movie-recommendation

April 7, 2024

Create a SparkContext or handle existing context

[ ]:
```
import os
import sys
import pyspark as ps
import warnings
from pyspark.sql import SQLContext
os.environ['PYSPARK_PYTHON'] = sys.executable
os.environ['PYSPARK_DRIVER_PYTHON'] = sys.executable
try:
    sc = ps.SparkContext('local[*]')
    print("Just created a SparkContext")
except ValueError:
    warnings.warn("SparkContext already exists in this scope")
```

Imports the unittest module, which provides a framework for writing and running unit tests in Python.

Imports the sys module, which provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

class TestRdd(unittest.TestCase):
Defines a test case class named TestRdd, which inherits from unittest.TestCase. This class will contain test methods for testing the functionality of an RDD (Resilient Distributed Dataset) in a distributed computing framework like Apache Spark.

def test_take(self):
Defines a test method named test_take within the TestRdd class. This method tests the take() operation on an RDD, which returns the first n elements of the RDD.

input = sc.parallelize([1,2,3,4])
Creates an RDD named input by parallelizing the list [1, 2, 3, 4] using the sc object, which is likely a SparkContext instance.

self.assertEqual([1,2,3,4], input.take(4))
Asserts that the result of calling take(4) on the input RDD is equal to the list [1, 2, 3, 4]. If the assertion fails, the test case will fail.

def run_tests():
Defines a function named run_tests that runs the test suite.

suite = unittest.TestLoader().loadTestsFromTestCase(TestRdd)
Creates a test suite by loading all test cases from the TestRdd class using the unittest.TestLoader.

unittest.TextTestRunner(verbosity=1,stream=sys.stderr).run(suite)
Runs the test suite using a TextTestRunner with verbosity level 1 (prints a dot for each successful test) and directs the output to sys.stderr.

run_tests()

[ ]:
```
import unittest
import sys

class TestRdd(unittest.TestCase):
    def test_take(self):
        input = sc.parallelize([1,2,3,4])
        self.assertEqual([1,2,3,4], input.take(4))

def run_tests():
    suite = unittest.TestLoader().loadTestsFromTestCase( TestRdd )
    unittest.TextTestRunner(verbosity=1,stream=sys.stderr).run( suite )

run_tests()
```

`import pyspark`: This line imports the PySpark library, which is the Python API for Apache Spark.

`from pyspark.sql import SparkSession`: This line imports the `SparkSession` class from the `pyspark.sql` module, which is used to create a Spark session.

`def start():`: This defines a function named `start()` that creates a Spark session.

`spark = SparkSession.builder \`
`.appName("DataFrameExample") \`
`.getOrCreate()`: These lines create a new Spark session with the application name "DataFrame-Example". The `getOrCreate()` method ensures that if a Spark session already exists, it returns that session instead of creating a new one.

`return spark`: This line returns the created Spark session.

`sc = start()`: This line calls the `start()` function and assigns the returned Spark session to the variable `sc`.

`data = [(...), ...]`: This is a list of tuples representing the data to be used for creating a Spark DataFrame.

[ ]:
```
help(sc)
```

Imports the json module, which provides functionality for parsing and working with JSON data.

```
fields = ['product_id', 'user_id', 'score', 'time']
fields2 = ['product_id', 'user_id', 'review', 'profile_name', 'helpfulness', 'score', 'time']
fields3 = ['product_id', 'user_id', 'time']
fields4 = ['user_id', 'score', 'time']
```

Defines lists of field names that are expected in the JSON data.

Define a function `validate` that takes a line and checks if it contains all the fields specified in `fields2`. If any field is missing, it returns `False`, otherwise it returns `True`.

```
reviews_raw = sc.textFile('data/movies.json')
```

Reads the contents of the file 'data/movies.json' into an RDD (Resilient Distributed Dataset) named `reviews_raw`.

```
reviews = reviews_raw.map(lambda line: json.loads(line)).filter(validate)
```

Applies two transformations to the `reviews_raw` RDD:
1. `map(lambda line: json.loads(line))`: Converts each line (string) into a Python dictionary using `json.loads`.
2. `filter(validate)`: Filters out any dictionaries that do not pass the `validate` function.
The resulting RDD is stored in `reviews`.

```
reviews.cache()
```

Caches the `reviews` RDD in memory for faster access.

[ ]:
```
import json
fields = ['product_id', 'user_id', 'score', 'time']
fields2 = ['product_id', 'user_id', 'review', 'profile_name', 'helpfulness',␣
 ↪'score', 'time']
fields3 = ['product_id', 'user_id', 'time']
fields4 = ['user_id', 'score', 'time']
def validate(line):
    for field in fields2:
        if field not in line:
            return False
    return True

reviews_raw = sc.textFile('data/movies.json')
reviews = reviews_raw.map(lambda line: json.loads(line)).filter(validate)
reviews.cache()
reviews.take(1)
```

```
num_movies = reviews.groupBy(lambda entry: entry['product_id']).count()
```

This line groups the reviews DataFrame by the 'product_id' column (which represents movies), and counts the number of unique 'product_id' values, giving the total number of movies in the dataset.

```
num_users = reviews.groupBy(lambda entry: entry['user_id']).count()
```

This line groups the reviews DataFrame by the 'user_id' column, and counts the number of unique 'user_id' values, giving the total number of users in the dataset.

```
num_entries = reviews.count()
```

This line counts the total number of rows (reviews) in the reviews DataFrame.

```
[ ]: num_movies = reviews.groupBy(lambda entry: entry['product_id']).count()
num_users = reviews.groupBy(lambda entry: entry['user_id']).count()
num_entries = reviews.count()
print (str(num_entries) + " reviews of " + str(num_movies) + " movies by " +␣
 ↪str(num_users) + " different people.")
```

This line filters the `reviews` RDD to only include entries where the `user_id` key exists and has a non-empty value.

`r1 = reviews.map(lambda r: ((r['product_id'],), 1))`

This line creates a new RDD `r1` by mapping each review in the `reviews` RDD to a tuple containing the `product_id` as the key and the value 1.

`avg3 = r1.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))`

This line performs a map and reduce operation on the `r1` RDD. It first maps each value (which is 1) to a tuple of (value, 1), then reduces by key, summing up the values and counts for each `product_id`.

`avg3 = avg3.filter(lambda x: x[1][1] > 20 )`

This line filters the `avg3` RDD to only include `product_id`s that have been reviewed by more than 20 users.

`avg3 = avg3.map(lambda x: ((x[1][0]+x[1][1],), x[0])).sortByKey(ascending=False)`

This line maps the `avg3` RDD to swap the key and value, with the new key being the sum of the count and value, and the value being the `product_id`. It then sorts the RDD by the new key in descending order.

```
[ ]: #Suggestion_users = reviews.filter(lambda entry: entry['user_id'])
#for review in Suggestion_users.collect():
r1 = reviews.map(lambda r: ((r['product_id'],), 1))
avg3 = r1.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0],␣
 ↪x[1] + y[1]))
avg3 = avg3.filter(lambda x: x[1][1] > 20 )
avg3 = avg3.map(lambda x: ((x[1][0]+x[1][1],), x[0])).sortByKey(ascending=False)

for movie in avg3.take(10):
    print ("http://www.amazon.com/dp/" + movie[1][0] + " WATCHED BY : " +␣
 ↪str(movie[0][0]) + " PEOPLE")
```

`r2 = reviews.map(lambda ru: ((ru['user_id'],), 1))`

Creates an RDD `r2` by mapping each review `ru` to a tuple containing the user ID as a single-element tuple and the value 1.

`avg2 = r2.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0], x[1]+ y[1]))`

Transforms `r2` by adding a count of 1 to each value, then reduces by key, summing the values and counts for each user ID.

`avg2 = avg2.filter(lambda x: x[1][1] > 20 )`

Filters avg2 to include only user IDs with a count greater than 20.

`avg2 = avg2.map(lambda x: ((x[1][0]+x[1][1],), x[0])).sortByKey(ascending=False)`

Transforms avg2 by adding the value and count for each user ID, then sorts the resulting RDD by the summed value in descending order.

[ ]:
```
r2 = reviews.map(lambda ru: ((ru['user_id'],), 1))
avg2 = r2.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0],␣
↪x[1]+ y[1]))
avg2 = avg2.filter(lambda x: x[1][1] > 20 )
avg2 = avg2.map(lambda x: ((x[1][0]+x[1][1],), x[0])).sortByKey(ascending=False)

for movie in avg2.take(10):
    print ("http://www.amazon.com/dp/" + movie[1][0] + " WATCHED : " +␣
↪str(movie[0][0]) + " MOVIES")
```

`filtered = reviews.filter(lambda entry: "George" in entry['profile_name'])`

This line filters the reviews dataset to only include entries where the profile_name field contains the string "George".

`print ("Found " + str(filtered.count()) + " entries.\n")`

Prints the number of entries found after filtering.

```
for review in filtered.collect():
    print ("Rating: " + str(review['score']) + " and helpfulness: " + review['helpfulness'])
    print ("http://www.amazon.com/dp/" + review['product_id'])
    print (review['summary'])
    print (review['review'])
    print ("\n")
```

This loop iterates over each review in the filtered dataset and prints:
- The rating score and helpfulness score
- The Amazon product URL
- The review summary

[ ]:
```
# Has someone written a review?
filtered = reviews.filter(lambda entry: "George" in entry['profile_name'])
print ("Found " + str(filtered.count()) + " entries.\n")
for review in filtered.collect():
    print ("Rating: " + str(review['score']) + " and helpfulness: " +␣
↪review['helpfulness'])
    print ("http://www.amazon.com/dp/" + review['product_id'])
    print (review['summary'])
    print (review['review'])
    print ("\n")
```

`reviews_by_movie = reviews.map(lambda r: ((r['product_id'],), r['score']))`

This line groups the reviews by movie ID and associates each movie ID with its corresponding review score.

`avg = reviews_by_movie.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0], x[1]`

This line calculates the average score for each movie by summing up the scores and counts, and then dividing the sum of scores by the sum of counts.

`avg = avg.filter(lambda x: x[1][1] > 20 )`

This line filters out movies that have fewer than 20 reviews, ensuring that the average scores are based on a sufficient number of reviews.

`avg = avg.map(lambda x: ((x[1][0]/x[1][1],), x[0])).sortByKey(ascending=True)`

This line creates a new RDD with the average score as the key and the movie ID as the value, and then sorts the RDD by the average score in ascending order.

[ ]:
```
# Get best and worst rated movies
reviews_by_movie = reviews.map(lambda r: ((r['product_id'],), r['score']))
avg = reviews_by_movie.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y:␣
↪(x[0] + y[0], x[1]+ y[1]))
avg = avg.filter(lambda x: x[1][1] > 20 )
avg = avg.map(lambda x: ((x[1][0]/x[1][1],), x[0])).sortByKey(ascending=True)

for movie in avg.take(10):
    print ("http://www.amazon.com/dp/" + movie[1][0] + " Rating: " +␣
↪str(movie[0][0]))
```

Imports the `datetime` module from the Python standard library, which provides classes for working with dates and times.

`timeseries_rdd = reviews.map(lambda entry: {'score': entry['score'], 'time': datetime.fromtime`

- `reviews` is assumed to be an RDD (Resilient Distributed Dataset) containing review entries.
- The `map` transformation is applied to the `reviews` RDD, which applies the provided lambda function to each entry in the RDD.
- The lambda function `lambda entry: {'score': entry['score'], 'time': datetime.fromtimestamp(entry['time'])}` creates a new dictionary for each entry with two keys:
  - `'score'`: The value of the `'score'` key from the original entry.
  - `'time'`: A `datetime` object created from the `'time'` value of the original entry, which is assumed to be a Unix timestamp. The `datetime.fromtimestamp` function is used to convert the Unix timestamp to a `datetime` object.

[ ]:
```
from datetime import datetime
timeseries_rdd = reviews.map(lambda entry: {'score': entry['score'], 'time':␣
↪datetime.fromtimestamp(entry['time'])})
```

Sample data from an RDD
The code samples 20,000 entries from the `timeseries_rdd` RDD (Resilient Distributed Dataset) without replacement and with a specific seed.

Create a pandas DataFrame
The sampled data is converted into a pandas DataFrame with columns 'score' and 'time'.

Print the first 3 rows of the DataFrame
The `head(3)` method is used to print the first 3 rows of the DataFrame.

Convert the 'score' column to float64 data type
The `astype` method is used to convert the 'score' column to float64 data type.

Set the 'time' column as the index
The `set_index` method is used to set the 'time' column as the index of the DataFrame, with `inplace=True` to modify the DataFrame in place.

Resample and plot the data

[ ]:
```
import pandas as pd
import numpy as np
%matplotlib inline
import matplotlib.pyplot as plt
def parser(x):
    return datetime.strptime('190'+x, '%Y-%m')

sample = timeseries_rdd.sample(withReplacement=False, fraction=20000.0/
 ↪num_entries, seed=1134)
timeseries = pd.DataFrame(sample.collect(), columns=['score', 'time'])

print(timeseries.head(3))
timeseries.score.astype('float64')
timeseries.set_index('time', inplace=True)

Rsample = timeseries.score.resample('Y').count()
Rsample.plot()
Rsample2 = timeseries.score.resample('M').count()
Rsample2.plot()
Rsample3 = timeseries.score.resample('Q').count()
Rsample3.plot()
```

Plot the data

[ ]:
```
for movie in avg.take(4):
plt.bar(movie[1][0],movie[0][0])
plt.title('Histogram of \'AVERAGE RATING OF MOVIE\'')
plt.xlabel('MOVIE')
plt.ylabel('AVGRATING')

for movie in avg2.take(3):
```

```
plt.bar(movie[1][0],movie[0][0])
plt.title('Histogram of \'NUMBER OF MOVIES REVIEWED BY USER\'')
plt.xlabel('USER')
plt.ylabel('MOVIE COUNT')

for movie in avg3.take(4):
plt.bar(movie[1][0],movie[0][0])
plt.title('Histogram of \'MOVIES REVIEWED BY NUMBER OF USERS\'')
plt.xlabel('MOVIE')
plt.ylabel('USER COUNT')
```

Imports the ALS (Alternating Least Squares) class from the pyspark.mllib.recommendation module, which is used for collaborative filtering.

```
def get_hash(s):
    return int(hashlib.sha1(s).hexdigest(), 16) % (10 ** 8)**
```

Defines a function get_hash that takes a string s as input, hashes it using the SHA-1 algorithm, converts the hexadecimal digest to an integer, and returns the integer modulo 10^8.

```
ratings = reviews.map(lambda entry: tuple([ get_hash(entry['user_id'].encode('utf-8')), get_has
```

Maps each entry in the reviews dataset to a tuple containing the hashed user_id, hashed product_id, and the score as an integer.

```
train_data = ratings.filter(lambda entry: ((entry[0]+entry[1]) % 10) >=2 )
test_data = ratings.filter(lambda entry: ((entry[0]+entry[1]) % 10) < 2 )
```

Splits the ratings data into train_data and test_data based on the sum of the hashed user_id and hashed product_id modulo 10. If the result is greater than or equal to 2, the entry is added to train_data, otherwise, it is added to test_data.

```
train_data.cache()
```

Caches the train_data dataset in memory for faster access.

[ ]:
```
from pyspark.mllib.recommendation import ALS
from numpy import array
import hashlib
import math
def get_hash(s):
    return int(hashlib.sha1(s).hexdigest(), 16) % (10 ** 8)

#Input format: [user, product, rating]
ratings = reviews.map(lambda entry: tuple([ get_hash(entry['user_id'].
 ↪encode('utf-8')),
get_hash(entry['product_id'].encode('utf-8')), int(entry['score']) ]))
train_data = ratings.filter(lambda entry: ((entry[0]+entry[1]) % 10) >=2 )
test_data = ratings.filter(lambda entry: ((entry[0]+entry[1]) % 10) < 2 )
train_data.cache()
```

```
#train_data.union(train_data)
print ("Number of train samples: " + str(train_data.count()))
print ("Number of test samples: " + str(test_data.count()))
```

- `rank = 20` sets the rank of the factorized user-item matrices to 20
- `numIterations = 20` sets the number of iterations for the ALS algorithm to 20
- `model = ALS.train(train_data, rank, numIterations)` trains the ALS recommendation model on the `train_data` with the specified rank and number of iterations

Helper function to convert strings to floats
- `convertToFloat(lines)` takes a list of strings `lines` and converts each element to a float, returning a new list with the float values

[ ]:
```
# Build the recommendation model using Alternating Least Squares
from math import sqrt
rank = 20
numIterations = 20
model = ALS.train(train_data, rank, numIterations)

def convertToFloat(lines):
    returnedLine = []
    for x in lines:
        returnedLine.append(float(x))
    return returnedLine

# Evaluate the model on test data
unknown = test_data.map(lambda entry: (int(entry[0]), int(entry[1])))
predictions = model.predictAll(unknown).map(lambda r: ((int(r[0]), int(r[1])), 
 ↪r[2]))
true_and_predictions = test_data.map(lambda r: ((int(r[0]), int(r[1])), r[2])).
 ↪join(predictions)
MSE = true_and_predictions.map(lambda r: (int(r[1][0]) - int(r[1][1])**2).
 ↪reduce(lambda x, y: x + y)/true_and_predictions.count())
true_and_predictions.take(10)
```

This line sets a minimum occurrence threshold for words to be considered in the analysis.

good_reviews = reviews.filter(lambda line: line['score']==5.0)
This line filters the reviews dataset to only include reviews with a score of 5.0 (presumably positive reviews).

bad_reviews = reviews.filter(lambda line: line['score']==1.0)
This line filters the reviews dataset to only include reviews with a score of 1.0 (presumably negative reviews).

```
good_words = good_reviews.flatMap(lambda line: line['review'].split(' '))
num_good_words = good_words.count()
```

```
good_words = good_words.map(lambda word: (word.strip(), 1)).reduceByKey(lambda a, b: a+b).filte
```

These lines extract the words from the positive reviews, count the total number of words, and then count the occurrences of each word, keeping only those that occur more than the minimum occurrence threshold.

```
bad_words = bad_reviews.flatMap(lambda line: line['review'].split(' '))
num_bad_words = bad_words.count()
bad_words = bad_words.map(lambda word: (word.strip(), 1)).reduceByKey(lambda a, b: a+b).filter
```

These lines perform the same operations as above, but for the negative reviews.

```
frequency_good = good_words.map(lambda word: ((word[0],), float(word[1])/num_good_words))
frequency_bad = bad_words.map(lambda word: ((word[0],), float(word[1])/num_bad_words))
```

These lines calculate the frequency of each word in the positive and negative reviews, respectively.

[ ]:
```
min_occurrences = 10
good_reviews = reviews.filter(lambda line: line['score']==5.0)
bad_reviews = reviews.filter(lambda line: line['score']==1.0)

good_words = good_reviews.flatMap(lambda line: line['review'].split(' '))
num_good_words = good_words.count()
good_words = good_words.map(lambda word: (word.strip(), 1)).reduceByKey(lambda␣
  ↪a, b: a+b).filter(lambda word_count: word_count[1] > min_occurrences)
bad_words = bad_reviews.flatMap(lambda line: line['review'].split(' '))
num_bad_words = bad_words.count()
bad_words = bad_words.map(lambda word: (word.strip(), 1)).reduceByKey(lambda a,␣
  ↪b: a+b).filter(lambda word_count: word_count[1] > min_occurrences)

frequency_good = good_words.map(lambda word: ((word[0],), float(word[1])/
  ↪num_good_words))
frequency_bad = bad_words.map(lambda word: ((word[0],), float(word[1])/
  ↪num_bad_words))

joined_frequencies = frequency_good.join(frequency_bad)
```

```
def relative_difference(a, b):
    return math.fabs(a-b)/a
```

Defines a function `relative_difference` that calculates the relative difference between two numbers `a` and `b`. It uses the `math.fabs` function to get the absolute value of the difference between `a` and `b`, and then divides it by `a`.

```
result = joined_frequencies.map(lambda f: ((relative_difference(f[1][0],f[1][1]),), f[0][0]) )
```

Applies a map transformation to the `joined_frequencies` dataset. For each element `f` in the dataset, it calculates the relative difference between the two values in `f[1]` (presumably frequencies) using the `relative_difference` function. It then creates a tuple with the relative difference as the first element and the key `f[0][0]` as the second element. The resulting tuples are sorted in descending order based on the relative difference.

[ ]:
```
import math
def relative_difference(a, b):
    return math.fabs(a-b)/a
result = joined_frequencies.map(lambda f:␣
 ↪((relative_difference(f[1][0],f[1][1]),), f[0][0]) ).
 ↪sortByKey(ascending=False)
result.take(50)

for movie in result.take(7):
    plt.bar(movie[1],movie[0][0])
    plt.title('Histogram of \'SENTIMENT ANALYSIS\'')
    plt.xlabel('WORD')
    plt.ylabel('NUMBER OF OCCURANCES')
```