learn-co-curriculum / **als-recommender-system-pyspark-lab**

No description, website, or topics provided.

4 commits **3** branches **0** packages **0** releases **2** contributors View license

Branch: **solution** New pull request Create new file Upload files Find file Clone or download

This branch is 3 commits ahead, 3 commits behind master. Pull request Compare

sumedh10 update readme Latest commit 97b2919 on Dec 12, 2019

data	initial commit	9 months ago
.gitignore	initial commit	9 months ago
.learn	initial commit	9 months ago
CONTRIBUTING.md	initial commit	9 months ago
LICENSE.md	initial commit	9 months ago
README.md	update readme	2 months ago
index.ipynb	update readme	2 months ago

README.md

Building a Recommendation System in PySpark - Lab

Introduction

In this lab, we will implement a movie recommendation system using ALS in Spark programming environment. Spark's machine learning library `ml` comes packaged with a very efficient implementation of the ALS algorithm that we looked at in the previous lesson. The lab will require you to put into practice your Spark programming skills for creating and manipulating PySpark DataFrames. We will go through a step-by-step process into developing a movie recommendation system using ALS and PySpark using the `MovieLens` dataset that we used in a previous lab.

Note: You are advised to refer to PySpark documentation heavily for completing this lab as it will introduce a few new methods.

Objectives

In this lab you will:

- Use Spark to train and cross-validate an ALS model
- Introduce a new user with rating to a rating matrix and make recommendations for them
- Create a function that will return the top n recommendations for a user

Building a Recommendation System

We have seen how recommendation systems have played an integral part in the success of Amazon (books, items), Pandora/Spotify (music), Google (news, search), YouTube (videos) etc. For Amazon, these systems bring more than 30% of their total revenue. For Netflix, 75% of movies that people watch are based on some sort of recommendation.

> The goal of recommendation systems is to find what is likely to be of interest to the user. This enables organizations to offer a high level of personalization and customer tailored services.

For online video content services like Netflix and Hulu, the need to build robust movie recommendation systems is extremely important. An example of a recommendation system is such as this:

1. User A watches Game of Thrones and Breaking Bad
2. User B performs a search query for Game of Thrones
3. The system suggests Breaking Bad to user B from data collected about user A

This lab will guide you through a step-by-step process into developing such a movie recommendation system. We will use the `MovieLens` dataset to build a movie recommendation system using the collaborative filtering technique with Spark's Alternating Least Squares implementation. After building that recommendation system, we will go through the process of adding a new user to the dataset with some new ratings and obtaining new recommendations for that user.

Importing the Data

- Initialize a `SparkSession` object
- Import the dataset found at `'./data/ratings.csv'` into a PySpark DataFrame

```
# import necessary libraries
from pyspark.sql import SparkSession

# instantiate SparkSession object
# spark = SparkSession.builder.master('local').getOrCreate()

spark = SparkSession\
        .builder\
        .appName('ALSExample').config('spark.driver.host', 'localhost')\
        .getOrCreate()


# read in the dataset into pyspark DataFrame
movie_ratings = spark.read.csv('./data/ratings.csv', header='true', inferSchema='true')
```

Check the data types of each of the columns to ensure that they are a type that makes sense given the column.

```
movie_ratings.dtypes
```

```
[('userId', 'int'),
 ('movieId', 'int'),
 ('rating', 'double'),
 ('timestamp', 'int')]
```

We aren't going to need the timestamp, so we can go ahead and remove that column.

```
movie_ratings = movie_ratings.drop('timestamp')
```

Fitting the Alternating Least Squares Model

Because this dataset is already preprocessed for us, we can go ahead and fit the Alternating Least Squares model.

- Import `ALS` from `pyspark.ml.recommendation` module
- Use the `.randomSplit()` method on the pyspark DataFrame to separate the dataset into training and test sets
- Fit the Alternating Least Squares Model to the training dataset. Make sure to set the `userCol`, `itemCol`, and `ratingCol` to the appropriate columns given this dataset. Then fit the data to the training set and assign it to a variable model

```
from pyspark.ml.evaluation import RegressionEvaluator

from pyspark.ml.recommendation import ALS

# split into training and testing sets
(training, test) = movie_ratings.randomSplit([0.8, 0.2])

# Build the recommendation model using ALS on the training data
# Note we set cold start strategy to 'drop' to ensure we don't get NaN evaluation metrics
als = ALS(maxIter=5,rank=4, regParam=0.01, userCol='userId', itemCol='movieId', ratingCol='rating',
          coldStartStrategy='drop')

# fit the ALS model to the training set
model = als.fit(training)
```

Now you've fit the model, and it's time to evaluate it to determine just how well it performed.

- Import `RegressionEvalutor` from `pyspark.ml.evaluation`
- Generate predictions with your model for the test set by using the `.transform()` method on your ALS model
- Evaluate your model and print out the RMSE from your test set

```
# importing appropriate library
from pyspark.ml.evaluation import RegressionEvaluator

# Evaluate the model by computing the RMSE on the test data
predictions = model.transform(test)
evaluator = RegressionEvaluator(metricName='rmse', labelCol='rating',
                                predictionCol='prediction')
rmse = evaluator.evaluate(predictions)
print('Root-mean-square error = ' + str(rmse))
```

```
Root-mean-square error = 0.9968853671625669
```

Cross-validation to Find the Optimal Model

Let's now find the optimal values for the parameters of the ALS model. Use the built-in `CrossValidator` in PySpark with a suitable param grid and determine the optimal model. Try with the parameters:

- regularization = [0.01, 0.001, 0.1]
- rank = [4, 10, 50]

```
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder

# initialize the ALS model
als_model = ALS(userCol='userId', itemCol='movieId',
                ratingCol='rating', coldStartStrategy='drop')
```

```
# create the parameter grid
params = ParamGridBuilder()\
          .addGrid(als_model.regParam, [0.01, 0.001, 0.1])\
          .addGrid(als_model.rank, [4, 10, 50]).build()


# instantiating crossvalidator estimator
cv = CrossValidator(estimator=als_model, estimatorParamMaps=params,evaluator=evaluator,parallelism=4)
best_model = cv.fit(movie_ratings)

# We see the best model has a rank of 50, so we will use that in our future models with this dataset
best_model.bestModel.rank
```

Incorporating the names of the movies

When we make recommendations, it would be ideal if we could have the actual name of the movie used rather than just an ID. There is another file called `'./data/movies.csv'` that contains all of the names of the movies matched up to the `movie_id` that we have in the ratings dataset.

- Import the data into a Spark DataFrame
- Look at the first 5 rows

```
movie_titles = spark.read.csv('./data/movies.csv',header='true',inferSchema='true')

movie_titles.head(5)
```

```
[Row(movieId=1, title='Toy Story (1995)', genres='Adventure|Animation|Children|Comedy|Fantasy'),
 Row(movieId=2, title='Jumanji (1995)', genres='Adventure|Children|Fantasy'),
 Row(movieId=3, title='Grumpier Old Men (1995)', genres='Comedy|Romance'),
 Row(movieId=4, title='Waiting to Exhale (1995)', genres='Comedy|Drama|Romance'),
 Row(movieId=5, title='Father of the Bride Part II (1995)', genres='Comedy')]
```

We will eventually be matching up the movie ids with the movie titles. In the cell below, create a function `name_retriever()` that takes in a `movie_id` and returns a string that represents the movie title.

> Hint: It's possible to do this operation in one line with the `df.where()` or the `df.filter()` methods

```
def name_retriever(movie_id, movie_title_df):
    return movie_title_df.where(movie_title_df.movieId == movie_id).take(1)[0]['title']


print(name_retriever(1023, movie_titles))
```

```
Winnie the Pooh and the Blustery Day (1968)
```

Getting Recommendations

Now it's time to actually get some recommendations! The ALS model has built-in methods called `.recommendForUserSubset()` and `.recommendForAllUsers()`. We'll start off with using a subset of users.

```
users = movie_ratings.select(als.getUserCol()).distinct().limit(1)
userSubsetRecs = model.recommendForUserSubset(users, 10)
recs = userSubsetRecs.take(1)
```

We can now see we have a list of rows with recommended items. Now try and get the name of the top recommended movie by way of the function you just created, using number one item for this user.

```
# use indexing to obtain the movie id of top predicted rated item
first_recommendation = recs[0]['recommendations'][0][0]

# use the name retriever function to get the values
name_retriever(first_recommendation,movie_titles)
```

```
'Pirate Radio (2009)'
```

Of course, you can also make recommendations for everyone, although this will take longer. In the next line, we are creating an RDD with the top 5 recommendations for every user and then selecting one user to find out his predictions:

```
recommendations = model.recommendForAllUsers(5)
recommendations.where(recommendations.userId == 3).collect()
```

Getting Predictions for a New User

Now, it's time to put together all that you've learned in this section to create a function that will take in a new user and some movies they've rated and then return n number of highest recommended movies. This function will have multiple different steps to it:

- Adding the new ratings into the DataFrame (hint: look into using the `.union()` method)
- Fitting the ALS model
- Make recommendations for the user of choice
- Print out the names of the top n recommendations in a reader-friendly manner

The function should take in the parameters:

- `user_id` : int
- `new_ratings` : list of tuples in the format (user_id, item_id, rating)
- `rating_df` : spark DF containing ratings
- `movie_title_df` : spark DF containing movie titles
- `num_recs` : int

Rate new movies

```
[Row(movieId=3253, title='Wayne's World (1992)', genres='Comedy'),
 Row(movieId=2459, title='Texas Chainsaw Massacre, The (1974)', genres='Horror'),
 Row(movieId=2513, title='Pet Sematary (1989)', genres='Horror'),
 Row(movieId=6502, title='28 Days Later (2002)', genres='Action|Horror|Sci-Fi'),
 Row(movieId=1091, title='Weekend at Bernie's (1989)', genres='Comedy'),
Row(movieId=441, title='Dazed and Confused (1993)', genres='Comedy'),
Row(movieId=370, title='Naked Gun 33 1/3: The Final Insult (1994)', genres='Action|Comedy')]
```

```
def new_user_recs(user_id, new_ratings, rating_df, movie_title_df, num_recs):
    # turn the new_recommendations list into a spark DataFrame
    new_user_ratings = spark.createDataFrame(new_ratings,rating_df.columns)

    # combine the new ratings df with the rating_df
    movie_ratings_combined = rating_df.union(new_user_ratings)

    # split the dataframe into a train and test set
#     (training, test) = movie_ratings_combined.randomSplit([0.8, 0.2],seed=0)

    # create an ALS model and fit it
    als = ALS(maxIter=5,rank=50, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating",
          coldStartStrategy="drop")
    model = als.fit(movie_ratings_combined)
```

```
    # make recommendations for all users using the recommendForAllUsers method
    recommendations = model.recommendForAllUsers(num_recs)

    # get recommendations specifically for the new user that has been added to the DataFrame
    recs_for_user = recommendations.where(recommendations.userId == user_id).take(1)

    for ranking, (movie_id, rating) in enumerate(recs_for_user[0]['recommendations']):
        movie_string = name_retriever(movie_id,movie_title_df)
        print('Recommendation {}: {}  | predicted score :{}'.format(ranking+1,movie_string,rating))
```

```
user_id = 100000
user_ratings_1 = [(user_id,3253,5),
                  (user_id,2459,5),
                  (user_id,2513,4),
                  (user_id,6502,5),
                  (user_id,1091,5),
                  (user_id,441,4)]
new_user_recs(user_id,
              new_ratings=user_ratings_1,
              rating_df=movie_ratings,
              movie_title_df=movie_titles,
              num_recs = 10)
```

```
Recommendation 1: Star Wars: Episode IV - A New Hope (1977)  | predicted score :5.517341136932373
Recommendation 2: Usual Suspects, The (1995)  | predicted score :5.442122936248779
Recommendation 3: In the Name of the Father (1993)  | predicted score :5.3851237297058105
Recommendation 4: Star Wars: Episode V - The Empire Strikes Back (1980)  | predicted score :5.381286144256592
Recommendation 5: Fight Club (1999)  | predicted score :5.361552715301514
Recommendation 6: Monty Python and the Holy Grail (1975)  | predicted score :5.347217559814453
Recommendation 7: Willy Wonka & the Chocolate Factory (1971)  | predicted score :5.328979969024658
Recommendation 8: Who Framed Roger Rabbit? (1988)  | predicted score :5.324649810791016
Recommendation 9: Clerks (1994)  | predicted score :5.305201530456543
Recommendation 10: Office Space (1999)  | predicted score :5.297811985015869
```

So here we have it! Our recommendation system is generating recommendations for the top 10 movies.

Level up (Optional)

- Create a user interface to allow users to easily choose items and get recommendations
- Use IMDB links to scrape user reviews from IMDB and using basic NLP techniques, create extra embeddings for ALS model
- Create a hybrid recommender system using features like genre

Summary

In this lab, you built a model using Spark, performed some parameter selection, and updated the model every time new user preferences came in. You looked at how Spark's ALS implementation can be used to build a scalable and efficient recommendation system. You also saw that such systems can become computationally expensive and using them with an online system could be a problem with traditional computational platforms. Spark's distributed computing architecture provides a great solution to deploy such recommendation systems for real-world applications (think Amazon, Spotify).