A Movie Recommendation Service

Source: https://www.codementor.io/@jadianes/building-a-recommender-with-apache-spark-python-example-app-part1-du1083qbw

Getting and processing the data

Create a SparkContext configured for local mode

In [10]:
```python
import pyspark
sc = pyspark.SparkContext('local[*]')
```

File download

Small: 100,000 ratings and 3,600 tag applications applied to 9,000 movies by 600 users. Last updated 9/2018.
Full: 27,000,000 ratings and 1,100,000 tag applications applied to 58,000 movies by 280,000 users. Includes tag genome data with 14 million relevance scores across 1,100 tags. Last updated 9/2018.

In [108…
```python
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'
```

Download locations

In [109…
```python
import os

datasets_path = os.path.join('/home/jovyan', 'work')

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

Getting files

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.

In [110…
```python
import urllib.request
# Commenting out given code to adjust for correct code
#small_f = urllib.urlretrieve (small_dataset_url, small_dataset_path)
#complete_f = urllib.urlretrieve (complete_dataset_url, complete_dataset_path)

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

Extracting files

In [111…
```python
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

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.

ratings.csv

In [112…
```python
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]

#parse
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()

print ('There are {} recommendations in the small dataset'.format(small_ratings_data.count()))
small_ratings_data.take(3)
```

```
There are 100836 recommendations in the small dataset
```

Out[112…
```
[('1', '1', '4.0'), ('1', '3', '4.0'), ('1', '6', '4.0')]
```

movies.csv

In [113…
```python
# Load the small dataset 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]

# Parse
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()

print ('There are {} movies in the small dataset'.format(small_movies_data.count()))
small_movies_data.take(3)
```

```
There are 9742 movies in the small dataset
```

Out[113…
```
[('1', 'Toy Story (1995)'),
 ('2', 'Jumanji (1995)'),
 ('3', 'Grumpier Old Men (1995)')]
```

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.

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.

In [114…
```python
# In source code users will see "0L", which is from the previous version of python (2.x)
# 0L should be written as 0 from now on
training_RDD, validation_RDD, test_RDD = small_ratings_data.randomSplit([6, 2, 2], seed=0)
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]))
```

Training phase

In [115…
```python
from pyspark.mllib.recommendation import ALS
import math

seed = 5
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 {} the RMSE is {}'.format(rank, error))
    if error < min_error:
        min_error = error
        best_rank = rank

print ('The best model was trained with rank {}'.format(best_rank))
```

```
For rank 4 the RMSE is 0.908078105265682
For rank 8 the RMSE is 0.916462973348527
For rank 12 the RMSE is 0.917665030756129
The best model was trained with rank 4
```

In [116…
```python
predictions.take(3)
```

Out[116…
```
[((372, 1084), 3.42419871162954),
 ((4, 1084), 3.866749726695713),
 ((402, 1084), 3.4099577968422152)]
```

In [117…
```python
rates_and_preds.take(3)
```

Out[117…
```
[((1, 457), (5.0, 4.381060760461434)),
 ((1, 1025), (5.0, 4.705295366590298)),
 ((1, 1089), (5.0, 4.979982471805129))]
```

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.

In [118…
```python
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 {}'.format(error))
```

```
For testing data the RMSE is 0.9113780946334407
```

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.

In [119…
```python
# 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]))).cac

print("There are {} recommendations in the complete dataset".format(complete_ratings_data.count()))
```

```
There are 27753444 recommendations in the complete dataset
```

Now we are ready to train the recommender model

In [120…
```python
training_RDD, test_RDD = complete_ratings_data.randomSplit([7, 3], seed=0)

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

Now we test on our testing set.

In [121…
```python
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 {}'.format(error))
```

```
For testing data the RMSE is 0.8318265262101795
```

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.

In [122…
```python
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 {} movies in the complete dataset".format(complete_movies_titles.count()))
```

```
There are 58098 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.

In [123…
```python
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).

In [133…
```python
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)
#    ]

# User One Rating
# new_user_ratings = [
#    (0, 51662, 5), # 300 (2007)
#    (0, 1732, 5), # Big Lebowski, The (1998)
#    (0, 1485, 5), # Liar Liar (1997)
#    (0, 2329, 4), # American History X (1998)
#    (0, 48516, 5), # Departed, The (2006)
#    (0, 183959, 4), # Tom Segura: Disgraceful (2018)
#    (0, 4993, 5), # Lord of the Rings: The Fellowship of the Ring, The (2001)
#    (0, 103688, 4), # Conjuring, The (2013)
#    (0, 56174, 3), # I Am Legend (2007)
#    (0, 183869, 4) # Hereditary (2018)
# ]

#User Two Ratings
new_user_ratings = [
    (0, 1801, 4), # Man in the Iron Mask, The (1998)
    (0, 7143, 3), # Last Samurai, The (2003)
    (0, 183869, 4), # Hereditary (2018)
    (0, 8961, 4), # Incredibles, The (2004)
    (0, 34162, 3), # Wedding Crashers (2005)
    (0, 192411, 1), # Freaky Friday (2018)
    (0, 192849, 5), # Bert Kreischer: Secret Time (2018)
    (0, 7451, 4), # Mean Girls (2004)
    (0, 8957, 4), # Saw (2004)
    (0, 58998, 5) # Forgetting Sarah Marshall (2008)
]

new_user_ratings_RDD = sc.parallelize(new_user_ratings)
print ('New user ratings: {}'.format(new_user_ratings_RDD.take(10)))
```

```
New user ratings: [(0, 1801, 4), (0, 7143, 3), (0, 183869, 4), (0, 8961, 4), (0, 34162, 3), (0, 192411, 1), (0,
192849, 5), (0, 7451, 4), (0, 8957, 4), (0, 58998, 5)]
```

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

In [134…
```python
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).

In [135…
```python
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 {} seconds'.format(round(tt,3)))
```

```
New model trained in 122.748 seconds
```

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.

In [136…
```python
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(lamb

# Use the input RDD, new_user_unrated_movies_RDD, with new_ratings_model.predictAll() to predict new ratings fo
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.

In [137…
```python
# 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)
```

Out[137…
```
[(6216,
  ((4.272719522674002, 'Nowhere in Africa (Nirgendwo in Afrika) (2001)'),
   717)),
 (124320, ((4.275840820533761, 'Once a Thief (1965)'), 1)),
 (83916, ((3.8816016300547886, 'Blues in the Night (1941)'), 9))]
```

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

In [138…
```python
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.

In [140…
```python
top_movies = new_user_recommendations_rating_title_and_count_RDD.filter(lambda r: r[2]>=25).takeOrdered(15, key

print ('TOP 15 recommended movies (with more than 25 reviews):\n{}'.format('\n'.join(map(str, top_movies))))
```

```
TOP 15 recommended movies (with more than 25 reviews):
('Elway To Marino (2013)', 5.257835236472239, 25)
('Connections (1978)', 5.202213372934146, 49)
('Cosmos', 5.177670126907799, 157)
('Rabbit of Seville (1950)', 5.163484150616505, 30)
('Baseball (1994)', 5.159287547672342, 42)
('"Last Lions', 5.12409635143586, 38)
('Harakiri (Seppuku) (1962)', 5.113893949692688, 679)
("Jim Henson's The Storyteller (1989)", 5.1133157364957675, 36)
("Who's Singin' Over There? (a.k.a. Who Sings Over There) (Ko to tamo peva) (1980)", 5.104265150473019, 45)
('"Century of the Self', 5.1040310208256585, 213)
('"Human Condition III', 5.098909786081757, 91)
('"In the blue sea', 5.085521273384728, 37)
('Planet Earth (2006)', 5.075628923151591, 1384)
('Planet Earth II (2016)', 5.073128889749725, 853)
("World of Tomorrow Episode Two: The Burden of Other People's Thoughts (2017)", 5.072479932851294, 39)
```

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.

In [131…
```python
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)
```

Out[131…
```
[Rating(user=0, product=116688, rating=1.3894139069577962)]
```