codementor COMMUNITY

Building a Movie Recommendation Service with Apache Spark & Flask - Part 1

Jose A Dianes
Data Analytics & Visualisation - SW Engineer PhD
Follow
Search post
Published Jul 07, 2015Last updated Sep 14, 2015
1
Write a post
SIGN UP

This Apache Spark tutorial will guide you step-by-step into how to use the MovieLens dataset to build a movie recommender using collaborative filtering with Spark's Alternating Least Saqures implementation. It is organised in two parts. The first one is about getting and parsing movies and ratings data into Spark RDDs. The second is about building and using the recommender and persisting it for later use in our on-line recommender system.

This tutorial can be used independently to build a movie recommender model based on the MovieLens dataset. Most of the code in the first part, about how to use ALS with the public MovieLens dataset, comes from my solution to one of the exercises proposed in the CS100.1x Introduction to Big Data with Apache Spark by Anthony D. Joseph on edX, that is also **publicly available since 2014 at Spark Summit**. Starting from there, I've added with minor modifications to use a larger dataset, then code about how to store and reload the model for later use, and finally a web service using Flask.

In any case, the use of this algorithm with this dataset is not new (you can Google about it), and this is because we put the emphasis on ending up with a usable model in an on-line environment, and how to use it in different situations. But I truly got inspired by solving the exercise proposed in that course, and I highly recommend you to take it. There you will learn not just ALS but many other Spark algorithms.

It is the second part of the tutorial the one that explains how to use Python/Flask for building a web-service on top of Spark models. By doing so, you will be able to develop a complete **on-line movie recommendation service**.

All the code for this tutorial is available in a GitHub repo. There is also a repo explaining many Spark-related concepts. Go there and make them yours.

Getting and processing the data

In order to build an on-line movie recommender using Spark, we need to have our model data as preprocessed as possible. Parsing the dataset and building the model everytime a new recommendation needs to be done is not the best of the strategies.

The list of task we can pre-compute includes:

- Loading and parsing the dataset. Persisting the resulting RDD for later use.
- Building the recommender model using the complete dataset. Persist the dataset for later use.

This notebook explains the first of these tasks.

File download

GroupLens Research has collected and made available rating data sets from the MovieLens web site. The data sets were collected over various periods of time, depending on the size of the set. They can be found here.

In our case, we will use the latest datasets:

- Small: 100,000 ratings and 2,488 tag applications applied to 8,570 movies by 706 users. Last updated 4/2015.
- Full: 21,000,000 ratings and 470,000 tag applications applied to 27,000 movies by 230,000 users. Last updated 4/2015.

```
complete_dataset_url = 'http://files.grouplens.org/datasets/movielens/ml-latest.zip'
small_dataset_url = 'http://files.grouplens.org/datasets/movielens/ml-latest-small.zip'
```

We also need to define download locations.

```
import os

datasets_path = os.path.join('..', 'datasets')

complete_dataset_path = os.path.join(datasets_path, 'ml-latest.zip')
small_dataset_path = os.path.join(datasets_path, 'ml-latest-small.zip')
```

Now we can proceed with both downloads.

```
import urllib

small_f = urllib.urlretrieve (small_dataset_url, small_dataset_path)
complete_f = urllib.urlretrieve (complete_dataset_url, complete_dataset_path)
```

Both of them are zip files containing a folder with ratings, movies, etc. We need to extract them into its individual folders so we can use each file later on.

```
import zipfile

with zipfile.ZipFile(small_dataset_path, "r") as z:
    z.extractall(datasets_path)

with zipfile.ZipFile(complete_dataset_path, "r") as z:
    z.extractall(datasets_path)
```

Loading and parsing datasets

Now we are ready to read in each of the files and create an RDD consisting of parsed lines.

Each line in the ratings dataset (`ratings.csv`) is formatted as:

```
userId,movieId,rating,timestamp
```

Each line in the movies (`movies.csv`) dataset is formatted as:

```
movieId,title,genres
```

Were *genres* has the format:

```
Genre1|Genre2|Genre3...
```

The tags file (`tags.csv`) has the format:

```
userId,movieId,tag,timestamp
```

And finally, the `links.csv` file has the format:

```
movieId,imdbId,tmdbId
```

The format of these files is uniform and simple, so we can use Python `split()` to parse their lines once they are loaded into RDDs. Parsing the movies and ratings files yields two RDDs:

- For each line in the ratings dataset, we create a tuple of `(UserID, MovieID, Rating)`. We drop the *timestamp* because we do not need it for this recommender.
- For each line in the movies dataset, we create a tuple of `(MovieID, Title)`. We drop the *genres* because we do not use them for this recommender.

So let's load the raw ratings data. We need to filter out the header, included in each file.

```
small_ratings_file = os.path.join(datasets_path, 'ml-latest-small', 'ratings.csv')

small_ratings_raw_data = sc.textFile(small_ratings_file)
small_ratings_raw_data_header = small_ratings_raw_data.take(1)[0]
```

Now we can parse the raw data into a new RDD.

```
small_ratings_data = small_ratings_raw_data.filter(lambda line: line!=small_ratings_raw_data_header)\
    .map(lambda line: line.split(",")).map(lambda tokens: (tokens[0],tokens[1],tokens[2])).cache()
```

For illustrative purposes, we can take the first few lines of our RDD to see the result. In the final script we don't call any Spark action (e.g. `take`) until needed, since they trigger actual computations in the cluster.

```
small_ratings_data.take(3)
```

```
[(u'1', u'6', u'2.0'), (u'1', u'22', u'3.0'), (u'1', u'32', u'2.0')]
```

We proceed in a similar way with the `movies.csv` file.

```
small_movies_file = os.path.join(datasets_path, 'ml-latest-small', 'movies.csv')

small_movies_raw_data = sc.textFile(small_movies_file)
small_movies_raw_data_header = small_movies_raw_data.take(1)[0]
```

```
small_movies_data = small_movies_raw_data.filter(lambda line: line!=small_movies_raw_data_header)\
    .map(lambda line: line.split(",")).map(lambda tokens: (tokens[0],tokens[1])).cache()

small_movies_data.take(3)
```

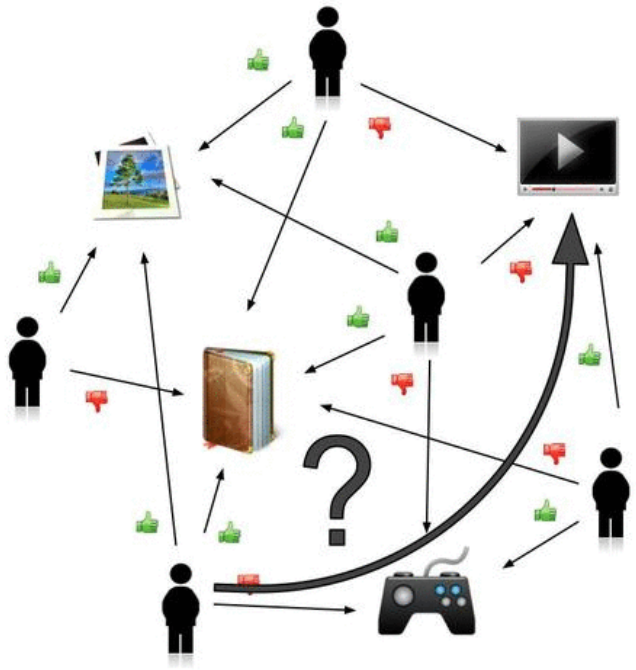
```
[(u'1', u'Toy Story (1995)'),
 (u'2', u'Jumanji (1995)'),
 (u'3', u'Grumpier Old Men (1995)')]
```

The following sections introduce *Collaborative Filtering* and explain how to use *Spark MLlib* to build a recommender model. We will close the tutorial by explaining how a model such this is used to make recommendations, and how to persist it for later use (e.g. in our Python/flask web-service).

Collaborative Filtering

In Collaborative filtering we make predictions (filtering) about the interests of a user by collecting preferences or taste information from many users (collaborating). The underlying assumption is that if a user A has the same opinion as a user B on an issue, A is more likely to have B's opinion on a different issue x than to have the opinion on x of a user chosen randomly.

The image below (from Wikipedia) shows an example of collaborative filtering. At first, people rate different items (like videos, images, games). Then, the system makes predictions about a user's rating for an item not rated yet. The new predictions are built upon the existing ratings of other users with similar ratings with the active user. In the image, the system predicts that the user will not like the video.



Spark MLlib library for Machine Learning provides a Collaborative Filtering implementation by using Alternating Least Squares. The implementation in MLlib has the following parameters:

- numBlocks is the number of blocks used to parallelize computation (set to -1 to auto-configure).
- rank is the number of latent factors in the model.
- iterations is the number of iterations to run.
- lambda specifies the regularization parameter in ALS.
- implicitPrefs specifies whether to use the explicit feedback ALS variant or one adapted for implicit feedback data.
- alpha is a parameter applicable to the implicit feedback variant of ALS that governs the baseline confidence in preference observations.

Selecting ALS parameters using the small dataset

In order to determine the best ALS parameters, we will use the small dataset. We need first to split it into train, validation, and test datasets.

```
training_RDD, validation_RDD, test_RDD = small_ratings_data.randomSplit([6, 2, 2], seed=0L)
validation_for_predict_RDD = validation_RDD.map(lambda x: (x[0], x[1]))
test_for_predict_RDD = test_RDD.map(lambda x: (x[0], x[1]))
```

Now we can proceed with the training phase.

```
from pyspark.mllib.recommendation import ALS
import math

seed = 5L
iterations = 10
regularization_parameter = 0.1
```

```
ranks = [4, 8, 12]
errors = [0, 0, 0]
err = 0
tolerance = 0.02

min_error = float('inf')
best_rank = -1
best_iteration = -1
for rank in ranks:
    model = ALS.train(training_RDD, rank, seed=seed, iterations=iterations,
                      lambda_=regularization_parameter)
    predictions = model.predictAll(validation_for_predict_RDD).map(lambda r: ((r[0], r[1]), r[2]))
    rates_and_preds = validation_RDD.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(predictions)
    error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean())
    errors[err] = error
    err += 1
    print 'For rank %s the RMSE is %s' % (rank, error)
    if error < min_error:
        min_error = error
        best_rank = rank

print 'The best model was trained with rank %s' % best_rank
```

```
For rank 4 the RMSE is 0.963681878574
For rank 8 the RMSE is 0.96250475933
For rank 12 the RMSE is 0.971647563632
The best model was trained with rank 8
```

But let's explain this a little bit. First, let's have a look at how our predictions look.

```
predictions.take(3)
```

```
[((32, 4018), 3.280114696166238),
 ((375, 4018), 2.7365714977314086),
 ((674, 4018), 2.510684514310653)]
```

Basically we have the UserID, the MovieID, and the Rating, as we have in our ratings dataset. In this case the predictions third element, the rating for that movie and user, is the predicted by our ALS model.

Then we join these with our validation data (the one that includes ratings) and the result looks as follows:

```
rates_and_preds.take(3)
```

```
[((558, 788), (3.0, 3.0419325487471403)),
 ((176, 3550), (4.5, 3.3214065001580986)),
 ((302, 3908), (1.0, 2.4728711204440765))]
```

To that, we apply a squared difference and the we use the `mean()` action to get the MSE and apply `sqrt`.

Finally we test the selected model.

```
model = ALS.train(training_RDD, best_rank, seed=seed, iterations=iterations,
                      lambda_=regularization_parameter)
predictions = model.predictAll(test_for_predict_RDD).map(lambda r: ((r[0], r[1]), r[2]))
rates_and_preds = test_RDD.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(predictions)
error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean())

print 'For testing data the RMSE is %s' % (error)
```

```
For testing data the RMSE is 0.972342381898
```

Using the complete dataset to build the final model

In order to build our recommender model, we will use the complete dataset. Therefore, we need to process it the same way we did with the small dataset.

```
# Load the complete dataset file
complete_ratings_file = os.path.join(datasets_path, 'ml-latest', 'ratings.csv')
complete_ratings_raw_data = sc.textFile(complete_ratings_file)
complete_ratings_raw_data_header = complete_ratings_raw_data.take(1)[0]

# Parse
complete_ratings_data = complete_ratings_raw_data.filter(lambda line: line!=complete_ratings_raw_data_header)\
    .map(lambda line: line.split(",")).map(lambda tokens: (int(tokens[0]),int(tokens[1]),float(tokens[2]))).cache()

print "There are %s recommendations in the complete dataset" % (complete_ratings_data.count())
```

```
There are 21063128 recommendations in the complete dataset
```

Now we are ready to train the recommender model.

```
training_RDD, test_RDD = complete_ratings_data.randomSplit([7, 3], seed=0L)

complete_model = ALS.train(training_RDD, best_rank, seed=seed,
                           iterations=iterations, lambda_=regularization_parameter)
```

Now we test on our testing set.

```
test_for_predict_RDD = test_RDD.map(lambda x: (x[0], x[1]))

predictions = complete_model.predictAll(test_for_predict_RDD).map(lambda r: ((r[0], r[1]), r[2]))
rates_and_preds = test_RDD.map(lambda r: ((int(r[0]), int(r[1])), float(r[2]))).join(predictions)
error = math.sqrt(rates_and_preds.map(lambda r: (r[1][0] - r[1][1])**2).mean())

print 'For testing data the RMSE is %s' % (error)
```

```
For testing data the RMSE is 0.82183583368
```

We can see how we got a more accurate recommender when using a much larger dataset.

How to make recommendations

Although we aim at building an online movie recommender, now that we know how to have our recommender model ready, we can give it a try providing some movie recommendations. This will help us coding the recommending engine later on when building the web service, and will explain how to use the model in any other circumstances.

When using collaborative filtering, getting recommendations is not as simple as predicting for the new entries using a previously generated model. Instead, we need to train again the model but including the new user preferences in order to compare them with other users in the dataset. That is, the recommender needs to be trained every time we have new user ratings (although a single model can be used by multiple users of course!). This makes the process expensive, and it is one of the reasons why scalability is a problem (and Spark a solution!). Once we have our model trained, we can reuse it to obtain top recomendations for a given user or an individual rating for a particular movie. These are less costly operations than training the model itself.

So let's first load the movies complete file for later use.

```
complete_movies_file = os.path.join(datasets_path, 'ml-latest', 'movies.csv')
complete_movies_raw_data = sc.textFile(complete_movies_file)
complete_movies_raw_data_header = complete_movies_raw_data.take(1)[0]

# Parse
complete_movies_data = complete_movies_raw_data.filter(lambda line: line!=complete_movies_raw_data_header)\
    .map(lambda line: line.split(",")).map(lambda tokens: (int(tokens[0]),tokens[1],tokens[2])).cache()

complete_movies_titles = complete_movies_data.map(lambda x: (int(x[0]),x[1]))

print "There are %s movies in the complete dataset" % (complete_movies_titles.count())
```

```
There are 27303 movies in the complete dataset
```

Another thing we want to do, is give recommendations of movies with a certain minimum number of ratings. For that, we need to count the number of ratings per movie.

```
def get_counts_and_averages(ID_and_ratings_tuple):
    nratings = len(ID_and_ratings_tuple[1])
    return ID_and_ratings_tuple[0], (nratings, float(sum(x for x in ID_and_ratings_tuple[1]))/nratings)

movie_ID_with_ratings_RDD = (complete_ratings_data.map(lambda x: (x[1], x[2])).groupByKey())
movie_ID_with_avg_ratings_RDD = movie_ID_with_ratings_RDD.map(get_counts_and_averages)
movie_rating_counts_RDD = movie_ID_with_avg_ratings_RDD.map(lambda x: (x[0], x[1][0]))
```

Adding new user ratings

Now we need to rate some movies for the new user. We will put them in a new RDD and we will use the user ID 0, that is not assigned in the MovieLens dataset. Check the dataset movies file for ID to Tittle assignment (so you know what movies are you actually rating).

```
new_user_ID = 0

# The format of each line is (userID, movieID, rating)
new_user_ratings = [
     (0,260,4), # Star Wars (1977)
     (0,1,3), # Toy Story (1995)
     (0,16,3), # Casino (1995)
     (0,25,4), # Leaving Las Vegas (1995)
     (0,32,4), # Twelve Monkeys (a.k.a. 12 Monkeys) (1995)
     (0,335,1), # Flintstones, The (1994)
     (0,379,1), # Timecop (1994)
     (0,296,3), # Pulp Fiction (1994)
     (0,858,5) , # Godfather, The (1972)
     (0,50,4) # Usual Suspects, The (1995)
    ]
new_user_ratings_RDD = sc.parallelize(new_user_ratings)
print 'New user ratings: %s' % new_user_ratings_RDD.take(10)
```

```
New user ratings: [(0, 260, 9), (0, 1, 8), (0, 16, 7), (0, 25, 8), (0, 32, 9), (0, 335, 4), (0, 379, 3), (0, 296, 7), (0, 858, 10), (0, 50, 8)]
```

Now we add them to the data we will use to train our recommender model. We use Spark's `union()` transformation for this.

```
complete_data_with_new_ratings_RDD = complete_ratings_data.union(new_user_ratings_RDD)
```

And finally we train the ALS model using all the parameters we selected before (when using the small dataset).

```
from time import time
```

```
t0 = time()
new_ratings_model = ALS.train(complete_data_with_new_ratings_RDD, best_rank, seed=seed,
                              iterations=iterations, lambda_=regularization_parameter)
tt = time() - t0

print "New model trained in %s seconds" % round(tt,3)
```

```
New model trained in 56.61 seconds
```

It took some time. We will need to repeat that every time a user adds new ratings. Ideally we will do this in batches, and not for every single rating that comes into the system for every user.

Getting top recommendations

Let's now get some recommendations! For that we will get an RDD with all the movies the new user hasn't rated yet. We will them together with the model to predict ratings.

```
new_user_ratings_ids = map(lambda x: x[1], new_user_ratings) # get just movie IDs
# keep just those not on the ID list (thanks Lei Li for spotting the error!)
new_user_unrated_movies_RDD = (complete_movies_data.filter(lambda x: x[0] not in new_user_ratings_ids).map(lambda x: (new_user_ID, x[0])))

# Use the input RDD, new_user_unrated_movies_RDD, with new_ratings_model.predictAll() to predict new ratings for the movies
new_user_recommendations_RDD = new_ratings_model.predictAll(new_user_unrated_movies_RDD)
```

We have our recommendations ready. Now we can print out the 25 movies with the highest predicted ratings. And join them with the movies RDD to get the titles, and ratings count in order to get movies with a minimum number of counts. First we will do the join and see what does the result looks like.

```
# Transform new_user_recommendations_RDD into pairs of the form (Movie ID, Predicted Rating)
new_user_recommendations_rating_RDD = new_user_recommendations_RDD.map(lambda x: (x.product, x.rating))
new_user_recommendations_rating_title_and_count_RDD = \
    new_user_recommendations_rating_RDD.join(complete_movies_titles).join(movie_rating_counts_RDD)
new_user_recommendations_rating_title_and_count_RDD.take(3)
```

```
[(87040, ((6.834512984654888, u'"Housemaid'), 14)),
 (8194, ((5.966704041954459, u'Baby Doll (1956)'), 79)),
 (130390, ((0.6922328127396398, u'Contract Killers (2009)'), 1))]
```

So we need to flat this down a bit in order to have `(Title, Rating, Ratings Count)`.

```
new_user_recommendations_rating_title_and_count_RDD = \
    new_user_recommendations_rating_title_and_count_RDD.map(lambda r: (r[1][0][1], r[1][0][0], r[1][1]))
```

Finally, get the highest rated recommendations for the new user, filtering out movies with less than 25 ratings.

```
top_movies = new_user_recommendations_rating_title_and_count_RDD.filter(lambda r: r[2]>=25).takeOrdered(25, key=lambda x: -x[1])

print ('TOP recommended movies (with more than 25 reviews):\n%s' %
        '\n'.join(map(str, top_movies)))
```

```
TOP recommended movies (with more than 25 reviews):
    (u'"Godfather: Part II', 8.503749129186701, 29198)
    (u'"Civil War', 8.386497469089297, 257)
    (u'Frozen Planet (2011)', 8.372705479107108, 31)
    (u'"Shawshank Redemption', 8.258510064442426, 67741)
    (u'Cosmos (1980)', 8.252254825768972, 948)
    (u'Band of Brothers (2001)', 8.225114960311624, 4450)
    (u'Generation Kill (2008)', 8.206487040524653, 52)
    (u"Schindler's List (1993)", 8.172761674773625, 53609)
    (u'Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb (1964)', 8.166229786764168, 23915)
    (u"One Flew Over the Cuckoo's Nest (1975)", 8.15617022970577, 32948)
    (u'Casablanca (1942)', 8.141303207981174, 26114)
    (u'Seven Samurai (Shichinin no samurai) (1954)', 8.139633165142612, 11796)
    (u'Goodfellas (1990)', 8.12931139039048, 27123)
    (u'Star Wars: Episode V - The Empire Strikes Back (1980)', 8.124225700242096, 47710)
    (u'Jazz (2001)', 8.078538221315313, 25)
    (u"Long Night's Journey Into Day (2000)", 8.050176820606127, 34)
    (u'Lawrence of Arabia (1962)', 8.041331489948814, 13452)
    (u'Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981)', 8.0399424815528, 45908)
    (u'12 Angry Men (1957)', 8.011389274280754, 13235)
    (u"It's Such a Beautiful Day (2012)", 8.007734839026181, 35)
    (u'Apocalypse Now (1979)', 8.005094327199552, 23905)
    (u'Paths of Glory (1957)', 7.999379786394267, 3598)
    (u'Rear Window (1954)', 7.9860865203540214, 17996)
    (u'State of Play (2003)', 7.981582126801772, 27)
    (u'Chinatown (1974)', 7.978673289692703, 16195)
```

Getting individual ratings

Another useful usecase is getting the predicted rating for a particular movie for a given user. The process is similar to the previous retreival of top recommendations but, instead of using `predcitAll` with every single movie the user hasn't rated yet, we will just pass the method a single entry with the movie we want to predict the rating for.

```
my_movie = sc.parallelize([(0, 500)]) # Quiz Show (1994)
individual_movie_rating_RDD = new_ratings_model.predictAll(new_user_unrated_movies_RDD)
individual_movie_rating_RDD.take(1)
```

```
[Rating(user=0, product=122880, rating=4.955831875971526)]
```

Not very likely that the new user will like that one... Obviously we can include as many movies as we need in that list!

Persisting the model

Optionally, we might want to persist the base model for later use in our on-line recommendations. Although a new model is generated everytime we have new user ratings, it might be worth it to store the current one, in order to save time when starting up the server, etc. We might also save time if we persist some of the RDDs we have generated, specially those that took longer to process. For example, the following lines save and load a ALS model.

```
from pyspark.mllib.recommendation import MatrixFactorizationModel

model_path = os.path.join('..', 'models', 'movie_lens_als')

# Save and load model
model.save(sc, model_path)
same_model = MatrixFactorizationModel.load(sc, model_path)
```

Among other things, you will see in your filesystem that there are folder with product and user data into Parquet format files.

Genre and other fields

We haven't used the `genre` and `timestamp` fields in order to simplify the transformations and the whole tutorial. Incorporating them doesn't represent any problem. A good use could be filtering recommendations by any of them (e.g. recommendations by genre, or recent recommendations) like we have done with the minimum number of ratings.

Conclusions

Spark's MLlib library provides scalable data analytics through a rich set of methods. Its Alternating Least Squares implementation for Collaborative Filtering is one that fits perfectly in a recommendation engine. Due to its very nature, collaborative filtering is a costly procedure since requires updating its model when new user preferences arrive. Therefore, having a distributed computation engine such as Spark to perform model computation is a must in any real-world recommendation engine like the one we have built here.

Through this tutorial we have described how to build a model using Spark, how to perform some parameter selection using a reduced dataset, and how to update the model every time that new user preferences come in. Additionally, we have explained how the recommender is used in different situations and how its results are joined with product metadata (e.g. movie titles) in order to present its results in a proper way.

In part 2, we will go one step further and use our model in a web environment in order to provide on-line movie recommendations.

HN Submission/Discussion
sparkPythondata-science
Report

Enjoy this post? Give **Jose A Dianes** a like if it's helpful.

7
41
Share
Jose A Dianes
Data Analytics & Visualisation - SW Engineer PhD
With more than a decade of experience, I have been involved in different aspects of Data Analytics and Enterprise Software applied to domains such as Life Sciences, Ambient Sensing, and Real-time Simulators. I have a special int...
Follow
Discover and read more posts from **Jose A Dianes**
get started
Enjoy this post?

Leave a like and comment for **Jose**

7
41
41Replies
Leave a reply

GitHub flavored markdown supported
submit
Rohan Kalra

2 months ago

Hello Jose,
It's a great post. But I would like to know how can we recommend movies based on additional parameters including rating for example, improve recommendations based on user's age, sex, country, etc i.e. use multiple features for recommendations?

Reply
Avinash Navlani
2 months ago

Hello Jose,
Thanks a lot for your great tutorial. but when I ran the code "model.save(sc, model_path)". This was the error message:

Py4JJavaError Traceback (most recent call last)
<ipython-input-33-483ac9ad0430> in <module>()
4 path= "/home/khyati/Documents/models/movie_lens_als.model"
5 # Save and load model
----> 6 model.save(sc, path)
7
8 #same_model = MatrixFactorizationModel.load(sc, model_path)

/usr/local/spark/python/pyspark/mllib/util.pyc in save(self, sc, path)
404 if not isinstance(path, basestring):
405 raise TypeError("path should be a basestring, got type %s" % type(path))
–> 406 self._java_model.save(sc._jsc.sc(), path)
407
408

/usr/local/spark/python/lib/py4j-0.10.4-src.zip/py4j/java_gateway.py in **call**(self, *args)
1131 answer = self.gateway_client.send_command(command)
1132 return_value = get_return_value(
-> 1133 answer, self.gateway_client, self.target_id, self.name)
1134
1135 for temp_arg in temp_args:

/usr/local/spark/python/pyspark/sql/utils.pyc in deco(*a, **kw)
61 def deco(*a, **kw):
62 try:
—> 63 return f(*a, **kw)
64 except py4j.protocol.Py4JJavaError as e:
65 s = e.java_exception.toString()

/usr/local/spark/python/lib/py4j-0.10.4-src.zip/py4j/protocol.py in get_return_value(answer, gateway_client, target_id, name)
317 raise Py4JJavaError(
318 "An error occurred while calling {0}{1}{2}.\n".
–> 319 format(target_id, ".", name), value)
320 else:
321 raise Py4JError(

Py4JJavaError: An error occurred while calling o387.save.
: org.apache.hadoop.mapred.FileAlreadyExistsException: Output directory file:/home/khyati/Documents/models/movie_lens_als.model/metadata already exists
at org.apache.hadoop.mapred.FileOutputFormat.checkOutputSpecs(FileOutputFormat.java:131)
at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$1.applymcVsp(PairRDDFunctions.scala:1191)
at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$1.apply(PairRDDFunctions.scala:1168)
at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopDataset$1.apply(PairRDDFunctions.scala:1168)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
at org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopDataset(PairRDDFunctions.scala:1168)
at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$4.applymcVsp(PairRDDFunctions.scala:1071)
at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$4.apply(PairRDDFunctions.scala:1037)
at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$4.apply(PairRDDFunctions.scala:1037)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
at org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopFile(PairRDDFunctions.scala:1037)
at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$1.applymcVsp(PairRDDFunctions.scala:963)
at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$1.apply(PairRDDFunctions.scala:963)
at org.apache.spark.rdd.PairRDDFunctions$$anonfun$saveAsHadoopFile$1.apply(PairRDDFunctions.scala:963)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
at org.apache.spark.rdd.PairRDDFunctions.saveAsHadoopFile(PairRDDFunctions.scala:962)
at org.apache.spark.rdd.RDD$$anonfun$saveAsTextFile$1.applymcVsp(RDD.scala:1489)
at org.apache.spark.rdd.RDD$$anonfun$saveAsTextFile$1.apply(RDD.scala:1468)

at org.apache.spark.rdd.RDD$$anonfun$saveAsTextFile$1.apply(RDD.scala:1468)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
at org.apache.spark.rdd.RDD.withScope(RDD.scala:362)
at org.apache.spark.rdd.RDD.saveAsTextFile(RDD.scala:1468)
at org.apache.spark.mllib.recommendation.MatrixFactorizationModel$SaveLoadV1_0$.save(MatrixFactorizationModel.scala:361)
at org.apache.spark.mllib.recommendation.MatrixFactorizationModel.save(MatrixFactorizationModel.scala:206)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:357)
at py4j.Gateway.invoke(Gateway.java:280)
at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
at py4j.commands.CallCommand.execute(CallCommand.java:79)
at py4j.GatewayConnection.run(GatewayConnection.java:214)
at java.lang.Thread.run(Thread.java:748)

Do you have any idea? How to resolve this error?

Show more
Reply
Rodrigo Pereira
4 months ago

Greetings Jose.
First of all, thanks a lot for this tutorial.
It has been of great utility for me.
I have just a question: is the MLLIB recommender user-user or item-item? The documentation does not make it clear.

Kind Regards,

Rodrigo

Reply
Show more replies

Satwik Kansal

Python Practices for Efficient Code: Performance, Memory, and Usability

This is an updated version of my previous blog post on few recommended practices for optimizing your Python code.

A codebase that follows best practices is highly appreciated in today's world. It's an appealing way to engage awesome developers if your project is Open Source. As a developer, you want to write efficient and optimized code, which is:

> Code that takes up minimum possible memory, executes faster, looks clean, is properly documented, follows standard style guidelines, and ...

read more

Join and start **discussions** with fellow developers

start a discussion