Big Data Processing Course

Spring Semester 2016

Homework Assignment 3.

May 19, 2016

Contents

Chapter 1

Introduction

1. Homework submission is strictly in **groups of 4 students**, groups with less than 4 should ask for an explicit permition from the lecturer.
2. Homework deadline is due to: **19/06/2016 23:59**.
3. Please pay attention to the due date. I urge you to start working on the homework right away and not wait until the last minute.
4. Please try to provide clear and careful solutions.
5. You should provide comments for your code so it will be completely clear what you are trying to achieve.
   - **WARNING!** Lack of comments might lead to points reduction.
6. Homework will pass automatic check, therefore please pay attention to output format and required processing of input data, points will be automatically reduced for non complaint output (even if this is only a single digit or character).
7. Please remember that a significat part iof your homework solution is understanding of requirements, therefore I urge you to read through this document very careful and make sure you understand each single part of it. Solutions verified with automatic tooling, therefore in case of misunderstanding of requirements or in case of non compliance, points will not be returned back.

Chapter 2

Assignment Details

In this **last** assignment you are going to work with Spark APIs. Similar to the homework 1 here you provided with **same** file which includes movies reviews information collected from online resources. You are required to use this file to provide implementation of data analytic algorithms to asnwer different queries which will be described in the following section 2.3. Now you need to use Spark APIs to implement these queries. You may and you should rely on your implementation submitted for the first homework. However please take careful look on mistakes you might have had and fix them.

2.1 Project requirements

This time you will start with very minimal maven project pom.xml, and you are required to add additional dependencies for the Spark framework on your own.

Listing 2.1: pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>univ.bigdata.course</groupId>
  <artifactId>final-project</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>uberJar</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <!-- YOU NEED TO ADD SPARK RELATED DEPENDENCIES HERE -->
  </dependencies>

  <build>
```

```
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.3</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.4.3</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <transformers>
            <transformer implementation="org.apache.maven.plugins.
                                shade.resource.ManifestResourceTransformer">
              <manifestEntries>
                <Main-Class>univ.bigdata.course.MainRunner</Main-Class>
              </manifestEntries>
            </transformer>
          </transformers>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

It's solely up to you to decide about interfaces, classes and methods, the only **obligatory** requirements: all queries have to be implemented using Spark.

2.2 Execution

There is a class with a main method **univ.bigdata.course.MainRunner** which serves as an entry point for your program. Maven configuration above will take care of creating **"uber"** executable jar. To produce executable jar you have to run

Listing 2.2: command line

```
mvn clean install
```

It will produce **final-project-1.0-SNAPSHOT.jar** file within **target** directory.

Next you should be possible to execute your program locally with following parameters:

1. ```
   ./bin/spark-submit --class univ.bigdata.course.MainRunner \
       --master local[*] \
       /path/to/final-project-1.0-SNAPSHOT.jar \
       commands commandsFileName
   ```

   Where **commandsFileName** is the name of the input files which includes information described in the section 2.2.1. Then your programm will have to execute all commands provided in the **commandsFileName** according to the information in 2.2.1 and produce proper output (pretty similar to your first assignment).

2. ```
   ./bin/spark-submit --class univ.bigdata.course.MainRunner \
       --master local[*] \
       /path/to/final-project-1.0-SNAPSHOT.jar \
       recommend recommendFileName
   ```

   Where **recommendFileName** is the name of the input files which includes information described in the section 2.2.2. Then your program will have to provide top-10 movies recommendations for the list of users from the **recommendFileName** for all information see 2.2.2.

3. ```
   ./bin/spark-submit --class univ.bigdata.course.MainRunner \
       --master local[*] \
       /path/to/final-project-1.0-SNAPSHOT.jar \
       map movies-train.txt movies-test.txt
   ```

   Where **movies-train.txt** file is used as input file which includes review information as described in 2.2.3 and will be used to 'train' the recommendation algorithm and **movies-test.txt** the name of the input file which includes review information as described in 2.2.3 and will be used as the 'test' dataset used to evaluate the quality of the results (the unseen dataset).

   Your algorithm needs to produce an RDD which holds a ranked list of recommendations from the dataset **movies-train.txt**. Note that the RDD must contain all the movies in **movies-train.txt**, the ratings they are given is not important but the rank is.

   The relevant movies for the purpose of MAP for each user will be movies the user gave a rating of 3 or higher in the dataset **movies-test.txt**. So MAP can be calculated for the ranked RDD according to the MAP formula. Please note that since the list of movies can potentially be very large do not assume it fits in the RAM of a single machine! Having done all the calculations above you are to output a MAP score to stdout (see http://web.stanford.edu/class/cs276/handouts/EvaluationNew-handout-6-per.pdf).
   ```

4. ```
./bin/spark-submit --class univ.bigdata.course.MainRunner \
    --master local[*] \
    /path/to/final-project-1.0-SNAPSHOT.jar \
    pagerank movies-simple.txt
```

   Computes PageRank algorithm provided by **MLib** on the users graph induced by movies review information. Where **movies-simple.txt** is the file which includes all movie reviews as in section 2.2.3. Each user which provided review to the movies represents the graph vertex. Two vertices are connected with an edge **iff** two users provided review for same movie. **Once users/verticies are connected with an edge, there is no need to add an additional edge for the cases where they reviewed more than one common movie**. Output should be sorted according to the pagerank score ties should be broken by sorting with users id using lexicographical order. You required to output only 100 first users.

We encourage you to learn how maven dependencies mechanism works, especially different scopes

https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.

For more details how to submit your program for Spark cluster see:
https://spark.apache.org/docs/latest/submitting-applications.html.

ATTENTION!
Please use your second homework and YARN cluster on virtual machines to check how spark-submit works on YARN. Here the example of how submition of commands query should look like:

```
./bin/spark-submit --class \
--master yarn \
--deploy-mode cluster \
--executor-memory 1G \
--num-executors 4 \
/path/to/final-project-1.0-SNAPSHOT.jar \
commands commandsFileName
```

For more details please refer to the link above.

2.2.1 Commands file format

- First line is the name of the input file which includes review information as described in 2.2.3
- Second line is the name of the output file where you expected to print all outputs of the queries. Expected output will be attached to the project **queries_out.txt**.
- Rest of the lines are the commands/queries, same as in first assignment in the format explained in 2.3.

2.2.2 Recommendation file format

- First line is the name of the input file which includes review information as described in 2.2.3
- Second line is the name of the output file where you need to output recommendations for movies for each user sorted in decreasing order in case of equality should be sorted lexicographically by movie product id. Example of expected output attached to the project **reccomends_out.txt**
- Rest of the lines are users ids.
```

NOTE: All input and output files might be located at any place on the disk (not necessarily in /src/-main/resources folder), therefore you expected to accept absolute paths only.

2.2.3 Movies reviews input file format

Reviews files consist of 7 tab separated fields as following:

1. **product/productId** - the id of the movie from the online resource
2. **review/userId** - id of the user which provided review
3. **review/profileName** - name of the user profile
4. **review/helpfulness** - how helpful given review (rate by other users)
5. **review/score** - the rating score given by user
6. **review/time** - the timestamp in milliseconds, the time when review was submitted
7. **review/summary** - review summary
8. **review/text** - review text

2.2.4 Content sample

```
product/productId: B00004CK40  review/userId: A39IIHQF18YGZA review/profileName:
C. A. M. Salas  review/helpfulness: 0/0 review/score: 4.0 review/time: 1175817600
review/summary: Reliable comedy review/text: Nice script, well acted comedy,
and a young Nicolette Sheridan. Cusak is in top form.

product/productId: B00004CK40  review/userId: AP8GQ56CJ8MYC review/profileName:
C. MACHADO  review/helpfulness: 0/0 review/score: 5.0 review/time: 1171929600
review/summary: I still love this movie review/text: I first saw this movie when
it was released in the 80's. I loved it. Some 20 years later I had the opportunity
to view it again, and I still love it. It's storyline is timeless as is it's humor.
I found the exchanges between the two lead characters so natural and FUNNY. It is
still one of my very favorite movies of all time. John Cusak is fabulous!! Please
see it. If you want to feel good, it's a movie that will help you get there.
```

2.3 Commands

Here is the list of the command queries you might find in commands input file as explained 2.2.1, NOTE this is a case sensetive queries:

1. **totalMoviesAverageScore** - compute overall average score for all reviewed movies.
2. **totalMovieAverage productId** - given movie (the product id), compute total average for that movie.
3. **getTopKMoviesAverage topK** - compute a list of overall total averages for all movies in the dataset. List should be sorted according the average score of movie and in case there are more than one movie with same average score, sort by product id lexicographically in natural order.
4. **movieWithHighestAverage** - locate the movie with the highest average score across all movies in the dataset. If more than one movie has same average score, then movie with lowest product id ordered lexicographically.
5. **mostReviewedProduct** - return product id (the movie) which has highest amount of distinct reviews. if more than one movie has same amount of user reviews tie should be broken by product id lowest (lexicographically) should come first.
6. **reviewCountPerMovieTopKMovies topK** - return a map of product id and number of review the movies has received according to the dataset you are provided with. Map should be sorted according to the number of reviews in decreasing order in case of equality ties should be broken by natural order of product id.
7. **mostPopularMoviewReviewedByKUsers numOfUsers** - compute most popular(with highest average score) movie which was reviewed by at least K users. If there are more than two movies with same number of reviews movie with lowest product id should be picked up, according to lexicographical order.
8. **moviesReviewWordsCount topK** - construct the map of words used for movies review and number of word occurences. Consider words as separated by space. Map ordered by words count in decreasing order, for words with same number of occurances lexicographocal order should be applied.
9. **topYMoviewsReviewTopXWordsCount topMovies topWords** - give a map of top X words count for top Y most reviewed movies, sorteded by words counts in decreasing order.
10. **topKHelpfullUsers K** - compute top k most helpful users that provided reviews for movies. Map ordered by number of reviews in decreasing order or according to the user id sorted lexicographically.
11. **moviesCount** - total movies count.

2.3.1 Example of commands input file

Listing 2.3: commands.txt

```
movies-simple.txt
queries_out.txt
getTopKMoviesAverage 2
getTopKMoviesAverage 2
totalMoviesAverageScore
movieWithHighestAverage
reviewCountPerMovieTopKMovies 4
```

```
mostReviewedProduct
moviesReviewWordsCount 100
topYMoviewsReviewTopXWordsCount 100 100
topYMoviewsReviewTopXWordsCount 100 10
mostPopularMovieReviewedByKUsers 20
mostPopularMovieReviewedByKUsers 15
mostPopularMovieReviewedByKUsers 10
mostPopularMovieReviewedByKUsers 5
topKHelpfullUsers 10
topKHelpfullUsers 100
moviesCount
```

2.3.2 Example of recommendation input file

Listing 2.4: recommend.txt

```
movies-simple.txt
recommendation_out.txt
AHCVWPLA1O4X8
A39CX0EE4BZCZC
A1L12LCCJ4VGJS
```

2.4 Floating operations

Exactly as was published in clarification for **Homework 1** about floating pointing operation, you should round results of all floating points operation up to **5 digits** precision after the dot. The rounding rule called **Round half up**. For example:

- Number **326.23567**8 should be rounded to 326.23568
- Number **0.83160**512987 should be rounded to 0.83161
- Number **17.83969**31496 should be rounded to 17.83969
- Number **121.02989**657 should be rounded to 121.0299

For more information about rounding you can find here: https://en.wikipedia.org/wiki/Rounding

NOTE: you are required to **ROUND** results and not simply truncate them.

Chapter 3

Evaluation

1. ```
   ./bin/spark-submit --class univ.bigdata.course.MainRunner \
       --master local[*] \
       /path/to/final-project-1.0-SNAPSHOT.jar \
       commands commandsFileName
   ```

   the output of this command suppose to be **EXACTLY** the same as the expected output provided in the sample and during the automatic check.

2. ```
   ./bin/spark-submit --class univ.bigdata.course.MainRunner \
       --master local[*] \
       /path/to/final-project-1.0-SNAPSHOT.jar \
       recommend recommendFileName
   ```

   the output of this command should provide reasonable recommendations for given users id, however **NO EXACT** match is required.

3. ```
   ./bin/spark-submit --class univ.bigdata.course.MainRunner \
       --master local[*] \
       /path/to/final-project-1.0-SNAPSHOT.jar \
       map movies-train.txt movies-test.txt
   ```

   here as well **NO EXACT** is required.

4. ```
   ./bin/spark-submit --class univ.bigdata.course.MainRunner \
       --master local[*] \
       /path/to/final-project-1.0-SNAPSHOT.jar \
       pagerank movies-simple.txt
   ```

   NO EXACT is required

3.1 Execution Model

Your program will be checked with two execution models. First one as described in 2.2 locally using maven command. And the second one is by running your program on cluster using YARN, therefore in order to check yourself you will need your results from previous homework 2. You can learn how to execute Spark on YARN cluster here: http://spark.apache.org/docs/latest/running-on-yarn.html. Moreover we encourage you to leverage your second homework to create YARN cluster with Vagrant and Virtualboxes in order to check yourself.

ATTENTION!!! Your submition has to **RUN** with BOTH execution models without any problems in order to get full grade and it's up to you to verify it before submitting your final work. E.g.

```
./bin/spark-submit --class \
--master yarn \
--deploy-mode cluster \
--executor-memory 1G \
--num-executors 4 \
/path/to/final-project-1.0-SNAPSHOT.jar \
commands commandsFileName
```

for all query types.

All your works will be tested using latest Java 8 updated version.

Chapter 4

Submission

This is an electronical submission via course portal on Moodle. To complete your submission you have to:

- Provide your students details at the top of each submitted file.
- Add **hw1_sol.pdf** file with short description of the implementation details and description which explains what was the responsibilities of each teammate during the assignment.
  - **NOTE:** Without this files 10pt will be reduced.
- Add **README.txt** file which will also include information in the format:

Listing 4.1: README.txt

```
First and Second Name;email1;ID1
First and Second Name;email2;ID2
First and Second Name;email3;ID3
First and Second Name;email4;ID4
```

- All files should be archived with zip and final name for submission should be **homework1.zip**.
- **NOTE:** submissions which will be missed **README.txt** will not be graded.

GOOD LUCK!