Lab 5: Recommender Systems

April 2nd, 2014

Objectives

This week's lab will explore collaborative filtering techniques for recommender systems. The objectives are as follows:

- Learn about the Netflix Prize, a popular data challenge that has had a big impact on the state of the art in recommender systems.
- Discover the Apache Mahout project[1], which provides many useful data analytics tools for Hadoop.
- Build a distributed map-reduce implementation of a simple memory-based recommender system using Mahout libraries and Hadoop.
- Understand and play with a high-performance recommender system based on UV decomposition.

Deliverables

- 3×20 recommendations for jack using the similarity-based recommender system and the two latent-factor-based recommenders (3, respectively 25 features).
- RMSE values for user `16355` for the 3 recommender systems.
- A plot of the movies in the 3D feature space generated using the `plot3d.py` script.

[1] `http://mahout.apache.org/`

1 The Netflix Prize

Netflix, a well-known American movie rental company, announced in 2006 the launch of a competition with the goal of improving the accuracy of their recommender system. They chose a metric, *RMSE* (read more about it below), and promised one million USD to the first team who would improve this metric by at least 10% (relative to the performance of Netflix's current system at the time). Contestants had access to a dataset consisting of about 18,000 movies, about 500,000 users and a total of a little over 100 million ratings. The competition ran fiercely over three years, with several thousands of teams submitting prediction sets; it ended in 2009 after a climactic race between the two main contenders. The competition generated a lot of enthusiasm in the recommender systems community and resulted in tangible progress in the field.

2 Dataset Processing

Remark: this lab uses Mahout, which consists of several commands and libraries. To properly configure your environment, run the following command every time you connect to one of the cluster-access servers:

```
source scripts/setup.sh
```

The script is available in the archive you donwloaded from Moodle. Make sure that you *source* it, not just execute it. This will set up environment variables and configure aliases that will make your life easier.

We are going to work with the raw dataset provided for the Netflix Prize. It is located on HDFS in `/ix/netflix`.

- Take a look at the dataset. Check out the `README` file, and try to understand what is the purpose of each file and folder. In particular: what does `training_set` contain? What are the content and the format of each of the files inside?

Because recommending movies for anonymous users is boring, we created a web application that lets you rate some movies and export your ratings. You can check it out at `http://guarded-fortress-4195.herokuapp.com/`.

- On the web application, rate at least 15 movies and save your ratings on the computer. The ratings exported from the web app are assigned to the user ID `12345678`, you will need it later when computing recommendations for yourself.
- In addition to your own ratings, we also provide you with the ratings of Jack, a fictitious user who has a strong preference for action and sci-fi movies. His ratings are available in the file `data/jack.tsv`, and his user ID is `12345679`.

Our first step is going to be the processing of the dataset in a format that will be easier to manipulate later on. The goal is to have a single, large file that contains one rating per line (in what follows, we'll refer to this file as `dataset.tsv`):

```
userID<TAB>movieID<TAB>rating<NEWLINE>
```

- Complete the function `processFile()` in `NetflixParser.java`. You can check your code with the unit test that is provided.
- Export a JAR file of your code, and transfer it to one of the cluster-access servers. Run the parser through the `hadoop` command to interact with the HDFS cluster:

```
hadoop jar ix-lab05.jar ix.lab05.processing.NetflixParser /ix/netflix/training_set
    dataset.tsv /path/to/my/ratings.tsv /path/to/jack.tsv
```

Save `dataset.tsv` somewhere on HDFS. When running the parser, don't forget to include your ratings, as well as Jack's ratings—there is no way to append them later on.

We also need the data in a sparse matrix form, in such a way that we will be able to perform linear algebra operations easily. This ratings matrix will be stored in a *sequence file* (read more about it in the appendix) and will have the following form. Each row (sequence element) has a key and a value; the key is the user ID and the value is a (sparse) vector containing the rating values for the movies he has rated (it can therefore be seen as a matrix of users × movies). Let's build this matrix with a MapReduce job!

- Complete the mapper function in `NetflixMatrixMapper.java` and the reducer function in `NetflixMatrixReducer.java`. Use the unit tests to check your implementation.
- Build the sequence file from `dataset.tsv` by running the job on the cluster. In order to ship the Mahout libraries to all the cluster's nodes, use the option `-libjars $MAHOUT_JAR` for your job—put it right after the name of the class to execute, before the other arguments.

```
hadoop jar ix-lab05.jar ix.lab05.processing.NetflixMatrix -libjars $MAHOUT_JAR
    dataset.tsv netflix_matrix
```

3 Similarity-based Recommender

In this part, you will implement a simple collaborative filtering recommender system that suggests movies based on a similarity measure between users. The similarity between two users is expressed as the distance between their rows in the ratings matrix R. To compute this distance, several measures can be used; a common distance measure is the *cosine distance* which computes the similarity between two users u and v as follows:

$$sim(u, v) = \frac{\mathbf{r}_u \cdot \mathbf{r}_v}{||\mathbf{r}_u|| ||\mathbf{r}_v||},$$

where $||\mathbf{r}_u||$ is the L2-norm of the row $\mathbf{r}_u$ of user u in the ratings matrix R.

> **Sparse matrix**: Note that R is a very sparse matrix, because most users have only rated a few movies. Thus, all matrix operations we mention here are only defined on non-zero entries of R.

To estimate the rating some user u would give to a movie i, we simply compute a weighted average of all ratings of movie i by the other users:

$$r_{u,i} = \frac{\sum_{\text{users } u' \neq u} sim(u', u) r_{u',i}}{\sum_{\text{users } u' \neq u} sim(u', u)}. \tag{1}$$

Note that the sums are over the users u' that have rated the movie i, not all users. To give some recommendations to a user, you thus simply have to compute the estimated rating $r_{u,i}$ for all movies i and then output the ones that have the highest estimated ratings.

3.1 Computing Ratings

Imagine you are given the following ratings matrix:

	movie1	movie2	movie3	movie4
user1	0	1	1	0
user2	2	1	1	3
user3	2	2	1	3

To recommend movies to `user1`, there are two steps required. First, we need to compute the similarity between `user1` and all other users. Second, we compute the estimated rating of each movie using the similarity of all other users and their ratings. This can be easily done using MapReduce:

- The mapper computes, for each user, its similarity with `user1`. It then outputs the similarity, along with all ratings of the user.

  Complete the class `RecommenderMapper` to implement this first operation. The method `VectorUtils.cosineSimilarity` will be useful. As an example, when given the ratings of `user2` as input, it should output the following:

  ```
  {similarity: 0.36515, ratings: {movie1: 2, movie2: 1, movie3: 1, movie4: 3}}
  ```

- The reducer then simply computes the weighted sum of all ratings, normalized by the sum of all similarities, to obtain the estimated ratings for `user1`. Again, for each movie, we only take into account users that have rated this movie when computing the weighted sum, not all users.

  Complete the class `RecommenderReducer` to have the reducer output the final estimated ratings for all movies, computed using Equation (1).

As usual, use the corresponding unit tests to check that your implementations are correct.

NullWritable as key? If you look closely at the MapReduce job you just implemented, you will notice that both input and output keys of the reducer are of type `NullWritable`. In the Hadoop lingo, this simply means that there are no keys, just values, and that all will be processed by one single reducer.

- Run the `Recommender` job you just implemented to recommend 20 movies to Jack (user ID: `12345679`). What is the number one recommendation?

  ```
  hadoop jar ix-lab05.jar ix.lab05.similarity.Recommender -libjars $MAHOUT_JAR
      netflix_matrix recommendations_jack /local/path/to/movie_titles.txt 12345679
  ```

java.lang.reflect.InvocationTargetException: If your job fails with a `RuntimeException`, make sure that you added `-libjars $MAHOUT_JAR` to the command line, to tell Hadoop to distribute the Mahour jar files.

- Run the job again to recommend 20 movies to yourself this time (user ID: `12345678`). Do the recommendations match your tastes?

3.2 Root mean squared error (RMSE)

One way of evaluating the performance of a recommender system is to measure the RMSE of its predicted ratings. In the Netflix dataset we provided, we intentionally removed some ratings for user `16355`. We will now compare the ratings predicted by the recommender system with the true ratings as given by that user[2]. Let N be the number of rated movies; given the estimated ratings $\hat{\mathbf{r}}$ and true ratings $\mathbf{r}$ for a user, the RMSE is defined as:

$$\text{RMSE}(\mathbf{r}, \hat{\mathbf{r}}) = \sqrt{\frac{\sum_{i=1}^{N}(r_i - \hat{r}_i)^2}{N}},$$

- Complete the function `evaluate()` in `RMSE.java`. This function computes the RMSE between predictions for user `16355` and his true ratings, found in `data/16355-truth.txt`. Using the class `Evaluator`, compute the RMSE of your recommender.

4 ALS-WR Algorithm

Now that you have built a simple memory-based recommender system that allowed you to understand the basics of collaborative filtering, we are going to look at a latent-factor technique that has been used recently with very good results: UV factorization.

> **Why UV factorization?** As the Netflix dataset has many missing values (only 1% of the movie ratings are observed, resulting in a sparse ratings matrix), we cannot apply the standard SVD algorithm. Instead, we use UV factorization (seen in the first Recommender Systems lecture as the *latent factor model*), that recovers U and V by alternatively minimizing the reconstruction error on U and V.

Finding the optimal UV factorization of a matrix is a difficult problem, and unfortunately it can't be solved as cleanly as SVD. One approach to the problem is to start with random matrices U and V. By alternatively optimizing each matrix (leaving the other fixed), we converge to a *good* solution—even though we do not get any guarantee of optimality.

To explore UV factorization as a technique for recommender systems, we will take advantage of the Apache Mahout project. It provides a distributed implementation of the *alternating-least-squares with weighted-λ-regularization* algorithm that runs on top of Hadoop[3]. This algorithm was actually developed by Zhou et al. [2008] in the context of the Netflix Prize, where it proved to be a very effective technique. You can find the paper on `http://goo.gl/YrUxB`.

In order to get a sense of how everything fits together, we will start by computing the UV factorization with matrices of rank 3. Run the following command:

```
mahout parallelALS --input <PATH> --output <PATH> \
  --tempDir als-tmp \  # Remove it after the run.
  --numFeatures 3 \
  --numIterations 10 \
  --lambda 0.065
```

- What is the meaning of each one of the arguments? Use the command's help, Google, and the paper presenting the algorithm Zhou et al. [2008]. *Hint: the format expected as input is tab-separated values. This should ring a bell...*

[2] Note that in a real setting, it is not clever to evaluate the RMSE over a *single* user. That particular user might be an outlier in the sense that predictions could be particularly easy—or particularly hard. By computing the RMSE over a test set of many users these effects usually dampen out.

[3] As an alternative to alternating-least-squares one might use *stochastic gradient descent*, an algorithm that you have seen in class. The former is however better suited to a parallel implementation

- *What is the role of* `-lambda`*? Explain why it is necessary, and how to can it be determined. Find and comment Fig. 1 in Zhou et al. [2008]. What would happen if lambda was set to* 0*?*
- *What is the output of the algorithm? Inspect the output directory. What letter is used to denote the matrix of item features, instead of* V*?*

To gain more intuition about the factorization and to link it with the last lab on dimensionality reduction, we are going to take a closer look at the matrix of item features.

- Run `FeaturesExtractor` to extract the item features matrix into a comma separated file. Just like for processing the dataset, export a JAR and launch it through the `hadoop` command on the cluster-access server.

```
hadoop jar ix-lab05.jar ix.lab05.factorization.FeaturesExtractor /hdfs/path/to/M >
    features.csv
```

- Using `scripts/plot3d.py`, explore how a selected set of movies are represented in this 3-dimensional feature space. Look up the movies on IMDB[4] if needed.

```
python scripts/plot3d.py features.csv data/movie_titles.txt
```

- Can you identify some clusters? For each of the dimensions, look at the movies which have extremal values—either very low or very high in a particular dimension. How would you describe them?

We will now turn our attention to building a recommender system based on this factorization. Although many variations exist, we will keep it simple, just like before: predict the ratings for all movies, and display the movies with the highest predicted rating.

- Complete the function `predictRating()` in `UVRecommender.java`. Use the unit tests to check your implementation.
- Run `UVRecommender` through `hadoop`. Generate 20 recommendations for Jack and for yourself. What do you think of the movies recommended to you?

Similarly to the first part, we can evaluate the performance of our recommender by computing the RMSE of the predicted ratings.

- Run `ix.lab05.factorization.Evaluator` for user `16355` through `hadoop`, using the ground truth provided in the file `data/16355-truth.txt`. What's the RMSE for that user?

Now that you have an understanding of how the algorithm works, both conceptually and concretely in terms of input, output and parameters, let us finally increase the number of features with the hope of improving the recommendations.

- Run the `parallelALS` command with the same iteration and regularization parameters, but this time with 25 features instead of 3.
- Based on this new factorization, generate 20 recommendations for you and for Jack. Do the recommendations change?
- Compute again the RMSE for user `16355`. Did it improve? If yes, by how much?

[4] `http://www.imdb.com/`

Appendix - Sequential File Format

We store the Netflix dataset in a sequential file format. SequenceFile is a flat file consisting of binary key/value pairs. It is extensively used in MapReduce as the input/output format, and for temporal outputs. There are 3 different SequenceFile formats:

- Uncompressed key/value records.
- Record compressed key/value records - only 'values' are compressed here.
- Block compressed key/value records - both keys and values are collected in 'blocks' separately and compressed. The size of the 'block' is configurable.

We use block compressed format to work with the Netflix dataset. The initial size of the dataset is 4.1 GB and the compressed size is 200MB. We also use SequentialAccessSparseVector structure, which is a vector that stores only non-zero values.

References

Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *Algorithmic Aspects in Information and Management*, pages 337–348. Springer, 2008.