# Spark with Python: collaborative filtering

SPARK WITH PYTHON: COLLABORATIVE FILTERING

## Table of content

## Collaborative filtering with ALS

The algorithm implemented for collaborative filtering (CF) in Spark MLlib is *Alternative Least Squares (ALS) with Weight Regularization*. The algorithm is described in the research paper *'Large-scale Parallel Collaborative Filtering for the Netflix Prize'*. I would assume there exist better algorithm out there. However, ALS is the one implemented in MLlib. So be it. I just try to sketch the general idea of the algorithm as in the following bullet list. Interesting read can go to other external reference for a comprehensive story.

- The idea is similar as matrix factorization. In particular, ALS assumes that the rating matrix $R$ can be factorized into two *non-negative* matrices, a user-preference matrix $U$ and a preference-rating matrix $M$. The intuition behind is that the non-negative matrix operation will correspond to a linear combination only in terms of additions. In addition, the additivity property is quite natural in real world applications, e.g., topic modelling in which a document is comprised with a collection of topics, image classification in which an image is essentially a collection of objects.

- The loss function used in ALS is so called *rooted mean square error (RMSE)* defined as

  $$\mathcal{L}(R,U,M) = \frac{1}{n}\sum_{i,j}(r_{i,j} - < u_i, m_j >)^2,$$

  where $n$ is the number of entries in the rating matrix $R$.

- In addition, ALS applies $L_2$-norm regularization on the parameter spaces $U$ and $M$.
- Combine the loss function, the objective of ALS can be formulated as

  $$\min_{U,M} \frac{1}{n}\sum_{i,j}(r_{i,j} - < u_i, m_j >)^2 + \lambda(\sum_i n_{n_i} u_i^2 + \sum_i n_{m_i} m_i^2),$$

  where $\lambda$ is the regularization parameter that controls the balance of the loss term and the regularization term, $n_{u_i}$ is the number of movies rated by user $i$, and $n_{m_i}$ is the number of users that rate movies $i$.

- The above optimization problem is convex in terms of either $U$ and $M$. Therefore, it can be solved with an iterative approach where solving $U$ whiling fixing $M$, and vice versa.
- When fixing $M$ and optimizing $U$, the problem is equivalent to a collection of ridge regression problems where each subproblem takes $u_i$ as parameter and $R$, $M$ as constance. Therefore, it can be optimized in parallel in terms of $u_i$.
- In particular, the subproblem can be solve analytically as a ridge regression.

## Spark Python code

##General information

- Collaborative filtering (CF) is heavily used in recommender system where the task is to find the missing values in the user-item association matrix.
- The following code is to use ALS algorithm implemented in Spark MLlib for recommendation.
- The data file used here is the well known MovieLens dataset. In particular, two variants are used in the experiences reported in the following section:
  1. 1 million ratings from 6000 users on 4000 movies
  2. 10 million ratings from 70000 users on 11000 movies.
- The format of the file is `UserID::MovieID::Rating::Time`.
- The basic idea of the Python script:
  1. First select from original dataset two subsets, one for training and the other for test.

2. I learn a ALS model based on training data which includes extensive parameter selections.
3. The performance on training data is then compared with an naive imputation model known as mean imputation.
4. After training phase, The model is applied on test data to estimate the preference of user-item pairs.
5. The performance of the model on test data is again compared with the naive mean imputation method.

- The complete Python script for the experiment can be found from my Github page.
- Remember that you can monitor the progress of the running Python code from command line interface `lynx http://localhost:8080`.
- When running ALS natively for very large dataset, e.g. 10 million ratings, the Spark will complain about the memory issues. The solution is to write memory require into the configuration files according to the following
    - Edit the file `conf/spark-env.sh`.
    - After adding following lines to the file, Spark will work nicely again :bowtie:

```
export SPARK_DAEMON_MEMORY=8g
export SPARK_WORKER_MEMORY=8g
export SPARK_DAEMON_JAVA_OPTS="-Xms8g -Xmx8g"
export SPARK_JAVA_OPTS="-Xms8g -Xmx8g"
export SPARK_LOCAL_DIRS='/cs/work/group/urenzyme/workspace/SparkViaPython/tmp/'
```

- Sometimes, Spark will also complain about memory issue. This is, in my case, local `/tmp/` directory is almost full. The solution is to specify another temporary directory for Spark by writing again the configuration file according to the following
    - Edit the file `conf/spark-env.sh`.
    - After adding following line to the file, Spark will work again :laughing:

```
export SPARK_LOCAL_DIRS='/cs/work/group/urenzyme/workspace/SparkViaPython/tmp/'
```

### Results

## 1 million ratings

- Statistics of the dataset

| Name | Number |
|---|---|
| ratings | 1000209 |

| Name | Number |
|---|---|
| **Training** | 978241 |
| **Test** | 21968 |

- Parameter selections

| Rank | $\lambda$ | Iteration | RMSE |
|---|---|---|---|
| 30 | 0.1 | 10 | 0.812829467087 |
| 30 | 0.1 | 15 | 0.805940272574 |
| 30 | 0.1 | 20 | 0.802684279113 |
| 30 | 0.01 | 10 | 0.630386656389 |
| 30 | 0.01 | 15 | 0.625034196609 |
| 30 | 0.01 | 20 | 0.622905635641 |
| 30 | 0.001 | 10 | 0.631097769333 |
| 30 | 0.001 | 15 | 0.623913185952 |
| 30 | 0.001 | 20 | 0.619115901472 |
| 20 | 0.1 | 10 | 0.81763450096 |
| 20 | 0.1 | 15 | 0.809997696948 |
| 20 | 0.1 | 20 | 0.806740178999 |
| 20 | 0.01 | 10 | 0.687371500823 |
| 20 | 0.01 | 15 | 0.684068044569 |
| 20 | 0.01 | 20 | 0.682312433409 |
| 20 | 0.001 | 10 | 0.689387829401 |
| 20 | 0.001 | 15 | 0.682992017723 |
| 20 | 0.001 | 20 | 0.68042871175 |
| 10 | 0.1 | 10 | 0.829041851 |
| 10 | 0.1 | 15 | 0.82183469039 |
| 10 | 0.1 | 20 | 0.819218398722 |
| 10 | 0.01 | 10 | 0.761779803655 |
| 10 | 0.01 | 15 | 0.757920617128 |
| 10 | 0.01 | 20 | 0.756874383345 |
| 10 | 0.001 | 10 | 0.759740415789 |
| 10 | 0.001 | 15 | 0.757288020603 |
| 10 | 0.001 | 20 | 0.75646083268 |

Best parameters

| Rank | $\lambda$ | Iteration | RMSE |
|---|---|---|---|
| 30 | 0.001 | 20 | 0.619115901472 |

- If taking the best parameter on training data, the performances on training and test sets are listed in the following table.

| | **ALS** | **Mean imputation** |
|---|---|---|
| **Training** | 0.62 | 1.12 |
| **Test** | 1.18 | 1.12 |

- If taking the second best parameter on training data, the performance on training and test sets are listed in the following table.

| | **ALS** | **Mean imputation** |
|---|---|---|
| **Training** | 0.62 | 1.12 |
| **Test** | 1.19 | 1.12 |

- If taking the third best parameter on training data, the performance on training and test sets are listed in the following table.

| | **ALS** | **Mean imputation** |
|---|---|---|
| **Training** | 0.62 | 1.12 |
| **Test** | 0.96 | 1.12 |

- It seems that we should not overfit training data :scream: :angry: But the question is how to select best parameter based on test data. I guess the most common way is to perform cross validation rather than the above training and test separation.

## 10 million ratings

- Statistics of the dataset

| **Name** | **Number** |
|---|---|
| ratings | 10000054 |
| **Training** | 9786084 |
| **Test** | 213970 |

- Parameter selections

| **Rank** | $\lambda$ | **Iteration** | **RMSE** |
|---|---|---|---|
| 30 | 0.1 | 10 | 0.789595015096 |
| 30 | 0.1 | 15 | 0.783871458306 |
| 30 | 0.1 | 20 | 0.781022279169 |
| 30 | 0.01 | 10 | 0.629501460062 |
| 30 | 0.01 | 15 | 0.624860412768 |

| Rank | | Iteration | RMSE |
|---|---|---|---|
| 30 | 0.01 | 20 | 0.622747739521 |
| 30 | 0.001 | 10 | 0.625608789618 |
| 30 | 0.001 | 15 | 0.619326932555 |
| 30 | 0.001 | 20 | 0.615837106427 |
| 20 | 0.1 | 10 | 0.787573198049 |
| 20 | 0.1 | 15 | 0.783031125788 |
| 20 | 0.1 | 20 | 0.781263906207 |
| 20 | 0.01 | 10 | 0.670921064045 |
| 20 | 0.01 | 15 | 0.667651507438 |
| 20 | 0.01 | 20 | 0.66548013614 |
| 20 | 0.001 | 10 | 0.668088452457 |
| 20 | 0.001 | 15 | 0.663146686623 |
| 20 | 0.001 | 20 | 0.661223585465 |
| 10 | 0.1 | 10 | 0.791031639862 |
| 10 | 0.1 | 15 | 0.785616978954 |
| 10 | 0.1 | 20 | 0.784268636801 |
| 10 | 0.01 | 10 | 0.7284122791 |
| 10 | 0.01 | 15 | 0.725975535523 |
| 10 | 0.01 | 20 | 0.725448043578 |
| 10 | 0.001 | 10 | 0.724738139253 |
| 10 | 0.001 | 15 | 0.722580933105 |
| 10 | 0.001 | 20 | 0.721248563089 |

- Performance on training and test sets

| | ALS | Mean imputation |
|---|---|---|
| **Training** | 0.62 | 1.06 |
| **Test** | 0.98 | 1.06 |

- As you can see ALS actually improves the RMSE :+1: :v:

### Coding details

- Python script of the following codes can be found from HERE.
- To use Spark Python interface we have to include Spark-Python package

```
from pyspark import SparkConf, SparkContext
from pyspark.mllib.recommendation import ALS
import itertools
from math import sqrt
```

```
import sys
from operator import add
```

- The next step is to configure the current python script with Spark context. In particular, we use local machine for testing the code and use cluster to run the script.

```
# set up Spark environment
APP_NAME = "Collaboratove filtering for movie recommendation"
conf = SparkConf().setAppName(APP_NAME)
conf = conf.setMaster('spark://ukko160:7077')
sc = SparkContext(conf=conf)
```

- After that, we have to read in the data file as RDD and take a look at the summary of the data

```
# read in data
data = sc.textFile(filename)
ratings = data.map(parseRating)
numRatings  = ratings.count()
numUsers    = ratings.values().map(lambda r:r[0]).distinct().count()
numMovies   = ratings.values().map(lambda r:r[1]).distinct().count()
print "--- %d ratings from %d users for %d movies\n" % (numRatings, numUsers, numMovies)
```

- The `parseRating` function is defined as

```
def parseRating(line):
  """
  Parses a rating record in MovieLens format userId::movieId::rating::timestamp.
  """
  fields = line.strip().split("::")
  return (int(int(fields[0])%10),int(int(fields[1])%10)), (int(fields[0]), int(fields[1]), floa
```

- Then we will partition the data into training, validation and test partitions. However,for the purpose of demonstration we use all data for training validation and test. In particular, we get all data from RDD and repartition the data.

```
numPartitions = 10
training    = ratings.filter(lambda r: not(r[0][0]<=0 and r[0][1]<=1) ).values().repartition(nu
test        = ratings.filter(lambda r: r[0][0]<=0 and r[0][1]<=1 ).values().cache()
numTraining = training.count()
numTest     = test.count()
print "ratings:\t%d\ntraining:\t%d\ntest:\t\t%d\n" % (ratings.count(), training.count(),test.cc
```

- After that we will run ALS with parameter selection on the training and validation sets. The performance of the model is measured with *rooted mean square error (RMSE)*.

## model training with parameter selection on the validation dataset

```
ranks       = [10,20,30]
lambdas     = [0.1,0.01,0.001]
numIters    = [10,20]
bestModel   = None
bestValidationRmse = float("inf")
bestRank    = 0
bestLambda  = -1.0
bestNumIter = -1
for rank, lmbda, numIter in itertools.product(ranks, lambdas, numIters):
  model                    = ALS.train(training, rank, numIter, lmbda)
  predictions              = model.predictAll(training.map(lambda x:(x[0],x[1])))
  predictionsAndRatings    = predictions.map(lambda x:((x[0],x[1]),x[2])).join(training.map(lamb
  validationRmse           = sqrt(predictionsAndRatings.map(lambda x: (x[0] - x[1]) ** 2).reduce
  print rank, lmbda, numIter, validationRmse
  if (validationRmse < bestValidationRmse):
    bestModel = model
    bestValidationRmse = validationRmse
    bestRank = rank
    bestLambda = lmbda
    bestNumIter = numIter
print bestRank, bestLambda, bestNumIter, bestValidationRmse
print "ALS on train:\t\t%.2f" % bestValidationRmse
```

- Use mean imputation to test the performance on training data.

```
meanRating = training.map(lambda x: x[2]).mean()
baselineRmse = sqrt(training.map(lambda x: (meanRating - x[2]) ** 2).reduce(add) / numTraining)
print "Mean imputation:\t\t%.2f" % baselineRmse
```

- The prediction of the best model on the test data can be computed from

```
  # predict test ratings
  try:
    predictions              = bestModel.predictAll(test.map(lambda x:(x[0],x[1])))
    predictionsAndRatings    = predictions.map(lambda x:((x[0],x[1]),x[2])).join(test.map(lambda
    testRmse           = sqrt(predictionsAndRatings.map(lambda x: (x[0] - x[1]) ** 2).reduce(add
  except Exception as myerror:
    print myerror
    testRmse           = sqrt(test.map(lambda x: (x[0] - 0) ** 2).reduce(add) / float(numTest))
  print "ALS on test:\t%.2f" % testRmse
```

- We can also compare the performance of ALS with naive approach where we predict all ratings with the average ratings.

```
# use mean rating as predictions
meanRating = training.map(lambda x: x[2]).mean()
baselineRmse = sqrt(test.map(lambda x: (meanRating - x[2]) ** 2).reduce(add) / numTest)
print "Mean imputation:\t%.2f" % baselineRmse
```

- When everything is done, we will stop Spark context with the following Python command.

```
# shut down spark
sc.stop()
```

## External sources

- "Alternating least square method for collaborative filtering" is an OK blog about basic knowledge of ALS and CF. The blog post also includes some running Python code. However, it is not about Spark MLlib.
- "Scalable Collaborative Filtering with Spark MLlib" is a nice article from Databricks in which the performance of Spark MLlib is compared with Mahout. It is worth looking at the actual code behind the scene.
- This post from Stackoverflow confirms my intuition that ALS in Spark-MLlib does not support the predictions for unseen users/movies. Basically, this means it would be tricky to select examples (ratings) to form training and test sets.
- This is the original documentation of ALS in Spark.
- Hand on exercises about recommender system in Spark origanized by Databricks.

Hongyu Su 13 October 2015

**0 Comments** **www.hongyusu.com** 1 **Login**

Recommend Share **Sort by Best**

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

ALSO ON **WWW.HONGYUSU.COM**

**Data ingestion: Flume and Sqoop**

3 comments • 5 months ago•

xiaohan2012 — 哈哈，在家上网，没事来你博客逛逛

**One class classification with Scikit**

1 comment • 8 months ago•

Yusn — Nice job! But I am curious about why the performance on train example is poor? But every other digits can be ...

*Content by* Hongyu Su
(Some rights reserved)

Powered by Jekyll
Styled by Mark Reid