Individual project CS460
Deadline: 19.05.2023

The aim of this project is to expose you to a variety of programming paradigms that are featured in modern scale-out data processing frameworks. In this project, you are expected to implement data processing pipelines over **Apache Spark**.

We provide a skeleton codebase on which you will implement missing functionality. We also provide test cases that will check the functionality of your code. Each student will be provided with a repository on GitLab (`http://gitlab.epfl.ch`, containing the initial skeleton code.

In what follows, we will go through the functionality that we expect you to implement.

Overview

You are given a dataset based on the MovieLens datasets. Your dataset includes:

- A set of movies (*e.g.*, movies, TV series) and their genres
- A log of "user rates movie" actions

A video streaming application requires large-scale data processing over the given dataset. The application has a user-facing component (which you are not concerned with) that serves recommendations and interesting statistics to the end-users. Specifically, for each title, it displays:

- All the movies that are "similar" in terms of keywords
- An average rating

In the background, the application issues Spark jobs that pre-compute the information that is served. Your task is to write Spark code for the required functionality. You will find more information in the corresponding tasks.

Dataset

There is a small, medium and large version of the dataset. The small version is distributed with the skeleton repo and the medium and large versions can be downloaded from moodle.

You are given two pipe-separated ('|') files that contain the initial data. For each file, you need to build suitable objects that correspond to records. We provide the classes of objects in the skeleton code.

movies.csv: In each row, this file contains the id and name of each title followed by a list of associated keywords/genre. For example, suppose the n_{th} title has k keywords, then the n_{th} line of the file is:

$$id_n|name_n|keyword_{n1}|keyword_{n2}|\ldots|keyword_{nk}$$

The number of keywords varies from title to title. The $name_n$ and all of the keywords are in quotation marks. e.g.

$$1|"ToyStory(1995)"|"Adventure|Animation|Children|Comedy|Fantasy"$$
$$260|"StarWars: EpisodeIV - ANewHope(1977)"|"Action|Adventure|Sci - Fi"$$

Each row is to be converted into to a $(Int, String, List[String])$ tuple, in which the first element is the title's id, the second element is the title's name, and the third element is the title's keywords. **ratings.csv:** In each row, this file contains the id of a user, the id of a title, a rating, and a timestamp. For example, the m_{th} line of the file is:

$$id_{user}|id_{movie}|rating|timestamp$$

e.g.

1|1|4.0|964982703
1|260|5.0|964981680

The timestamp field is in Unix Epoch format, i.e. the number of seconds since 00:00:00 UTC on 1 January 1970.

Each row is to be converted into a $(Int, Int, Option[Double], Double, Int)$ quadruplet, in which the first element is the user's id, the second element is the title's id, the third element is the old rating from this user for the title ($None$ if no previous rating), the fourth element is the new rating, and the last element is a timestamp for the rating action.

Apart from the initial batch load, additional ratings are appended to the data at runtime. The updates are given as arrays of the corresponding object type.

[Note: you will still have to transform the objects into the appropriate data structure for each task]

Milestone 1: Analyzing data with Spark

The first milestone is to get hands-on experience with Apache Spark and the RDDs. In this milestone, you need to load the initial data (movies and ratings) and answer simple analytical questions by transforming and manipulating data accordingly.

Task 1: Bulk-load data

Your task is to implement the **load()** methods for **MoviesLoader** and **RatingsLoader** classes to load data from **movies.csv** and **ratings.csv**, respectively. Your code reads the CSV files and converts rows into the corresponding tuples. After converting the data into the appropriate RDD, **load()** also persists the RDDs in memory to avoid reading and parsing the files for every data processing operation.

Read more details about Spark RDD caching and persistence here at `https://spark.apache.org/docs/2.4.8/rdd-programming-guide.html#rdd-persistence`

Task 2: Simple analytics with Spark RDD

After successfully loading and converting data into Spark RDD, your task is to perform simple data manipulation and extract the required analytics. In task 2, we will first pre-process the data and then, with each sub-task, answer to simple analytical questions.

Task 2.1: Pre-process movies and rating data for analytics

You first need to implement the *init*() function, which takes *ratings* and *title* RDD as input. The init function pre-processes the RDD for future analytical processing. Specifically, your task is to

- Pre-processes movies RDD by grouping movies by ID.
- Similarly, we are mostly interested in ratings of each movie every year. Therefore, the init() function should also pre-process ratings RDD by grouping it first by year and then within each group, further creating groups by movie ID, and then persists for future analytical operations.
- Further, to fully utilize and optimize for distributed data processing model of Apache Spark, we also want to partition our processed results before persisting them. Read more about partitioning RDD and why it is important at `https://www.oreilly.com/library/view/learning-spark/9781449359034/ch04.html`. You need to use a **HashPartitioner** to partition your processed RDDs, and then persist it in the variables **titlesGroupedById** and **ratingsGroupedByYearByTitle**.

(note: partitioning is not a requirement/step but a performance optimization for distributed processing)

Task 2.2: Number of movies rated each year

In this task, you need to implement the function **getNumberOfMoviesRatedEachYear**, which should return an RDD of pairs (year, number of ratings in that year).

Task 2.3: Most popular movie each year

In this task, you need to implement the function **getMostRatedMovieEachYear**, which analyzes the most popular movie, that is, having the most number of ratings, for each year. The function should return an RDD of pairs (year, movie name). If there is a tie for the most popular movie, return the movie with the greatest movie id only.

Task 2.4: Genre of most rated movie each year

In this task, you need to implement the function **getMostRatedGenreEachYear**, which analyzes the genre of the most popular movie, that is, the movie having the most number of ratings, for each year similar to task 2.3. The function should return an RDD of pairs (year, list of genres). If there is a tie, return the genres of the movie with the greatest movie id only.

Task 2.5: Most and least popular genre of all time

In this task, you need to implement the function **getMostAndLeastRatedGenreAllTime**, which analyzes the most and the least popular genre across the dataset. Similar to task 2.4, you need to first find the most rated movie (if tie, do as task 2.4), and then across genres of all most rated movies of each year, return the most and least frequency occurring genre. In case there is a tie between frequencies of genre, return the lexicographically sorted first genre. The function should return the least and most as pairs of (genre, number of ratings).

Task 2.6: Get all movies by a specified list of genre

In this task, you need to implement the function **getAllMoviesByGenre**, which filters the movies dataset and returns all the movie titles which are included in the list of genres.

Task 2.7: Get all movies by a specified list of genres using Spark Broadcast Variables

In this task, you need to implement the function **getAllMoviesByGenre_usingBroadcast**. Similar to task 2.6, you need to filter the movie dataset and return all the movie titles in the specified genres. However, in this task, you are required to use Spark broadcast variables. You can broadcast data by invoking the **broadcastCallback** function as supplied in the function input.

Read more about Spark broadcast variables at `https://spark.apache.org/docs/2.4.8/rdd-programming-guide.html#broadcast-variables`.

1

Milestone 2: Movie-ratings pipeline

Our application displays an average rating for each title. The average rating pipeline aggregates ratings in order to compute their average for all available titles (and display 0.0 for titles that have not yet been rated). Furthermore, it enables ad-hoc queries that compute the average rating across titles that are related to specific combinations of keywords. Finally, when a batch of new ratings is provided by the application, the pipeline incrementally maintains existing aggregate ratings. All the functionality is implemented in the **Aggregator** class. In the rest of the Section, we discuss how this pipeline is implemented.

Task 3.1: Rating aggregation

The ratings log includes all the "user (re-)rates movie" actions that the application has registered. The **init()** computes the average rating for each title by aggregating the corresponding actions. Titles with no ratings have a 0.0 rating. For each title in the aggregate result, you need to maintain its related keywords. **init()** persists the resulting RDD in-memory to avoid recomputation for every data-processing operation.

To retrieve the result of the aggregation, the application uses the **getResult()** method which returns the title name followed by the rating.

[Note 1: Keep in mind that there can be more than one rating for each user-title pair.]

[Note 2: There are multiple possible implementations for the aggregation and multiple representations for the result.]

Task 3.2: Ad-hoc keyword rollup

The **getKeywordQueryResult()** implements queries over the aggregated ratings. The parameter of the queries is a set of keywords and the queries compute the average rating among the titles that contain all of the given keywords. The average of averages excludes unrated titles and all titles contribute equally. The result is returned to the driver which then returns it to the end-user. If all titles are unrated, the query returns 0.0, whereas if no titles exist for the given keywords the query returns -1.0.

Task 3.3: Incremental maintenance

The application periodically applies batches of append-only updates in the **updateResult()** method. Updates are provided as arrays of *rating* tuples. Given the aggregates before the update and the array of updates, the method computes new updated aggregates that consider both old and new ratings. You need to persist the resulting RDD in memory to avoid additional recomputation and to unpersist the old version.

Hint: Incremental maintenance can use the previous rating element to reduce recomputations.

Milestone 3: Prediction Serving

Our application needs to recommend new movies for users to watch. Recommendations presented to a given user will be the top n predictions of ratings of unseen items

The application identifies titles that are "similar" each user's history to make recommendations. In order to retrieve "similar" titles, it batches a large number of near-neighbor queries i.e. queries that return titles with similar keywords. Then, using locality-sensitive hashing (LSH), it computes the near-neighbors. Finally, it will return the top n predictions of ratings on the unseen (unrated by the user) movies within the near-neighbors.

In the rest of the Section, we describe what you need to implement concretely. First, we provide a brief introduction of LSH, which contains all the required information for implementing the project. Task 4.1 describes indexing titles to support distributed LSH lookups. Task 4.2 describes the implementation of distributed LSH lookups themselves. Task 4.3 describes implementing a simple predictor for user rating. Task 4.4 describes using a more advanced predictor. Task 4.5 Brings all the tasks in this section together to generate recommendations

Background: Locality-sensitive Hashing

Often, we need to compute how "similar" two sets of objects are. Intuitively, the set $\{action, thriller, crime\}$ is not similar to $\{comedy, family\}$, whereas it is somewhat similar to $\{action, thriller, superhero\}$. For this reason, several similarity metrics have been proposed. One commonly used metric is Jaccard similarity, which is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

When $J(A, B)$ is close to 1 the sets are similar, and when $J(A, B)$ is close to 0 the sets are not similar. Usually, a threshold t is used as a cut-off point (similar iff $J(A, B) > t$, not-similar otherwise). Given a set A, the near-neighbor problem is defined as finding all sets B such that $J(A, B) > t$ for a given threshold t.

However, computing all similar sets is expensive. Let D be the set of all sets of objects. Then, computing all similar sets requires computing all pairwise Jaccard similarities which are $O(|D|^2)$. Furthermore, for computing each $J(A, B)$, the cost is at least $O(\max\{|A|, |B|\})$. Therefore, exhaustively computing all Jaccard similarities does not scale for large and high-dimensional datasets.

Consequently, researchers have proposed approximate solutions for the near-neighbor problem. A popular algorithm is locality-sensitive hashing (LSH). LSH uses a hash function h that computes a *signature* $h(A)$ for each set A. The hash function is chosen such that:

- if $J(A, B) \leq t$, then $h(A) = h(B)$ with probability at least p_1
- if $J(A, B) \geq (1 + \epsilon)t$, then $h(A) = h(B)$ with probability at most p_2

LSH, thus, uses each set's signature to retrieve all other sets with the same signature in asymptotically lower time. Even though the retrieved sets contain some *false positives* and some *false negatives*, they mostly consist of *true positives* when the hash function is effective.

In the context of this project, you are not concerned with choosing and tuning the hash function; we already provide the hash function to you. Instead, you need to focus on matching the signatures.

The following subtasks guide you through implementing and optimizing an LSH pipeline.

Task 4.1: Indexing the dataset

To identify signature matches efficiently, data needs to be organized accordingly. The **LSHIndex** class restructures the title RDD so that subsequent lookups can find matches with the same signature efficiently. This requires the following steps, which are implemented in the constructor:

1. **Hashing:** In the first step, you need to use the provided hash function to compute the signature for each title. The signature is used to label the title thereupon.
2. **Bucketization:** In the second step, you need to cluster titles with the same signature together. Suppose your RDD consist of the following five annotated titles:

```
(555, "LotR", ["ring"], 5)
(556, "Star Wars", ["space"], 3)
```

```
(557, "The Hobbit", ["ring"], 5)
(558, "Inception", ["dream"], 1)
(559, "Star Trek", ["space"], 3)
```

Then, bucketization should result in the following data structure:

```
(1, [(558, "Inception", ["dream"])])
(3, [(559, "Star Trek", ["space"]), (556, "Star Wars", ["space"])])
(5, [(557, "The Hobbit", ["ring"]), (555, "LotR", ["ring"])])
```

3. **Partitioning (Optional):** The result of bucketization is distributed. Therefore, each lookup is directed to one specific cluster of titles. To improve efficiency and reduce data movement at runtime, you need to partition the data structure by signature. Moreover, you need to cache the result of the partitioning for subsequent uses.

Task 4.2: Near-neighbor lookups

At runtime, the application submits batches of near-neighbor queries for processing. Each query is expressed as a list of keywords. Computing the near-neighbors requires the following steps:

1. **Hashing:** You need to use the provided hash function to compute the signature for each query. The signature is used to label the query thereupon.
2. **Lookup:** The signature is used to retrieve the set of titles with the same signature as the query. This will result in a shuffle for the given queries.

You need to implement the **lookup()** method in **LSHIndex** which performs signature matching against the constructed buckets. Also, you need to implement **lookup()** in class **NNLookup** which handles hashing and also uses **LSHIndex.lookup()** for matching.

Task 4.3: The baseline predictor

The baseline predictor predicts a user's rating on an unseen movie based on how the movie has been rated by other users. The following baseline incorporates some user bias, in the form of a user average rating, in predictions and then averages normalized deviations from each user's average. To understand why, observe the following two things. First, some users tend to rate more positively than others and therefore have different average ratings over all items. We denote the average rating by user u over all movies as $\overline{r}_{u,\bullet}$. (We denote the user u's rating of movie m as $r_{u,m}$) You will therefore pre-process ratings to instead express how much each rating deviates from the user's average rating ($r_{u,m} - r_{\overline{u},\bullet}$). Second, the average rating for a user does not necessarily sit in the middle of the rating scale 1, 2, 3, 4, 5 and therefore maximum deviations may be asymmetric in the positive and negative directions. Moreover, the range of deviations, in the positive and negative directions, may differ for different users. The average of many deviations from different users may therefore result in a larger deviation than the range of some users, leading to an incorrect range, i.e. < 1 or > 5, when making predictions. We, therefore, normalize the deviations such that for all users, their deviations will be in the range $[-1, 1]$ with -1 corresponding to a rating of 1 (maximum negative deviation), 1 corresponding to a rating of 5 (maximum positive deviation), and 0 corresponding to the average rating for any user.

The normalized deviation ($\hat{r}_{u,i}$) is therefore the following:

$$\hat{r}_{u,i} = \frac{r_{u,i} - \bar{r}_{u,\bullet}}{\text{scale}\left(r_{u,i}, \bar{r}_{u,\bullet}\right)}$$

with a scale specific to a user's average rating:

$$\text{scale}\ \ (x, \bar{r}_{u,\bullet}) = \begin{cases} 5 - \bar{r}_{u,\bullet} & \text{if } x > \bar{r}_{u,\bullet} \\ \bar{r}_{u,\bullet} - 1 & \text{if } x < \bar{r}_{u,\bullet} \\ 1 & \text{if } x = \bar{r}_{u,\bullet} \end{cases}$$

The global average deviation for a movie $\left(\bar{\hat{r}}_{\bullet,m}\right)$ is the average of the deviations for all users on this movie (where $U(m)$ is the set of users with a rating for movie m):

$$\bar{\hat{r}}_{\bullet,m} = \frac{\sum_{u \in U(m)} \hat{r}_{u,i}}{|U(m)|}$$

The prediction of rating for a user u on movie m, which converts back the global average deviation to a numerical rating in the range $[1, 5]$, is then:

$$p_{u,m} = \bar{r}_{u,\bullet} + \bar{\hat{r}}_{\bullet,m} * \text{scale}\left(\left(\bar{r}_{u,\bullet} + \bar{\hat{r}}_{\bullet}, m\right), \bar{r}_{u,\bullet}\right)$$

Note that if $\bar{\hat{r}}_{\bullet,m} = 0$, or there is no rating for m in the training set, then $p_{u,m} = \bar{r}_{u,\bullet}$. If u has no rating in the training set, simply use the global average, i.e. $p_{u,m} = \bar{r}_{\bullet,\bullet}$

You need to implement the above-described predictor in the **BaselinePredictor** class.

Task 4.4: Collaborative Filtering

Collaborative filtering is commonly used for recommender systems. These techniques aim to fill in the missing entries of a user-item association matrix. MLlib is Spark's machine learning (ML) library. In this task, you are required to use alternating least squares (ALS) algorithm to learn and predict movie ratings for the input pair (movie, user). Read more about ALS algorithm at `https://stanford.edu/~rezab/classes/cme323/S15/notes/lec14.pdf` and `https://spark.apache.org/docs/latest/api/scala/org/apache/spark/mllib/recommendation/ALS.html`.

Specifically, you are required to use Spark mllib's ALS interface to train and predict the rating data. In the **init()** function, you will train the ALS algorithm with the rating input data, and in the **predict** function, your implementation will take the user and movie as input and return the predicted rating.

Task 4.5: Making Recommendations

Now that we have our LSH index over the movies and prediction models, we can start serving recommendations to the users.

In the **Recommender** class, you need to implement the **recommendBaseline** and **recommendCollaborative**, which will use the implementations from **BaselinePredictor** and **CollaborativeFiltering**, respectively. Both recommender functions will first consult LSHIndex (**NNLookup**) for retrieving similar movie titles, and then, the respective prediction models will predict the rating the requesting user would have given it. Then, return the top K recommendations. Both **recommendBaseline** and **recommendCollaborative** should be stateless, i.e. any state should be set by the **Recommender** constructor.

Infrastructure

You can fully implement and test your implementation locally, on your own computer. However, we will also set up a cluster on IC infrastructure for you to run your Spark programs (we will give you access and usage details during the exercise session of 27.02). We highly recommend you to try your solutions in the cluster, so that you get the chance to experiment with HDFS and distributed Spark.

Deliverables

We will grade your last commit on your GitLab repository. Your implementation must be on the **main** branch. **You do not submit anything on moodle**. Your repository must contain a **README.md** file.

Grading: Keep in mind that we will test your code automatically. Do not change the interfaces of the provided methods.
Any project that fails to conform to the original skeleton code and interfaces will fail in the auto grader, and hence, will be graded as a zero. More specifically, you should not change the function and constructor signatures provided in the skeleton code. You are allowed to add new classes, files and packages, but only under the *scala.app* package. Any code outside the app package will be ignored and not graded. You are free to edit the Main.scala file and/or create new tests, but will be ignored during auto-grading. Tests that timeout will lose all the points for the timed-out test cases, as if they returned wrong results.

Common issues

- **Verifying/Setting the correct Java version in IntelliJ Idea**

  In the top navigation, click on File -> Project structure. You should observe the SDK version in the 'Project' tab. The project SDK should be set to JAVA 1.8.

If the value is <No SDK> or set to any other than Java 1.8, then click on the dropdown, either select Java 1.8, if available otherwise, Add SDK -> Download JDK. In the 'Download JDK' dialogue, select version 1.8 and the vendor Amazon Corretto (we have tested it to work smoothly); it will take a few seconds to download (you can see the status in the bottom right status bar in the original IntelliJ window). Click Apply, and then OK.

After setting the correct JDK version, the Scala project must be reloaded. Click the sbt button on the right side of the IntelliJ window, and then 'Reload all sbt projects' (icon: refresh symbol). It will load your sbt project, download dependencies, and set up your environment.

- ***winutils.exe not found*** **error in Windows operating system**
  - Download winutils.exe binary from WinUtils repository (use hadoop-2.7.1 version): `https://github.com/steveloughran/winutils/tree/master/hadoop-2.7.1`
  - Save winutils.exe to binary in the directory `C:\hadoop\bin\`
  - Setup environment variables:
    - In windows search, search for Edit environment variables for your account, and then add the following in the User variables section `https://docs.oracle.com/en/database/oracle/machine-learning/oml4r/1.5.1/oread/creating-and-modifying-environment-variables-on-windows.html`
    - Set $HADOOP\_HOME$ environment variable to `C:\hadoop`
    - Set $Path$ environment variable to `PATH=%HADOOP_HOME%\bin;%PATH%`
    - Restart IntelliJ

- **Invalid Spark URL**

  If you get invalid Spark URL error, then please manually set the spark driver to use the 'localhost' by setting the configuration variable 'spark.driver.bindAddress' as in the following example.

```
val conf = new SparkConf()
            .setAppName("app")
            .setMaster("local[*]")
            .set("spark.driver.bindAddress", "localhost")
val sc = SparkContext.getOrCreate(conf)
```

- **test_02_Aggregator_medium_Aggregation failing**

  Some students saw that this test would fail due to a typo in the dataset ("11'09""01 - September 11 (2002)" != "11'0901 - September 11 (2002)") If you downloaded the medium data set from moodle before March 10, you might run into this issue depending on how you implemented your MoviesLoader. To fix, re-download the medium dataset from moodle.

- **Milestone 2 tests failing due to sort ordering** We observed that some students are getting different expected results in Milestone 2, given the number of counts being the same for at least two records (one selected randomly). The project description and tests were updated for Task 2 to clarify ordering to use to break ties. The tests are automatically updated on the testing server. To update the tests locally:
  - Download the sort_order.patch from moodle.
  - From the top navigation, go to Git -> Patch -> Apply Patch.
  - Select the sort_order.patch file in the file explorer.
  - Verify that the patch will update MainTest.scala
  - Click OK for the patch to be applied
  - Commit the new MainTest.scala to the repository
  - Run tests locally

Change log:

- **2223.03.08** Clarify sort order for ties in Milestone 2
- **2223.03.08** Add common issues (with solutions)