Term Project

Group Number: 45

106062322 江岷錡

本次實作 Recommendation 的 Calculate Similarity (Basic Part) + Calculate Prediction (Bonus Part)，Dataset 的部分使用助教提供的 MovieLens Dataset，以下將分別講解實作：

Calculate Similarity (Basic)

讀取資料

在開始前，我執行一個 `readData` Mapper ，將資料整理成 `(movie_id, (user_id, rating))` 格式後，輸出成 `originalData` RDD。

Subtract Min - Normalization

當資料已整理成 `movie_id` 與 `user_id` 的 Mapping 時，即可算 cosine similarity 。根據 Spec ，在計算 Cosine Similarity 前，需先將資料進行 subtract min 的 normalization 。

因此，我先將 `(movie_id, (user_id, rating))` 格式，轉成 `(movie_id, (1, rating))` 。

```
movieAvgs = originalData
            .map(lambda data: (data[0], (1, data[1][1])))
```

接著，將每個 `(movie_id, (1, rating))` reduce 相加後，value 前半部便是 `該 movie 被多少人 rating` 、value 後半部便是 `每個人 rating 的相加` 。

再將 `每個人 rating 的相加 / movie 被多少人 rating` ，即可得出任一 movie 的 Average 。

```
            ...
            .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
            .map(lambda data: (data[0], data[1][1]/data[1][0]))
```

將 `originalData` 與 `movieAvgs` join 後，即將每個 rating 減去 average rating ，並輸出成 `(movie_id, [(user_id1, modified_rating1), (user_id2, modified_rating2), ...])` 。

```
joinedData =
originalData
.join(movieAvgs)
.map(lambda data: (data[0], [(data[1][0][0], (data[1][0][1] - data[1][1]))]))
.reduceByKey(lambda x, y: x+y)
```

Cosine Similarity - Calculate the length (||Rx||,||Ry||)

當 `normalized (subtract avg.)` 後，及可得到以 `movie_id` 為 key 的 user rating 集合。接續，便開始計算每個電影的長度。

$$||r_x||(r:movie)$$

我 for-loop `[(user_id1, modified_rating1), (user_id2, modified_rating2), ...])` 此 Array，將每個 `modified_rating` 進行平方，再進行相加，最終再開根號。完成操作後，輸出成 `(movie_id, ||movie||)` 。

```
# Output: (movie_id, consineBottom)
def calConsineBottom(data):
    ratingSum = 0
    for userRatingTriple in data[1]:
        userRating = userRatingTriple[1]
        ratingSum = ratingSum + pow(userRating, 2)
    squaredRatingSum = math.sqrt(ratingSum)
    return (data[0], squaredRatingSum)
```

Cosine Similarity - Calculate the Dot Product (Rx * Ry)

計算完每個 movie 的長度後，便需要計算兩兩 movie 的相乘長度。

$$r_x . r_y$$

觀察測資資料後，我們發現 movie 間的 user rating 分佈，都相差極大。若以 `movie_id` 為 base ，去計算 movie 間的 similarity ，反而會花費空間存無效資料。因此，我以 `user_id` 為 base，調出任一 user 的 movie rating list ，再將 movie 及 movie 兩兩 pair 。

也因此，我先整理出以 `user_id` 為 key 的 movie list 。透過 `convertedToUserKey` Mapper，我將資料轉成 `(user_id, [(movie_id1, user_rating1), (movie_id2, user_rating2), ...])` 的格式。

```
def convertedToUserKey(data):
    expandedData = []
    movieId = data[0]
    for userRatingTriple in data[1]:
        userId = userRatingTriple[0]
        userRating = userRatingTriple[1]
        expandedData.append((userId, [(movieId, userRating)]))
    return expandedData

convertedToUserKeyData
= joinedData.flatMap(convertedToUserKey).reduceByKey(lambda x, y: x + y)
```

當我們有 `(user_id, [(movie_id1, user_rating1), (movie_id2, user_rating2), ...])` 的資料格式後，便能針對 `movie` 與 `movie` 進行 pairing 。

透過雙層 for-loop，我遍歷 `[(movie_id1, user_rating1), (movie_id2, user_rating2), ...]` 後進行兩兩 paring，便可得到 `((movie_id1, movie_id2), some_rating)` 的資料格式。在 pairing 時，我確保 movie_id 較小的會放前面， movie_id 較大的會放後方，以方便後續比對。

而因為每個 user ，都會得出一串 movie & movie pairing，最終還需透過 reducer 將 value 相加。

```
def moviePairMapping(data):
    exportedData = []
    for OuterMovieRatingPair in data[1]:
        for innerMovieRatingPair in data[1]:
            firstMovieId = OuterMovieRatingPair[0]
            secondMovieId = innerMovieRatingPair[0]
            if (firstMovieId >= secondMovieId): continue

            firstMovieUserRating = OuterMovieRatingPair[1]
            secondMovieUserRating = innerMovieRatingPair[1]
```

```
            productRating = firstMovieUserRating*secondMovieUserRating

            exportedData.append(((firstMovieId, secondMovieId), productRating))

    return exportedData

consineTop
= convertedToUserKeyData.flatMap(moviePairMapping).reduceByKey(lambda x, y: x +
 y)
```

Cosine Similarity - Dot Product / ||Rx||*|Ry||

目前的 `consineTop` RDD，已經有兩兩 movie 的 dot product 結果。由於在 `cosine similarity` 計算時，dot product 會放在分子。也因此，當 dot product 為 0 時，便會算出 `0` similarity ，故在此步我先過濾掉 dot product 為 0 的 pairing 。

```
consineTop = consineTop.filter(lambda data: data[1] != 0)
```

而資料整理後，便開始與 movie 各自的長度(`||rx||`) 做 pairing 。由於前面算出的 cosine similarity ，是以 `movie_id` 為 key 的 RDD ，為求方便 Mapping，我將剛算出的 `consineTop` RDD ，各自提出 `movie_id1` 及 `movie_id2` 後生成新 RDD ，再與 movie 各自的長度(`||rx||`) 進行 mapping 。

```
coleasedConsine
= consineTop.flatMap(lambda data: [(data[0][0], data), (data[0][1], data)]).join
(consineBottom)
```

接著，再將 RDD 整理成 `((sim_pair), ||rx||), ((sim_pair), ||ry||)` 後，進行 Reducer 相乘。（即是 `||rx||*||ry||`）

```
finalResult
= coleasedConsine.map(lambda data: (data[1][0], data[1][1])).reduceByKey(lambda
 x, y: x*y)
```

相乘後，目前的資料格式為 `(((sim_movie_id1, sim_movie_id2), dotProduct), ||rx||*||ry||)` 。我將上述整理成 `(((sim_movie_id1, sim_movie_id2),`

`dotProduct/(||rx||*||ry||))` ，得到 `(movie_id1, movie_id2), similarity` 的資料格式，完成 Basic 的作業要求。

```
finalResult = finalResult.map(lambda data: (data[0][0], data[0][1]/data[1]))
```

最終再將結果 Output 至 `basic_output.txt` File 。

Calculate Prediction (Bonus Part)

讀取資料

完成 Basic Part 後，已將結果儲存成 txt file 。重新從檔案讀出後，我將資料存至 `bonusData` RDD。而根據 Spec ，當某 movie 與另一 movie 的 similarity 為負號時，便不需進行 prediction 計算。也因此，我在讀入檔案時，便避免將小於 0 similarity 的 movie 讀入。

```
def bonusRead(lines):
  ...
  if sim <= 0: continue
  ...
```

計算 Prediction

由於後續在計算 prediction 時，每當看未被評分 Movie ，就要進行以下三個步驟：

1. 找出該 movie 與其他 movie 的相似度，由最相似到最不相似。
2. 確認與該 movie 相似的其他 movies ，該 user 也有評分過。
3. 只找出上面兩個規則的交集中，最相似的前 10 名。

首先，我先整理出以 `movie_id` 為 key ，並將與其相似的 movie 統整至 value ，整理成 `(movie_id, [(sim_movie_id1, sim1), (sim_movie_id2, sim2), ...])` 此格式。

在整理過程中，為確保後續計算時，會從最相似開始計算，因此我做了 sorting ，讓 `[(sim_movie_id1, sim1), (sim_movie_id2, sim2), ...])` 會從相似度最高排到相似度最低。

```
def rankingSim(data):
    simMovieList = data[1]
    sortedSimMovieList = sorted(simMovieList, key = lambda data: data[1], reverse
=True)
    return (data[0], sortedSimMovieList)

movieSims
= bonusData
  .flatMap(lambda data: [(data[0][0], [(data[0][1], data[1])]), (data[0][1], [(da
ta[0][0], data[1])])])
  .reduceByKey(lambda x, y: x+y)
  .map(rankingSim)
```

同時，調用原始資料，整理出 `(user_id, [(rated_movie_id1, rating1), (rated_movie_id2, rating2), ...])` 的格式，方便後續從 User 為視角，找出缺少的 rating value 。

```
userMoviesRating = originalData
    .map(lambda data: (data[1][0], [(data[0], data[1][1])]))
    .reduceByKey(lambda x, y: x+y)
```

為求方便後續計算，我也 collect 出兩個 Python List，丟入後續的 Mapper Function :

1. `allMovieIdList` ： 所有 movie id 的集合
2. `movieSimsList` : 任一 Movie 的所有正相似 Movie List

完成上述處理後，便進行 prediction 的 Mapper 計算:

```
bonusResult =
userMoviesRating.flatMap(lambda x: predictMovieRating(x, allMovieIdList, movieSim
sList))
```

在 Mapper 開始計算前，先統計出 User 已評價的 movie id list `ratedMovieIdList` 。接著，再和 `allMovieIdList` 取差集，找出未評分過的 `missingRatedMovieIds` 。

同時，為求後續能以 `O(1)` 的方式，存取 `任一 Movie 的相似 List` & `自己評價過的 Movie List` ，我將 `movieSimsList` 與 `ratedMovies` 轉成 Dictionary 格式（Hash Table）。

```
def predictMovieRating(data, allMovieIdList, movieSimsList):
    ratedMovieIdList = [ pair[0] for pair in data[1] ] # Pure ids
    missingRatedMovieIds = [item for item in allMovieIdList if item not in ratedM
ovieIdList] # Pure ids
    movieSimsDict = dict(movieSimsList) # { movie_id: [(sim_movie_id, sim), ...]}
    ratedMovieDict = dict(data[1]) # { movie_id: rating }
    ...
```

完成上述準備後，便可以單一 User 為視角，找出其空缺的 rating 。

```
def predictMovieRating(data, allMovieIdList, movieSimsList):
  ...
  finalResult = []

  for missingMovieId in missingRatedMovieIds:
      if (missingMovieId not in movieSimsDict): continue
      highSimMovies = movieSimsDict[missingMovieId]
      accumulateSim = 0 # Lower
      accumulateRating = 0 # Upper
      count = 0
      for highSimMovie in highSimMovies:
          highSimMovieId = highSimMovie[0]
          highSimMovieSimilarity = highSimMovie[1]
          if (highSimMovieId in ratedMovieDict):
              accumulateRating = accumulateRating + ratedMovieDict[highSimMovieI
d]*highSimMovieSimilarity
              accumulateSim = accumulateSim + highSimMovieSimilarity
              count = count + 1
          if (count >= 10): break
      if (accumulateRating != 0): finalResult.append(((data[0], missingMovieId),
 (accumulateRating/accumulateSim)))
  return finalResult
```

由於有些 movie 和其餘所有 movie ，都只有負關係。對於這些 Movie 而言，他們的 `movieSimsDict` 會是空 Dict 。也因此，我在此處有做檢查避免程式錯誤。

```
if (missingMovieId not in movieSimsDict): continue
```

完成檢查後，便針對 missing rating 的 movie ，將其所有正相似的 movie_id 呼叫出來。

```
highSimMovies = movieSimsDict[missingMovieId]
```

將其所有正相似的 Movie Id 呼叫出來後，便要開始進行 prediction 計算。此處有幾項規則：

1. Prediction 的計算法則為： `accumulateRating/accumulateSim`
2. 不但要找出正相似的 Movie Id List，也必須和 user 評價 List 進行交集。
3. 最多只會計算 10 個正相似的電影。

```
accumulateSim = 0 # Lower
accumulateRating = 0 # Upper
count = 0
for highSimMovie in highSimMovies:
    highSimMovieId = highSimMovie[0]
    highSimMovieSimilarity = highSimMovie[1]
    if (highSimMovieId in ratedMovieDict):
        accumulateRating = accumulateRating + ratedMovieDict[highSimMovieId]*high
SimMovieSimilarity
        accumulateSim = accumulateSim + highSimMovieSimilarity
        count = count + 1
    if (count >= 10): break
```

完成上述計算後，即可輸出結果成 `((user_id, movie_id), predicted_val)`。由於可能遇見 `user 評價的 movie` 與 `所有正相似的 movie` 完全 **無交集** 的情況，此處有進行檢查避免程式錯誤。

```
if (accumulateRating != 0):
  finalResult.append(((data[0], missingMovieId), (accumulateRating/accumulateSi
m)))
return finalResult
```

最終再將結果 Output 至 `bonus_output.txt` File 。