Team hadoop05
Nikolaos Tsikoudis
Dimokritos Stamatakis
William Edgecomb
Tyler Lichten

COSI 129a: Assignment 4 Report

Overview

We utilize the Spark mllib.recommendation library to train our model and generate recommendations, and our code is written in Scala. The mllib.recommendation.Rating objects used to train our model are constructed in a straightforward manner from the raw data of Amazon reviews. Each review used to train the model gets a rating object constructed from a product ID, a user ID, and the 1 through 5 score (rating) given in the review. Following training, the model's RDD of product features is referenced, and a for loop is used to associate each of the 100 products in file *items.txt* with its resulting top 10 recommendations. Within each iteration, map is called on the RDD of product features to get the dot product of each *product feature vector, target feature vector* pair. The products with the 10 highest resulting dot products are then recommended.

Resources/Articles

To build the Ratings objects and train our model, we followed the example at:
http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html
To inform our coding, we relied (to a smaller extent) on the documentation at:
https://github.com/apache/spark/tree/master/mllib/src/main/scala/org/apache/spark/mllib/recommendation
As a reference for understanding more broadly the theory of collaborative filtering techniques, we read the paper at:
https://datajobs.com/data-science-repo/Recommender-Systems-%5BNetflix%5D.pdf
Additionally, we referred to the rest of the resources provided in the Resources section of the assignment PDF to a lesser extent than those resources listed above.

Techniques and Thought Process

Training the Model:

The decision to use only tuples of (userID, productID, rating) to train our model was based on two considerations. One, we were warned about constraints on memory, and this design allows us to train with very lightweight data. And the second reason was ease of programming, as the models of mllib.recommendation.MatrixFactorizationModel are trained on this information precisely.

Thus for preprocessing our main task is eliminate the unneeded information associated with each review and keep only tuples of (productID, userID, rating). We achieve this through a series of RDD transformations, first eliminating the unnecessary fields, and then grouping together the three data items for each review. At this point, in some but not the final version of our code, we filter out those reviews for which the product reviewed has no other reviews in the dataset. And then since Rating requires product ID and user ID parameters of type Int, and our IDs at this point are Strings, we create hash maps to convert from String to Int, and also in the case of product IDs a hash map to convert from Int to String, as this is required for outputting our final recommendations.

We experimented with filtering out singly-reviewed products with the intent to reduce memory overhead, and also to reduce noise in our data. For with only one review of a product, the data we have for that product is likely to be of poor statistical significance. Indeed, in calculating the mean squared error of our model as per the above referenced mllib example, we noticed a significant reduction in error after filtering out reviews in which the product is reviewed only once. For similar reasons, we also experimented with filtering out reviews in which the user ID was "unknown" according to the dataset. However both of these filtering operations proved problematic (see section *Issues*).

Generating Recommendations:

Our task in generating recommendations is to recommend products given only a single target product—thus no user is associated with the target product. Our model's class, MatrixFactorizationModel, provides a means to generate recommendations in this manner. In training the model (using ALS.train() from mllib.recommendation.ALS), behind the scenes the product feature matrix is calculated using the alternating least squares method and is referenced in the class's fields as an RDD (along with the user feature matrix). Furthermore, the dot product of two product feature vectors is a measure of the similarity between the two associated products. What we do then is reference the product feature matrix, and for each target product use RDD transformations to calculate its dot product with every product represented in the matrix except itself, take the 10 with the greatest result and print them (a file containing the output, *recommendations.txt*, is included in our submission). To calculate the dot product we define our own function that gets as arguments two arrays which are feature vectors and returns the result of the dot product as double.

Running our code

In our home directory on the Akubra cluster is saved a scala file, *recommend_products.scala*. That file is the entirety of our project's code.

To run the code from our home directory, first enter:

```
export DATA_PATH=/shared3/data-medium.txt
```

After entering the above code, then enter this next:

```
spark-shell --master yarn --deploy-mode client --queue hadoop05 --driver-memory 4g
--executor-memory 4g --executor-cores 2 -i recommend_products.scala
```

The program will output all targeted products from *items-medium.txt* and their 10 recommended products.

Issues

One persistent issue is that the products in *items.txt* include products that we intended to filter out, i.e. those in which only "unknown" users have reviewed it as well as those which have only one review. If we do filter along these lines, then such products in *items.txt* are not represented in our product feature matrix and we have no way to provide recommendations, and furthermore without error-handling our code will throw exceptions. As a workaround, at whatever expense to our results and to performance, we decided to simply allow all reviews into our model—thus not do any filtering of reviews, and allow "unknown" to be treated as a single user by the model. This choice simplifies the process of generating recommendations, but reduces our chances of success in running on the large dataset.

We learned early the value of using RDD data structures and transformations over traditional methods. An early attempt to put Strings representing all reviews into an array resulted in an OutOfMemory error. Instead we found means to work with the dataset through RDD transformations, which solved memory problems and sped up processing. A significant hurdle was figuring out how to get groupings of (product ID, user ID,, score), since in the dataset each is on its own line. We eventually settled on a combination of methods *zipWithIndex* and *reduceByKey*. However early attempts at this were only partially successful, as the iterator value that results from reduceByKey would sometimes scramble the order of the three subvalues. We resolved this issues by initially adding character prefixes to each of product IDs, userIDs, and scores, and then sorting the iterator after reduceByKey to recover the desired order.

Once we were nearing the final version of our code we had no memory issues or inefficiencies with the medium data set. However on the large dataset we tended to get an OutOfMemory error at the step where *ALS.train* is called. To help, we tried using the *unpersist()* method in order to free up memory from RDDs that were no longer required.

Best Results:

In the end we successfully ran our code on the medium dataset, but not on the large dataset. However we have a version of our code where we do filter out singly-reviewed products and reviews with "unknown" users, and then error-handle in case a product ID in *items.txt* is not in our model. It is possible that with that code we can succeed with the large dataset.