recommender for some movie recommendations. I will pretend a new user and input my favorite movie "Iron Man" again into this new recommender system. Let's see what movies it recommends to me. Hopefully, they are not the same list of popular movies that I have watched many times before.

For demo purpose, I submit my spark application locally by running following commends in terminal: (instruction of commands can be found here)

```
spark-submit --master local[4] --driver-memory 4g
             --executor-memory 8g src/als_recommender.py
             --movie_name "Iron Man" --top_n 10
```



List of Movie Recommendations Based On My Favorite Movie: "Iron Man"

Yay!! Successfully run our movie recommender with Spark.

This new list of recommended movies are completely different than the list from previous KNN recommender, which is very interesting!! I have never watch any one of movies from this new list. I find it very surprising that the new recommender proposed me unusual movies. They might be too unusual for other users, which is problematic.

One idea to further improve our movie recommender system is to blend this new list of movie recommendations with the previous list from KNN recommender. We basically implement a hybrid recommender system and this hybrid recommender can offer both popular and less-know content to users.

Summary

In this post, we covered how to improve collaborative filtering recommender system with **matrix factorization**. We learned that matrix factorization can solve "popular bias" and "item cold-start" problems in collaborative filtering. We also leveraged **Spark ML** to implement distributed recommender system using **Alternating Least Square (ALS)**. The **Jupyter Notebook version** for this blog post can be found *here*. If you want to play around my **source code**, you can find it *here*.

In my next post, we will dig deeper at Matrix Factorization techniques. We can develop a more generalized form of matrix factorization model with **neural network** implementation in **Keras.** Stay tuned! Until then, have fun with machine learning and recommenders!

. . .

Like what you read? Checkout more data science / machine learning projects at my Github: ***Kevin's Data Science Portfolio***

Machine Learning Data Science Recommendation System Distributed Systems Spark

About Help Legal