Movie Recommendation by Collaborative Filtering

Carl Chang

Introduction

In this project, we applied item-item-based Collaborative Filtering to predict the ratings for movies that users didn't rate. First, we calculate the similarity between movies by using cosine similarity with subtracting mean (Pearson correlation coefficient), then predict the rating of each movie by calculating the weighted average of the movies that are similar to it.

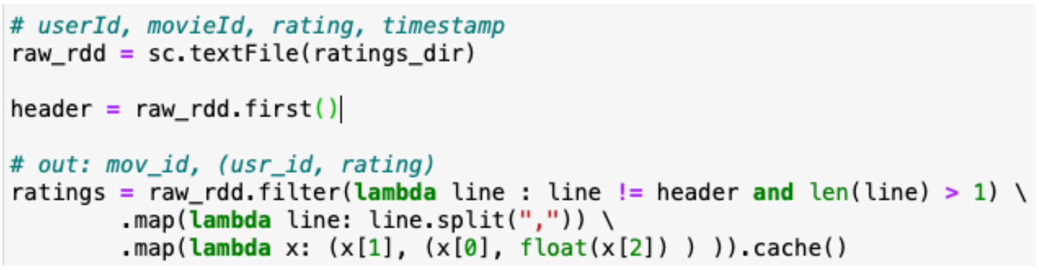
Dataset

MovieLens Latest Dataset Small (ml-latest-small). It contains 610 users and their ratings across 9742 movies and was collected between March 29, 1996, and September 24, 2018. The user ratings used in this project are in the format of (userID, movieID, rating, timestamp).

Implementation

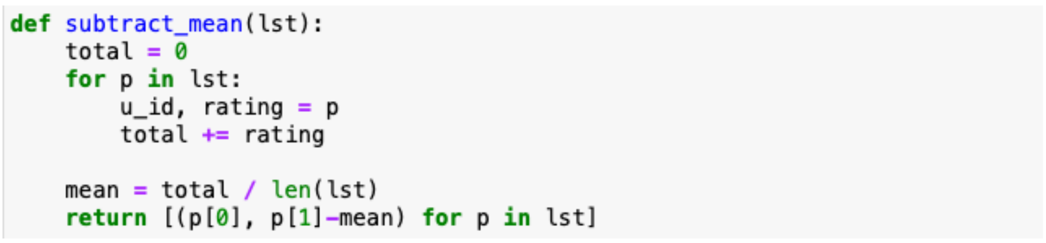
Part I. Item-item similarity

First we read the input file “ratings.csv” to a RDD and parse it to get (mov_id, (usr_id, rating)) pairs.



```python
# userId, movieId, rating, timestamp
raw_rdd = sc.textFile(ratings_dir)

header = raw_rdd.first()

# out: mov_id, (usr_id, rating)
ratings = raw_rdd.filter(lambda line : line != header and len(line) > 1) \
        .map(lambda line: line.split(",")) \
        .map(lambda x: (x[1], (x[0], float(x[2]) ) )).cache()
```

Secondly, we calculate the mean rating of each movie. We can use .groupByKey( ) to group all the (usr_id, rating) with the same mov_id into a list and calculate the mean value, then subtract the rating with the mean value; these steps can be done in a mapper function. Noted that we can also calculate the mean using .reduceByKey() or aggregateByKey(), which are more efficient than .groupByKey( ). Still, we need to .join() the mean values back to the mov_id pair and use a mapper function to subtract the mean value. Therefore, for simplicity and readability, we use .groupByKey() instead.



```python
def subtract_mean(lst):
    total = 0
    for p in lst:
        u_id, rating = p
        total += rating

    mean = total / len(lst)
    return [(p[0], p[1]-mean) for p in lst]
```

After subtracting the mean, we can compute the similarity between each movie. The similarity is calculated according to the following formula. We have subtracted the mean of each movie in the last step. The remaining work is trivial.

$$sim(x, y) = \frac{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)(r_{ys} - \bar{r}_y)}{\sqrt{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)^2} \sqrt{\sum_{s \in S_{xy}} (r_{xs} - \bar{r}_x)^2}}$$

S_{xy} = movies rated by both user x and user y

In practice, we take the cross product of two (mov_id, [list of (usr_id, rating), …) pairs and compute the similarity in a mapper function *calc_sim*. The trick here is to transform a list of pairs directly to a Python Dict data type and check whether a movie was rated by both users in constant time.

```python
def calc_sim(x):
    p1_id, p1_lst = x[0]
    p2_id, p2_lst = x[1]

    if p1_id == p2_id:
        return (p1_id, p2_id), 1.0

    d1, d2 = dict(p1_lst), dict(p2_lst)

    sum_of_prod = 0
    sum_of_sqr1, sum_of_sqr2 = 0,0

    for k in d1.keys():
        sum_of_sqr1 += d1[k]**2
        if k in d2.keys():
            sum_of_prod += d1[k] * d2[k]

    for k in d2.keys():
        sum_of_sqr2 += d2[k]**2

    if sum_of_prod == 0:
        sim = 0
    else:
        sim = sum_of_prod / (math.sqrt(sum_of_sqr1) * math.sqrt(sum_of_sqr2))

    return (p1_id, p2_id), sim
```

After calculate the similarity between movies, the mapper function *calc_sim* returns key-value pairs as ((mov_id, mov_id), similarity).

Part II. Rating prediction

To predict rating values for unrated movies, first, we map the ratings RDD and group it by user_id, so we get an RDD *user* that contains (usr_id, [(mov_id, rating), (mov_id, rating), …]) pairs. Next, we take the cross-product of *user* and *item* (a set consists of all the distinct items). The cross-product (obtained from .cartesian( )) is the combination of all users and all movies, and we can use this to predict the ratings in a new mapper function.

```python
def new_mapper(x):
    u_id, u_rating_lst = x[0]
    m_id = x[1]

    u_rating_dic = dict(u_rating_lst)

    if m_id not in u_rating_dic.keys():

        return [(m_id, (u_id, u_rating_lst))]
    else:
        return []
```

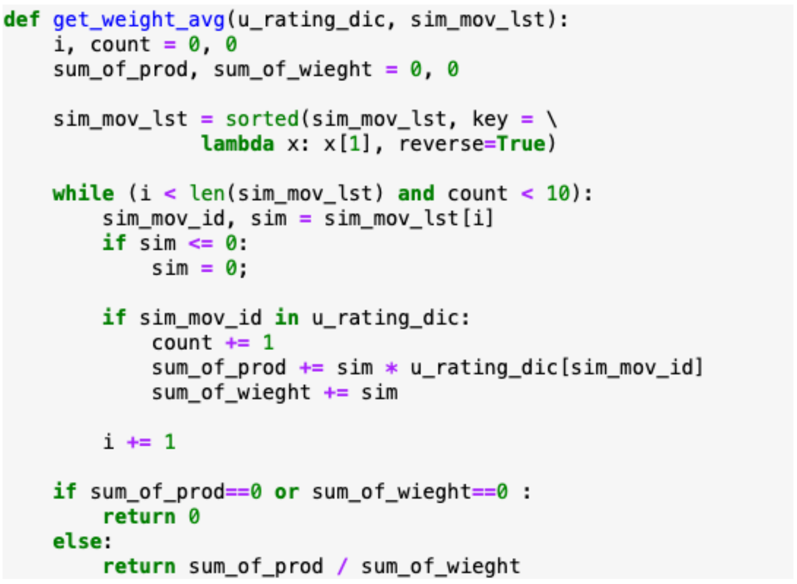
As the screenshot above shows, if a mov_id wasn't in the rated list of a user_id, it means that the user hadn't rated the movie yet, so we keep it for the prediction step. Otherwise, we return an empty list since we don't need the rated user-movie pairs in further process. The empty list can be eliminated after running .flatMap().

The prediction strategy we use in this project is taking the top ten similar movies and calculating the weighted average. The weights are the similarity we calculated in Part I.

$$r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}$$

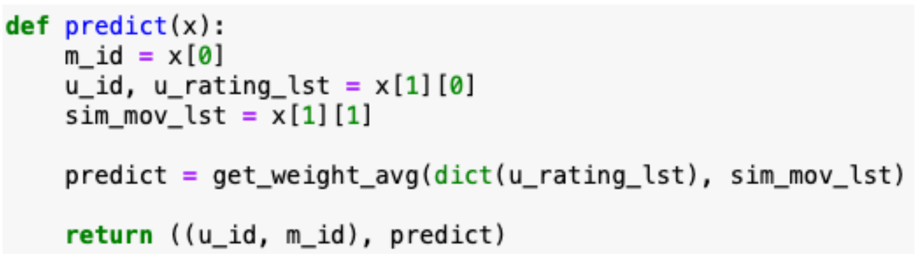
r_{xi} : The (predict) rating of user x on movie i ; s_{ij}: similarity of movie i and j

If the similar movie wasn't rate by the user, we pass it and look for the next similar movie, and set the weight to zero if that movie has a negative similarity



```python
def get_weight_avg(u_rating_dic, sim_mov_lst):
    i, count = 0, 0
    sum_of_prod, sum_of_wieght = 0, 0

    sim_mov_lst = sorted(sim_mov_lst, key = \
                lambda x: x[1], reverse=True)

    while (i < len(sim_mov_lst) and count < 10):
        sim_mov_id, sim = sim_mov_lst[i]
        if sim <= 0:
            sim = 0;

        if sim_mov_id in u_rating_dic:
            count += 1
            sum_of_prod += sim * u_rating_dic[sim_mov_id]
            sum_of_wieght += sim

        i += 1

    if sum_of_prod==0 or sum_of_wieght==0 :
        return 0
    else:
        return sum_of_prod / sum_of_wieght
```

After getting all the new user-movie pairs with predicted rating, the remaining part is to rearrange the pairs and sort it by the keys for the final output, and these can be done using .sortByKey().



```python
def predict(x):
    m_id = x[0]
    u_id, u_rating_lst = x[1][0]
    sim_mov_lst = x[1][1]

    predict = get_weight_avg(dict(u_rating_lst), sim_mov_lst)

    return ((u_id, m_id), predict)
```

Challenges

The major challenge in this project is the size of the data. Though the dataset we used is a small subset of MovieLens Dataset, it is still quite a burden to run it on a PC. For example, when computing the item-item similarity in Part I., there are C(9742, 2) = 47448411 pairs in total, moreover, in Part II. we have to join them with user-movie pairs to predict the ratings. Hence, although the output from our PySpark program for a tiny test data is correct, when running it on the whole dataset, it takes a lot of time and sometimes pops out some errors such as: “OSError: [Errno 23] Too many open files in system:”

After surfing the web and rewriting my program, I was able to run my program on Jupyter Notebook and write output to files. The following are some steps I made:

- Adjust the parameters in Spark.Conf() such as “spark.default.parallelism”, “spark.driver.memory”, “spark.driver.maxResultSize”.
- Increase number of file descriptors by “ulimit -S -n {number of files}”
- Decrease the usage of sortByKey() to the whole data, since it is time-consuming. When finding top ten similar movies for a mov_id, we can sort the list of a mov_id instead of using .sortByKey() to the whole data then group them by mov_id.
- Decrease the usage of .filter(), since it creates a new filtered RDD. Instead, we can return empty list in a mapper function and use .flatMap().
- Use .cache(), which creates a checkpoint for the RDD that can be reused later.
- Use .unpersist() to discard unused RDD.

Discussion

The algorithm we used in the project is simple to implement but has some defects.
For instances:

1. If all users who rated a movie leave the same rating value, this movie's ratings will all become zero after subtracting the mean. Consequently, it will have zero similarity with all the other movies since the dot product will be zero.

2. We take the weighted averages of an unrated movie's top ten similar movies and predict the rating. However, the movies similar to the unrated movie may also be unrated, so they will have no contribution to the final weighted average even though this movie has high similarities. One possible solution is to take the average rating of this similar movie if it is also unrated by the user.