

# tutorial\_spark

December 28, 2015

## 1 MLlib Tutorial

IPython notebooks consist of multiple “cells”, which can contain text (markdown) or code. You can run a single cell, multiple cells at a time, or every cell in the file. IPython also gives you the ability to export the notebook into different formats, such as a python file, a PDF file, an HTML file, etc.

Spend a few minutes and make yourself familiar with the IPython interface, it should be pretty straight forward.

The Machine Learning Library contains common machine learning algorithms and utilities. A summary of these features can be found [here](#).

In this tutorial, we will explore [collaborative filtering](#) on a small example problem. For your submission, you must turn in a PDF of this IPython notebook, but fully completed with any missing code.

### 1.1 Collaborative Filtering Example

We will go through an exercise based on the example from the collaborative filtering page. The first step is to import the MLlib library and whatever functions we want with this module.

```
In [2]: from pyspark.mllib.recommendation import ALS, Rating
```

#### 1.1.1 Loading the Input Data

Next, we must load the data. There are some example datasets that come with Spark by default. Example data related to machine learning in particular is located in the `$SPARK_HOME/data/mllib` directory. For this part, we will be working with the `$SPARK_HOME/data/mllib/als/test.data` file. This is a small dataset, so it is easy to see what is happening.

```
In [5]: data = sc.textFile("/Users/george/Panzer/Softwares/spark-1.5.2-bin-hadoop2.6/data/mllib/als/test.data")
```

Even though, we have the environment `$SPARK_HOME` defined, but it can't be used here. You must specify the full path, or the relative path based off where you initiated IPython.

The `textFile` command will create an RDD where each element is a line of the input file. In the below cell, write some code to (1) print the number of elements and (2) print the fifth element. Print your result in a single line with the format: “There are X elements. The fifth element is: Y”.

```
In [9]: rows = data.collect()
        x = len(rows)
        y = rows[4]
        print("There are %d elements. The fifth element is : %s"%(x,y))
```

```
There are 16 elements. The fifth element is : 2,1,5.0
```

### 1.1.2 Transforming the Input Data

This data isn't in a great format, since each element in the RDD is currently a string. However, we will assume that the first column of the string represents a user ID, the second column represents a product ID, and the third column represents a user-specified rating of that product.

In the below cell, write a function that takes a string (that has the same format as lines in this file) as input and returns a tuple where the first and second elements are ints and the third element is a float. **Call your function parser.**

We will then use this function to transform the RDD.

```
In [10]: def parser(line):
         splits = line.strip().split(",")
         return (int(splits[0]), int(splits[1]), float(splits[2]))
```

```
In [11]: ratings = data.map(parser).map(lambda l: Rating(*l))
         ratings.collect()
```

```
Out[11]: [Rating(user=1, product=1, rating=5.0),
          Rating(user=1, product=2, rating=1.0),
          Rating(user=1, product=3, rating=5.0),
          Rating(user=1, product=4, rating=1.0),
          Rating(user=2, product=1, rating=5.0),
          Rating(user=2, product=2, rating=1.0),
          Rating(user=2, product=3, rating=5.0),
          Rating(user=2, product=4, rating=1.0),
          Rating(user=3, product=1, rating=1.0),
          Rating(user=3, product=2, rating=5.0),
          Rating(user=3, product=3, rating=1.0),
          Rating(user=3, product=4, rating=5.0),
          Rating(user=4, product=1, rating=1.0),
          Rating(user=4, product=2, rating=5.0),
          Rating(user=4, product=3, rating=1.0),
          Rating(user=4, product=4, rating=5.0)]
```

Your output should look like the following: [Rating(user=1, product=1, rating=5.0), Rating(user=1, product=2, rating=1.0), Rating(user=1, product=3, rating=5.0), Rating(user=1, product=4, rating=1.0), Rating(user=2, product=1, rating=5.0), Rating(user=2, product=2, rating=1.0), Rating(user=2, product=3, rating=5.0), Rating(user=2, product=4, rating=1.0), Rating(user=3, product=1, rating=1.0), Rating(user=3, product=2, rating=5.0), Rating(user=3, product=3, rating=1.0), Rating(user=3, product=4, rating=5.0), Rating(user=4, product=1, rating=1.0), Rating(user=4, product=2, rating=5.0), Rating(user=4, product=3, rating=1.0), Rating(user=4, product=4, rating=5.0)]

If it doesn't, then you did something wrong! If it does match, then you are ready to move to the next step.

### 1.1.3 Building and Running the Model

Now we are ready to build the actual recommendation model using the Alternating Least Squares algorithm. The documentation can be found [here](#), and the papers the algorithm is based on are linked off the [collaborative filtering page](#).

```
In [12]: rank = 10
         numIterations = 10
         model = ALS.train(ratings, rank, numIterations)
```

```
In [13]: # Let's define some test data
testdata = ratings.map(lambda p: (p[0], p[1]))

# Running the model on all possible user->product predictions
predictions = model.predictAll(testdata)
predictions.collect()

Out[13]: [Rating(user=4, product=4, rating=4.9961077306520405),
Rating(user=4, product=1, rating=1.0015598285948302),
Rating(user=4, product=2, rating=4.9961077306520405),
Rating(user=4, product=3, rating=1.0015598285948302),
Rating(user=1, product=4, rating=1.0015455887603184),
Rating(user=1, product=1, rating=4.9961794580448355),
Rating(user=1, product=2, rating=1.0015455887603184),
Rating(user=1, product=3, rating=4.9961794580448355),
Rating(user=2, product=4, rating=1.0015455887603184),
Rating(user=2, product=1, rating=4.9961794580448355),
Rating(user=2, product=2, rating=1.0015455887603184),
Rating(user=2, product=3, rating=4.9961794580448355),
Rating(user=3, product=4, rating=4.9961077306520405),
Rating(user=3, product=1, rating=1.0015598285948302),
Rating(user=3, product=2, rating=4.9961077306520405),
Rating(user=3, product=3, rating=1.0015598285948302)]
```

#### 1.1.4 Transforming the Model Output

This result is not really in a nice format. Write some code that will transform the RDD so that each element is a user ID and a dictionary of product->rating pairs. Note that for the a Ratings object (which is what the elements of the RDD are), you can access the different fields by via the `.user`, `.product`, and `.rating` variables. For example, `predictions.take(1)[0].user`.

Call the new RDD `userPredictions`. It should look as follows (when using `userPredictions.collect()`):

```
[(4,
 {1: 1.0011434289237737,
  2: 4.996713610813412,
  3: 1.0011434289237737,
  4: 4.996713610813412}),
 (1,
 {1: 4.996411869659315,
  2: 1.0012037253934976,
  3: 4.996411869659315,
  4: 1.0012037253934976}),
 (2,
 {1: 4.996411869659315,
  2: 1.0012037253934976,
  3: 4.996411869659315,
  4: 1.0012037253934976}),
 (3,
 {1: 1.0011434289237737,
  2: 4.996713610813412,
  3: 1.0011434289237737,
  4: 4.996713610813412})]
```

```
In [22]: def format_ratings(lst):
ratings = {}
```

```

    for rating in lst:
        ratings[rating.product] = rating.rating
    return ratings
userPredictions = predictions.groupBy(lambda r: r.user).mapValues(format_ratings)
userPredictions.collect()

```

```

Out[22]: [(4,
  {1: 1.0015598285948302,
   2: 4.9961077306520405,
   3: 1.0015598285948302,
   4: 4.9961077306520405}),
 (1,
  {1: 4.9961794580448355,
   2: 1.0015455887603184,
   3: 4.9961794580448355,
   4: 1.0015455887603184}),
 (2,
  {1: 4.9961794580448355,
   2: 1.0015455887603184,
   3: 4.9961794580448355,
   4: 1.0015455887603184}),
 (3,
  {1: 1.0015598285948302,
   2: 4.9961077306520405,
   3: 1.0015598285948302,
   4: 4.9961077306520405})]

```

### 1.1.5 Evaluating the Model

Now, lets calculate the mean squared error.

```

In [23]: userPredictions = predictions.map(lambda r: ((r[0],r[1]), r[2]))
        ratesAndPreds = ratings.map(lambda r: ((r[0],r[1]), r[2])).join(predictions)
        MSE = ratesAndPreds.map(lambda r: (r[1][0] - r[1][1])**2).mean()
        print("Mean Squared Error = " + str(MSE))

```

Mean Squared Error = 0.0

### 1.1.6 Reflections

Why do you believe that this model achieved such a low error. Is there anything we did incorrectly for testing this model?

-----  
 The model was tested on the same training dataset. Thus it yielded such a low error. For testing the model approaches like

- a) splitting the data into a training set to build the model and test set to verify the model can be used.
- b) k-fold cross validation can also be adopted, where k times we split the data into training and test set in the ratio k-1:1 and compute the MSE for each split.