Collaborative filtering with Spark

Charlotte Caucheteux[1*], Alexandre Momeni[1**]

[1]Centre de Mathématiques Appliquées, École Polytechnique

*charlotte.caucheteux@polytechnque.edu. ** alexandre.momeni@polytechnque.edu.

Collaborative filtering aims at learning predictive models of user preferences, interests or behavior from community data. In this paper, we report on our experience scaling up collaborative filtering using cloud computing and a Spark framework. In particular, we describe our approach for generating personalized recommendations using variants of the MovieLens data set. Our solution efficiently distributes computation in parallel and is relatively simple to implement.

Introduction

Collaborative filtering is a technology that aims at learning predictive models of user preferences, interests or behavior from community data, i.e. a database of available user preferences. The approach adopted in this paper is called probabilistic Latent Semantic Analysis (pLSA), a statistical model proposed by Hoffman (1999). In practise, this method requires complex computations on ever growing data sets, making scalability and parallelization essential.

We will first present an overview pLSA and introduce the Spark framework. We then present several pLSA implementations and present our results across variants of the MovieLens dataset. Finally, we draw conclusions and discuss future possible improvements.

Probabilistic Latent Semantic Analysis

The domains we consider consist of a set of users $U = u_1, ..., u_n$ and a set of movies S = $S = s_1, ..., s_m$. We assume observations will be just co-corrucences of u and s. We will focus our efforts on this simple co-occurence model, although we have implemented a multinomial model in the last section which includes numerical ratings. In our scenario, we can think of the preference data as being implicit. The data consists of a set of user-movie item pairs (u,s) which are assumed to be generated independently. We introduce the unobserved class variable Z with states z for every user-movie pair, so that user u and movie s are rendered conditionally independent. The possible set of states z is assumed to be finite and of size k. Formally, the model can be written in the form of a mixture model given by the equation:

$$p(s|u;\theta) = \sum_z p(z|u)\, p(s|z)$$

Following the maximum likelihood approach to statistical inference, we propose to fit the model parameters θ by maximizing the (conditional) log-likelihood. The Expectation Maximization (EM) algorithm is the standard method for statistical inference that can be used to maximize the log-likelihood in mixture models like pLSA.

Das (2007) outlines a method to parallelize the EM algorithm for computing pLSA parameters through the following the equations:

E-Step:

$$q^*(z;u,s;\hat{\theta}) = p(z|u,s;\hat{\theta}) = \frac{\hat{p}(z|u) \cdot \frac{N(z,s)}{N(z)}}{\sum_z \hat{p}(z|u) \cdot \frac{N(z,s)}{N(z)}}$$

M-Step:

$$N(z,s) = \sum_u q^*(z;u,s;\hat{\theta})$$

$$N(z) = \sum_s \sum_u q^*(z;u,s;\hat{\theta})$$

$$\hat{p}(z|u) = \frac{\sum_s q^*(z;u,s;\hat{\theta})}{\sum_z \sum_s q^*(z;u,s;\hat{\theta})}$$

Spark Framework

Spark is a cluster computing framework developed at Berkeley. It was explicitly designed to support iterative algorithms, such as EM, more efficiently than data flow frameworks like MapReduce. The programming model centers on parallel collections of objects called resilient distributed datasets (RDDs). Users can define RDDs from files in a storage system and transform them through data-parallel operations. However, unlike in these existing systems, users can also control the persistence of an RDD, to indicate to the system that they will reuse an RDD in multiple parallel operations. In this case, Spark will cache the contents of the RDD in memory on the worker nodes, making reuse substantially faster. At the same time, Spark tracks enough information about how the RDD was built to reconstruct it efficiently if a node fails. This in-memory caching is what makes Spark faster than MapReduce for iterative computations. Furthermore, Spark has a variety of features which can help improve parallelism and robustness (e.g broadcasts, accumulators) which we will explore below.

The Algorithms

Throughout the project, we developped three different algorithms to deal with the challenges of parallelism in the Spark Framework. A brief description of each algorithm and their rationale

is described below.

1. *Join*

The Join algorithm is a basic implementation of the EM algorithm described above. We construct each component of the quotients in the Maximization phase separately and use a series $.join()$ operations to compute the sufficient statistics. We then perform a final $.join()$ operation for the Expectation phase. This has high network traffic given the number of shuffles that need to take place and high computational cost.

2. *GroupBy*

The GroupBy algorithm replaces the $.join()$ operations in the Maximization phase with a $.groupByKey()$ which is performed locally. This avoids network traffic until the Expectation phase where we perform the $.join()$ operation. Whilst groupByKey() has this key advantage, it also considerably decreases parallelism in the Maxmisation phase since the keys are the communities. Indeed, for each key z, the value consists of a list which can contain up to the number of users and the number of movies for $N(z,s)/N(z)$ and $P(z|u)$ respectively. Furthermore, not only is the parallelism reduced, but the risk of breaks increases given local memory limitations. Note that we do not have this issue in the Expectation phase since each key has for value a list of size at most the number of communities, which remains small.

3. *Broadcast*

The Broadcast algorithms solves our concerns above. Indeed, we notice that the denominator for both the sufficient statistics $\frac{N(z,s)}{N(z)}$ and $p(z|u)$ only depends on z and thus is a small data set. As a reminder, in the GroupBy algorithm we compute these quantities with the following pseudo-code: PzuNorm = Pzu/D and NzsNorm = Nzs/D, where D only depends on z (more details in the accompanying notebook). To ensure the robustness of the Maximization phase, we thus make use of a broadcast variable - a built-in feature of Spark that allows efficient share read-only reference data across a Spark cluster. In particular, in this algorithm we will broadcast

the denominator D(z) and then compute the quotients PzuNorm and NzsNorm locally.

Test Data

We provide further details on our datasets for our comparative study. The MovieLens data set consists of movie and user data collected using a web-based research recommender system. We have created three MovieLens variants from the original dataset

1. Small - 1,000 lines; 15 users; 669 movies
2. Medium - 10,000 lines; 70 users, 3622 movies
3. Large - 100,000 lines; 671 users, 9066 movies

Evaluation Methodology And Results

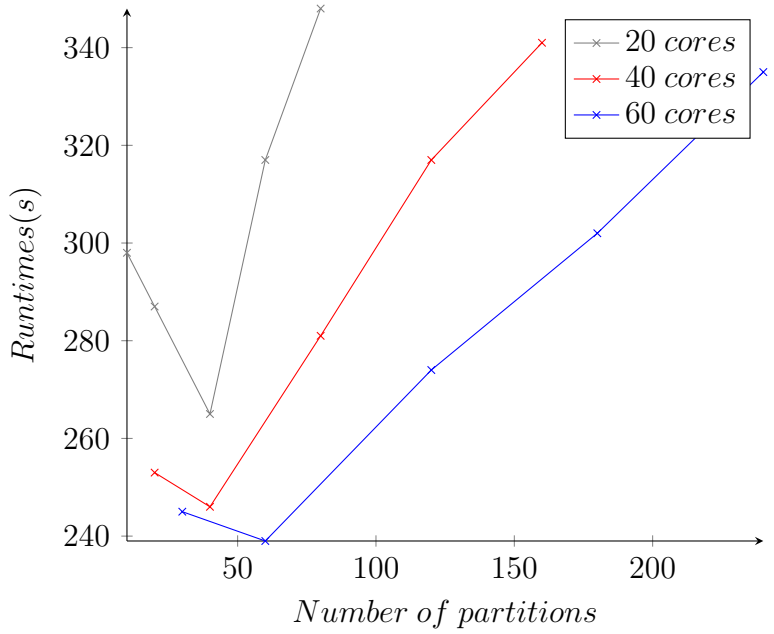
Our AWS EMR Instance comprises of one Master (m4.xlarge 8 vCore, 16 GB memory, EBS Storage:32 GB) and several Workers (m4 4 vCore, 8 GB memory, EBS Storage:32 GB).

We conduct the four steps below to assess the performance of our algorithms :

1. Varying the number of partitions: Spark Default number and variants (0.25x, 0.5x, 1x, 1.5x, 2x) using the $.coalesce()$ operation to re-partition at the end of each iteration
2. Varying the number of cores in the cluster: 5, 10 and 15 workers
3. Turning off caching
4. Varying the data set size

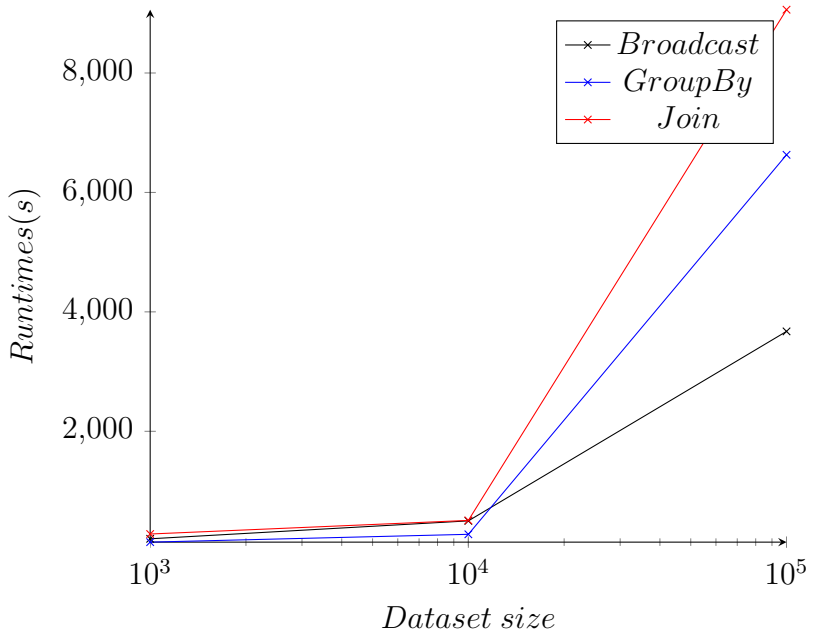
Given resource constraints, we focused on the medium data set for the first three and use the GroupBy algorithm. When we vary the data set size for different algorithms, we use 18 workers (the maximum possible on AWS). Throughout this section, we assume 20 EM iterations and 5 communities. The results for 1,2 and 4 are summarized in Figure 1 and Figure 2 below:

Fig. 1. *Runtimes (s) for different cluster configurations and partition sizes*



Remark: Taking half of Spark's default partition number seems to improve performance.

Fig. 2. *Running time on different data set sizes*



Remark: GroupyBy performs best for small data sets while Broadcast offers the best scalability

We also compared the run times of the EM algorithm without caching (i.e., reloading data on each iteration) and with in-memory caching. Without caching, the runtime was 1.3x slower just on the medium data set using GroupBy.

Multinomial

Since many applications of collaborative filtering involve explicit user ratings, the pLSA model needs to be generalized appropriately. Notable approaches include taking a Normal or Multinomial prior. Given we focus on large scale applications, we will use the later. Indeed, we can round ratings to the nearest integer value as to discretise the distribution and limit the number of total possible ratings. Formally, we can extend the model to the following mixture model

$$p(v|u, s; \theta) = \sum_z p(v|s, z)\ p(z|u)$$

The algorithm changes only very slightly

1. We will weigh the probability $P(z|u, s)$ by the corresponding rating $n(s, u)$, instead of 1.

2. We will normalize $P(z)$ by the sum of all ratings

3. We account for the rating in the calculation of q where it take new value $P(z, u, s) \cdot n(u, s)$.

Where $n(u, s)$ is the rating of user u for the movie s.

E-step :

$$P(z|s, u) = \frac{P(z) * P(u|z) * P(s|z)}{\sum_z P(z) * P(u|z) * P(s|z)}$$

$$Q^*(z|u, s) = P(z|u, s) * n(s, u)$$

M-step :

$$p(u|z) = \frac{\sum_s Q^*}{\sum_{s,u} Q^*}$$

$$p(s|z) = \frac{\sum_u Q^*}{\sum_{s,u} Q^*}$$

$$p(z) = \frac{\sum_{s,u} Q^*}{\sum_{s,u} n(s,u)}$$

Conclusion

We have presented our experience scaling up collaborative filtering in the cloud and identified lessons that we believe useful for any mixture based model using the EM algorithm. Our project affirmed the value of in-memory computation for iterative algorithms, but also highlighted the importance of broadcasting and efficient bandwidth utilization, and partition size for performance optimization. We have also implemented a multinomial model which can include rating data, which we hope to use for large scale prediction in future work.