Homework 3
Final Project
Big Data Course 2016-B

Group Name:

G-City

Team Members:

Ilan Godik	- 316315332 -	ilan3580@gmail.com
Yuval Alfassi	- 318401015 -	u67v67@gmail.com
Gil Henkin	- 316278118 -	gil7788@gmail.com
Tony Tannous	- 205735046 -	ttannous@campus.haifa.ac.il
Charlie Mubariky	- 316278118 -	Charlie_mabariky.1996@hotmail.com

Lecturer:

Mr. Roy Levin

Inspector:

Mr. Artem Berger

Submission Date:

30.6.2016

Contents

Introduction

This is a report describing the details of our implementation to the homework assignment given in the Big-Data course.
This project is constructed of four parts.
In the first part we have to process and retrieve information about movie reviews.
In the second part we suggest recommended movies for certain users, using the ALS algorithm as the recommendation engine.
The third part is a empiric validation system for the second part. It tells us how good is the recommendation engine we have built. We use the Mean Average Precision algorithm to calculate how accurate and precise our recommendations are.
In the fourth part we build a graph from the reviews. We use the Pagerank algorithm to decide who are the most dominant hundred users in the dataset.
In addition, we had to configure the pom.xml file and engineer a Vagrant cluster to run our program on, and we generated 3 synthetic random datasets for meaningful validation, with dense enough data.

Code Structure

Programming Language:

We used both Java and Scala, but mainly Scala.

Main function:

Our code starts with a Java main method which is found at the MainRunner class. This main function passes its given arguments to our Scala main which is found at the SparkMain class.
In the SparkMain's main function, we parse the input according to the command to execute, and then call the appropriate execution function.
In the SparkMain class we initiate a Spark configuration object and a Spark context object.

Data types and design:

MovieReview:

Contains a movie review data. Contains the movie ID of the review, the user ID of the reviewer, the profile name of the reviewer, the helpfulness value of the review, the review's score, the review's timestamp, the summary of the review, and the review itself in text.

Movie:

A movie ID, together with a vector of it's reviews.

MovieIOInternals:

Class for IO and parsing of the movie reviews.

Command:

A large sum type, representing all the possible structured commands that can be requested to be executed in part 1.

TestedUser:

Class represents a user whom we have to recommend to. Contains the user's ID, the movies he has see, and the movies he hasn't seen from the test set.

LongALS, LongMatrixFactorizationModel:

We ported the corresponding MLLib classes for executing ALS to work with Longs intead of Ints. These classes wrapped the ml.ALS class functionality, which implements the algorithm and supports Longs as well as Ints.

In addition, we added several methods providing an RDD of all recommended movies for a user, instead of only the topK recommendations.

part1.MoviesFunctions, part2.Recommendation, part3.ExecuteMap, part4.PageRank:

These are the objects that contain the implementations for each of the parts of the assignment.

Part 1
Information Retrieval

Goal:

Retrieve information about a big dataset of movie reviews.

Given:

A big data-set of movie reviews and several command requests.

Parsing:

Firstly, we parse the given commands, which is found at ParseCommand.parseCommandsTask. It returns either a string describing the source of the parsing error, or a CommandsTask in the case of success.
CommandsTask contains the path to the input & output files, together with a vector of Commands, which are ADTs (algebraic data types), representing all the possible command forms and their inputs.

Execution:

We parse the input file of the reviews to a RDD[MovieReview], and then iterate the given commands, executing each one of them. Each Command implements a function called execute, which, when given an RDD[Movie], will call the appropriate function and produce the result string, which would later be outputted to the output file.
All the functions are implemented very similarly to how they were implemented in HW1, as we used the Java Streams API for HW1.
Many of the functions got even shorter, as we exploited much more core reuse this time around.

Part 2
Movie Recommendations

Goal:

Recommend movies to a set of given users.

Given:

A big data-set of movie reviews, and user IDs to recommend to.

Method:

Recommend movies to the users by the ALS algorithm, which factorizes the rating matrix to vectors of latent features for each user and each movie, based on the user's past preferences and collaborative ratings of movies, influencing the 'type' of the movie.

Parsing:

At first, we parse the input file in ParseCommand.parseRecommendationTask.

Execution:

Execution of this stage is found at the Recommendation class.
At first, we normalize the RDD of reviews so each movie would have an average score of zero for the ALS algorithm. This is necessary in order to account for the cold start problem: what movies to suggest for users who have not rated any movies yet.
Because the ALS doesn't work with String inputs of usernames and movie names, but with Long values (with our port) - we convert the usernames and movie names to unique ids of type Long - we use the cryptographic hash function SHA-256 and take its lowest 64 bits, which is as good as a hash function of size 64 bit, as hash functions are modeled as oracle random functions, which ensures us probabilistically that we'll start to encounter collisions only after 2^32 instances.
We used hashes in order to avoid alignment of IDs by some ordering of the data and between datasets, and to avoid lookups of IDs in an association table.

We convert our normalized reviews' usernames and movie names to Long values, on which we run the ALS training algorithm, and get a LongMatrixFactorizationModel in return.
Parameters: We run ALS on a rank of 75 and 15 iterations. We varied the parameters, and tested on the many datasets we had, trying to provide enough dimensionality to be able to discriminate, but not too many to avoid overfitting of the data.
Afterwards, we get from the model a feature array for each user, which represents his particular taste (of latent features) of movies.
Then, we get the ALS's computed score for each movie by calling the function LongMatrixFactorizationModel.allRecommendations with the features array, which yields scores for all the movies. From these movies we need only movies which the user hasn't seen already so we filter out all the movies which the user has already seen. We take the ten best recommendations and append it to the given output file.
Note - the entire interaction with the ALS algorithm is with Long values which are the hashes of our movie names and usernames.
The ALS Model holds Long features instead of the Strings IDs we are interested in. We hold an assoc table of Long IDs with Strings to recover the String IDs we are interested in.

Part 3
MAP Algorithm

Goal:

Calculate a Mean Average Precision value for the recommendations ranking which is a feedback value for how good is our recommendation engine.

Given:

A train set of reviews and a test set of reviews.

Method:

Train an ALS model on the train set, get relevant recommendations for each user, rank the unseen movies for each user, calculate average precision for each of those rankings, and return the mean of all of those.

Execution

First, we train an ALS Factorization Model on the normalized reviews. Then, for each user in the test set (Notice it's an RDD of user -> reviews, and for each means .toLocalIterator - the driver goes over each user locally and submits their corresponding jobs), we score all movies via allRecommendations, filtering movies only the relevant movies - we rank only movies which the user hasn't seen already, otherwise we would be testing (recommending) on the train set. We go over the movies the user has seen in the test set for which he gave a score of 3 or more, and mark the ranks of these movies according to the scores given by the ALS Factorization. From those rankings (Array[Int] - first seen at rank x1,...), we calculate the MAP value by calling the calcMap function which returns the mean of all the average precisions of the rankings for the users. At last, we print the MAP value to the stdout.

Part 4
PageRank algorithm

Goal:

Find the top hundred users who are ‘important’ and ‘dominant’ in the sense of their reviews.

Given:

A dataset of reviews.

Method:

Create reviews graph and run Pagerank algorithm on it, take the best 100 users.

Execution

First, we construct a graph of all the users, the graph is constructed as follows: the vertices are the users. Two user’s vertices are connected with an edge if both have reviewed the same movie. self loops or parallel edges are disallowed.
The construction of the graph is done in the function createGraph which is found in the PageRank class.
After constructing the graph we calculate the pagerank score for each user, and output the best one hundred users to stdout.

Work distribution

- ❖ Ilan:
  - Configuration of pom.xml file
  - Data representation & Conversions
  - IO & Parsing
  - Part 2 implementation
  - Part 3 design
  - Cluster setup
  - Testing
- ❖ Yuval:
  - Part 1 input parsing
  - Part 2 implementation, refactoring and testing
  - Part 3 MAP design, management and testing
  - Creating artificial meaningful datasets
- ❖ Tony:
  - Part 1 & 3 implementation
  - Cluster setup and run code on cluster
  - Testing
- ❖ Charlie:
  - Managing and implementing part 1
  - Managing and developing part 4
  - Testing
- ❖ Gil:
  - Managing and implementing part 1
  - Managing and developing part 4
  - Testing