Movie Recommendation

- Import the necessary libraries.
- Create a Spark context, which represents a connection to a spark cluster, used to create RDDs. The spark context is created on all the CPUs available on the system (local[*])

In [ ]:
```
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[*]')
#sqlContext = SQLContext(sc)
    print("Just created a SparkContext")
except ValueError:
    warnings.warn("SparkContext already exists in this scope")
```

Reviews are read into an RDD using the .textFile() function which reads the data from the given file path The lists define the fields required in the Dataframe. A validate function is used to check if all fields are present in the line. For each line in the RDD, it is converted into a JSON format, and it is filtered using the validate function, that is, only the lines which return true on the validate function (contain all fields specified in field list) are retained. This RDD is then cached to store it in the cluster node's memory so it can be reused for subsequent actions. This avoids performing repeated computations.

In [ ]:
```
 import json
fields = ['product_id',
'user_id',
'score',
'time']
fields2 = ['product_id',
'user_id',
'review',
35
'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()
```

Data Exploration

Display the first row in the created RDD

In [ ]:
```
reviews.take(1)
```

Calculate and display certain statistics for the reviews:

- The number of movies is calculated by counting the number of distinct product IDs
- The number of distinct users is calculated by counting the number of distinct User IDs
- The total number of reviews is the count of records in the RDD

In [ ]:
```
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(nu
```

- Using the map function, each record in the review table is converted to the form (product_id of record, 1)
- the reduceByKey function aggregates the data from all partitions, calculating the sum of reviews for each movie (distinct product ID)
- This is filtered to only retain movies with greater than 20 reviews
- It is mapped to the form (count, product_id), and sorted by key to store all the movies in descending order of the count

In [ ]:
```
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)
```

Display the movie link and the number of reviews for it, by taking the first 10 records of the RDD

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

- Using the map function, each record in the review table is converted to the form (user_id of record, 1)
- the reduceByKey function aggregates the data from all partitions, calculating the sum of reviews from each user (distinct user ID)
- This is filtered to only retain users with greater than 20 reviews
- It is mapped to the form (count, user_id), and sorted by key to store all the users in descending order of the number of reviews they have made

In [ ]:
```
 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 )
```

Display the user ID and the number of movies they watched by taking the first 10 records of the RDD

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

Filter the reviews to retain only those given by users whose profile name contains 'George'. Display the number of such reviews, and display each of them in sequence.

In [ ]:
```
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['helpfulne
print ("http://www.amazon.com/dp/" + review['product_id'])
print (review['summary'])
print (review['review'])
print ("\n")
```

- Using the map function, each record in the review table is converted to the form (product_id, score)
- This is further mapped to the form ((product_id, score), 1)
- the reduceByKey function aggregates the data from all partitions, calculating the sum of scores and the number of reviews for each movie (distinct product ID)
- This is filtered to only retain movies with greater than 20 reviews
- It is mapped to the form (average score, product_id), and sorted by key to store all the movies in descending order of the average score (average score is calculated by dividing sum of scores by number of reviews)

In [ ]:
```
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)
```

Display the movie link and its average score, by taking the first 10 records of the RDD

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

Create an RDD consisting of scores along with the time they were entered, by extracting the 'score' and 'time' attribute of each entry in the reviews table. The .totimestamp method is used to convert the timestamp into a time value.

In [ ]:
```
from datetime import datetime
timeseries_rdd = reviews.map(lambda entry: {'score': entry['score'],'time': datetim
```

- A random smaple of 20000 entries is taken from the Timeseries RDD created previously
- This is converted to a Pandas dataframe, and the first 3 rows are displayed

- The resample method of Pandas creates a unique sampling distributuion based on the specified rule. In the code, the score is resampled to provide the count of the scores taken:
  - Yearly (resample('Y'))
  - Monthly (resmaple('M'))
  - Quarterly (resmaple('Q'))

These are plotted, show the trends of the counts of the reviews as they are sampled for different frequencies

In [ ]:
```
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,
timeseries = pd.DataFrame(sample.collect(),columns=['score', 'time'])
print(timeseries.head(3))
timeseries.score.astype('float64')
#timeseries.time.astype('datetime64')
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()
```

Using the previously created aggregate RDDS, bar plots are created of:

- the average rating of 4 movies
- the number of reviews made by 3 users
- the number of users thats reviewed 4 movies

In [ ]:
```
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')
```

In [ ]:
```
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')
```

In [ ]:
```
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')
```

Recommendation system using Matrix Factorization

- A hash function is defined to convert strings into numeric representations, using the hashlib library
- A ratings RDD is created by extracting the user_id and product_id, converted to numeric using the hash function and the score from each review.
- This is distributed into train and test sets based on the Modulo 10 value of the sum of the numeric values of the user_id and product_id. 80% of the data is distributed to the train set, and the remaining 20% in the test set. The train data is cached to persist it in memory so it can be reused for subsequent actions.

In [ ]:
```
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

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()

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

- The ALS module is imported from the pyspark.mllib.recommendation library. This uses the Alternating Least Square algorithm to make recommendations:
  - It decomposes a user-product interaction matrix into two smaller dimension matrices which contain latent information about user preferences and product attributes.
  - In this case the model is made by running the ALS algorithm for 20 iterations, with a rank of 20, indicating the number of latent features it tries to discover (dimension of the decomposed matrices)
- The test data is then mapped to extract only the user_id and product_id, removing the scores. Predictions are then made for each entry, using the predictAll function of the trained model.
- These predictions are mapped to the form ((user_id, product_id), prediction), so that the key for each record is the combination of user and product ID
- This is joined on the key of distinct user and product ID combinations to the test data, so each record also containd the predicted score.
- The mean square error is then calculated to evaluate the accuracy of the predictions. MSE is calculated as the mean value of the sum of squared diffrence between the true value and the prediction for each record.

In [ ]:
```
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
MSE = true_and_predictions.map(lambda r: (int(r[1][0]) - int(r[1][1])**2).reduce(la
```

The first 10 true values and predictions of score are displayed from the RDD

In [ ]:
```
true_and_predictions.take(10)
```

Sentiment Analysis

- The reviews are filtered into good and bad based on their score, with good reviews being those with a score of 5, and bad reviews with a score of 1
- A list of good words is created by creating a flatMap (flattened RDD) of all the words in each review. Similarly, a RDD of bad words is created
- For each word in the RDD of good words, it is mapped to the form (word, 1) and reduced by the key (the word) to calculate the count of each word in the RDD. Finally it is filtered to retain only those words which occur more than a minimum frequency. The same is done for the RDD of bad words.
- The number of good words and bad words is stored as the count of all the words in the respective flattened RDDs

In [ ]:
```
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)
```

The frequency of each good and bad word is calculated by dividing its count by the total number of good and bad words respectively

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

- The word frequesncies of the good and bad reviews are joined
- The relative difference of each word frequency in the good and bad reviews are calculated
- This is sorted in descending order of relative difference to get the most significant expressions which characterise a movie as either good or bad
- The first 50 words with their relative frequency in good and bad reviews is displayed

In [ ]:
```
joined_frequencies = frequency_good.join(frequency_bad)

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]),)
    .sortByKey(ascending=False)

 result.take(50)
```

A bar plot is made of the relative difference of frequency (number of occurences) in good and bad reviews for 7 words

In [ ]:
```
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')
```