Recommender Systems - Lab exercise

Vincent Leroy, Natasha Tagasovska

2017

1 Introduction

Recommender systems are among the most popular applications of data analytics. The ultimate goal is to predict whether a customer would like a certain item: a product, a movie, or a song. A key concern is the scaling factor of such systems, since computational complexity increases with the size of a company's customer base. In this exercise you will learn how to use Spark's MLlib to build a movie recommender system.

2 Dataset

The dataset we will use for this exercise comes from the MovieLens Website. It is very often used by researchers and industrials to experiment with a variety of recommender systems and compare their performance. Since this course advertises Web-Scale analytics, we obviously select the largest dataset available: MovieLens 20M Dataset. This dataset contains 20M user ratings for over 27k movies (ratings.csv). In this file, users and movies are identified with integers, and ratings are floating point values from 1 to 5. Using this information, we can compute recommendations using a *collaborative filtering approach*. Furthermore, meta-data about movies is available. For this exercise, we will use the 18 movie genres used to annotate movies. This information is available in `movies.csv` along with the title of each movie.

3 Collaborative Filtering

Collaborative filtering recommender systems rely on user ratings to predict unknown ratings. For this exercise, we will use a method of the Matrix Factorization family called **ALS** that was presented in class. ALS is available in Spark through the MLlib library.

3.1 Train a model and tune parameters

To get started with the exercise download the code and build the sbt project as usual. In terminal, at the root of the project (you should see `build.sbt` when you type ls), run sbt eclipse. Now, in Scala-IDE, use the import wizard and the Import existing projects into workspace option to open your project in the IDE.

In `ALSParameterTuning.scala` you will find some warm-up code that will help you get familiar with the ALS algorithm. First of all we need to load the dataset into the RDD `ratings`. The tuples in this RDD are instance of:

`org.apache.spark.mllib.recommendation.Rating(user: Int, product: Int, rating: Double)`.

In order to get more accurate predictions, we normalize the ratings per user with `avgRatingPerUser`. Having this done, we are ready to train an ALS model.

The train method of ALS we are going to use has the following parameters:

- *rank* - is the number of latent factors in the model.
- *iterations* - is the number of iterations of ALS to run.
- *lambda* - specifies the regularization parameter in ALS.

Different values of the parameters result in different models and consequently different quality in predictions. To compare different models we will use Mean Squared Error, (MSE) as an evaluation criterion. To be more objective in the evaluation process, we split the data in train and test sets of sizes 9:1. We train and tune parameters on the 90% of the data, and then test on the 10% we put aside at the beginning.

In `ALSParameterTuning.scala` we provide the code for the split and calculation of MSE. MSE is a simple sum of the errors we make in our prediction compared to the true ratings.

Ideally, we want to try a large number of combinations of parameters in order to find the best one, i.e. the one with lowest MSE. Due to time constraints, you should start by testing only 12 combinations:

1. Change the values for `ranks, lambdas and numIters` and create the cross product of 2 different ranks (8 and 12), 2 different lambdas (0.01, 1.0 and 10.0), and two different numbers of iterations (10 and 20). What are the values for the best model? Store these values, you will need them for the next exercise.

3.2 Getting your own recommendations

Now that you have experimented with ALS, you know which parameters should be used to configure the algorithm. You can switch to `collaborative_filtering.scala`.Our objective now is to go beyond aggregated performance measures such as MSE, and see if you would be satisfied by the recommendations of the system you just built. To do this, we need to add you as a user in the dataset.

1. Build the `movies Map[Int,String]` that associates a movie identifier to the movie title. This data is available in `movies.csv`. Our goal is now to select which movies you will rate to build your user profile. Since there are 27k movies in the dataset, if we select these movies at random, it is very likely that you will not know about them. Instead we will select the 200 most famous movies and ask you to rate 40 among them.

2. Build `mostRatedMovies` that contains the 200 movies that were rated by the most users. This is very similar to wordcount, and finding the most frequent words in a document.

   Obtain `selectedMovies List[(Int, String)]` that contains 40 movies selected at random in `mostRatedMovies` as well as their title. To select elements at random in a list, a good strategy is to shuffle the list (i.e. put it in a random order) and take the first elements. Shuffling the list can be done with `scala.util.Random.shuffle`.

3. Congratulations, you can now use your recommender system by executing the program! The function `elicitateRatings(selectedMovies)` will give you 40 movies to rate and you can answer directly in the console in the Scala IDE. Give a rating from 1 to 5, or 0 if you do not know this movie. Are you happy about your recommendations?

4 Content-based Nearest Neighbors

For collaborative filtering, we relied purely on rankings and did not use movie attributes (genres) at all. For this part of the exercise (`nearestneighbors.scala`), we will use a different method: content-based recommendation. Our goal here is to build for each user a vector of features (genres) describing their interests. Then, we will find the k users that are most similar using those vectors and cosine similarity to obtain a recommendation.

1. For this exercise, we will transform ratings into binary information. There are movies the user liked and movies the user did not like. Build the `goodRatings RDD` by transforming the ratings RDD to only keep, for each user, ratings that are above their average. For instance, if a user rates on average 2.8, we only keep their ratings that are greater or equal to 2.8.

2. Build the `movieNames Map[Int,String]` that associates a movie identifier to the movie name. You have already done this in the previous part of this exercise.

3. Build the `movieGenres Map[Int, Array[String]]`that associates a movie identifier to the list of genres it belongs to. This information is available in the `movies.csv` file, in the third column, and movies are separated by "|". If you use split, you will need to write "\\|" as a parameter.

4. We provide the code that builds the `userVectors RDD`. This RDD contains `(Int, Map[String, Int])` pairs in which the first element is a user ID, and the second element is the vector describing the user. If a user has liked 2 action movies, then this vector will contain an entry ("action", 2). Write the userSim function that computes the cosine similarity between two user vectors. The mathematical formula is available on the slides. To perform a square root operation, use Math.sqrt(x) .

Congratulations, you can now experiment with your recommender system by modifying the vector of testUser and see which recommendations you get. Try to understand the way we compute `knn`, the list of k user that are most similar to `testUser`, and `recom`, the list of movies recommended to the user.