Distributed Bayesian Personalized Ranking in Spark

Alfredo Láinez Rodrigo
Stanford University
alainez@stanford.edu

Luke de Oliveira
Stanford University
lukedeo@stanford.edu

Abstract

Bayesian Personalized Ranking (BPR) is a general learning framework for item recommendation using implicit feedback (e.g. clicks, purchases, visits to an item), by far the most prevalent form of feedback in the web. Using a generic optimization criterion based on the maximum posterior estimator derived from a Bayesian analysis, BPR differentiates itself from other common recommendation algorithms in two main aspects. First, it directly optimizes the final objective of ranking. Second, it abstracts away the underlying rating computation model (which could be, for instance, matrix factorization or k-nearest-neighbors). In this paper, we define a distributed version of BPR using matrix factorization for the Spark ecosystem, which we implement in both Scala and Python (`https://github.com/alfredolainez/bpr-spark`).

1 Introduction

Recommender systems currently play a central role in the modern Internet. As a major factor for user engagement and revenue, much of the content seen on the web is presented as a personalized recommendation of some form. Furthermore, the easy availability of data used by recommender systems makes them a perfect example of high-availability systems that need to scale to multiple computers.

Following the widely-used recommendation module in Apache Mahout, Spark included its own version of alternating least squares from its inception. Alternating least squares (ALS) [HKV08] is the *de facto* standard method for building collaborative filtering recommender systems for massive amounts of data. Stemming from the popularity of matrix factorization methods after the Netflix prize, the algorithm succeeded due to its easy parallelization and the simple treatment of implicit feedback (implicit feedback, as opposed to explicit user-provided ratings, is easier to obtain and commonly regarded as more indicative of real user interests). ALS reduces a non-convex formulation to a series of iterations alternating easy quadratic problems. Spark's memory usage maps nicely into the paradigm of iterative machine learning algorithms, and so the ALS recommendation module of *mllib* has seen widespread industry adoption, while methods based on *MapReduce* are starting to be discontinued.

Bayesian Personalized Ranking [Ren+09] also works naturally with implicit feedback. In the item recommendation problem, we are usually interested in creating a user-specific ranking of items that the user might be interested in. Alternating least squares, for instance, tries to predict a rating for each user and item by factoring a rating matrix into two matrices consisting of user and item vectors, whose elements correspond to "latent" factors that explain user preferences. A ranking can then be obtained by decreasingly ordering the predicted ratings. On the contrary, BPR directly optimizes a ranking measure $>_u$ for items that is effectively translated to learning the parameters of an underlying recommendation model. In particular, BPR provides a generic, model-agnostic optimization criterion derived from a Bayesian analysis of the ranking problem. This can then be learned via stochastic gradient descent over bootstrapped samples of triples *(user, item seen, item not seen)*.

While BPR can be used with different recommendation system schemes (in the original paper, matrix factorization and kNN are shown), we have settled on using MF (matrix factorization). Our solution in Spark distributes all the available feedback across the cluster, where each machine is responsible for a fraction of the ratings and the users. Parallel stochastic gradient descent is performed to learn the parameters of MF, which are a user matrix and an item matrix. The former tends to be much bigger and so it is distributed among the cluster. The latter, normally small, is shared across machines for reasons that will be explained in section 3.

Finally, note that the success and widespread use of alternating least squares in the context of massive datasets is not by chance. This work does not try to outperform the already high scalability of the ALS implementation in Spark, but rather provide access to a different technique that can benefit from a distributed framework and can thus be applied in a scalable way.

2 Bayesian Personalized Ranking from Implicit Feedback

In this section, we describe BPR as explained in [Ren+09]. We are interested in the task of providing a user with a ranked list of recommended items. Hence, we want to learn a personalized ordering $>_u$ for each user u. We consider U the set of all users and I the set of all items. The source of information is implicit feedback, this is, past interactions of users with items, which we formalize as $S \subseteq U \times I$. We assume that $>_u$ meets the properties of a total order (totality, antisymmetry and transitivity). We also define $I_u^+ := \{i \in I : (u, i) \in S\}$ and $U_i^+ := \{u \in U : (u, i) \in S\}$.

In order to learn $>_u$, we need to reconstruct parts of it from the available data. We assume that if $(u, i) \in S$, this is, user u has viewed item i, then u prefers this item over all other non-observed items. We do not assume any preference for items both viewed or both non viewed. Hence, we can define our training set D_S as the set of all $(user, item+, item-)$ triples:

$$D_S := \{(u, i, j) | i \in I_u^+ \wedge j \in I \setminus I_u^+\}$$

We derive now the optimization criterion. Consider Θ the set of parameters of the underlying model class (for MF, a user and item matrix). The Bayesian formulation of finding $>_u$ is to maximize the posterior probability

$$p(\Theta| >_u) \propto p(>_u |\Theta)p(\Theta)$$

BPR makes some key assumptions. First, it assumes that users act independently of each other. Second, that the ordering of each pair of items is independent of the ordering of the other pairs. Hence, we can write

$$\begin{aligned} p(>_u |\Theta) &= \prod_{u \in U} p(>_u |\Theta) \\ &= \prod_{(u,i,j) \in U \times I \times I} p(i >_u j|\Theta)^{\delta((u,i,j) \in D_S)} \cdot (1 - p(i >_u j|\Theta)^{\delta((u,j,i) \notin D_S)}) \\ &= \prod_{(u,i,j) \in D_S} p(i >_u j|\Theta) \end{aligned}$$

where δ is the indicator function and we have used antisymmetry and totality of $>_u$ to arrive at the last step. Finally, it is necessary to model $p(i >_u j|\Theta)$ in a way that it is a probability and the total order is respected. BPR defines this as

$$p(i >_u j|\Theta) := \sigma(\hat{x}_{uij}(\Theta))$$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the well-known sigmoid function and $\hat{x}_{uij}$ is a real-valued function of the model parameter Θ which captures in a quantitative way the preference that user u has for item i over j. For instance, for the case of matrix factorization as the underlying model, we define

$$\hat{x}_{uij} := \hat{x}_{ui} - \hat{x}_{uj}$$

where $\hat{x}_{ui}$ is the specific rating predicted for user u and item i. In MF, we are estimating an $|U| \times |I|$ rating matrix X by using two low-rank matrices, W (with size $|U| \times k$) and H (with size $|I| \times k$), so that $\hat{X} = WH^t$, with k the number of latent factors. Since the rows of W and H are seen as feature vectors in the latent space, we have

$$\hat{x}_{uij} := \hat{x}_{ui} - \hat{x}_{uj} = w_u \cdot h_i - w_u \cdot h_j$$

where w_i is the ith row of W and h_j is the jth row of H.

To complete the Bayesian analysis, we define our prior $p(\Theta)$ as a normal distribution with zero mean and covariance matrix Σ_Θ, where we set $\Sigma_\Theta = \lambda_\Theta I$ for convenience

$$p(\Theta) \sim N(0, \Sigma_\Theta)$$

Putting all together, our optimization objective is then

$$\begin{aligned} BPR - OPT :&= ln\, p(\Theta| >_u) \\ &= ln\, p(>_u |\Theta) p(\Theta) \\ &= ln \prod_{(u,i,j) \in D_S} \sigma(\hat{x}_{uij}) p(\Theta) \\ &= \sum_{(u,i,j) \in D_S} ln\, \sigma(\hat{x}_{uij}) - \lambda_\Theta \|\Theta\|^2 \end{aligned}$$

2.1 BPR - Learning

BPR has a differentiable objective, and so gradient descent can be used for maximization. However, D_S has $O(|S| \cdot |I|)$ training triples, which can make each traversal through the data infeasible. Hence, stochastic gradient descent (SGD) seems to be a more sensible choice. Since originally we only have positive feedback elements (this is, pairs user-item (u, i)), we need to bootstrap triples from D_S before performing the optimization. This yields a simple learning procedure consisting of drawing training triplets from D_S and performing gradient updates until convergence.

For each triplet (u, i, j), the stochastic gradient descent update is given by

$$\Theta \leftarrow \Theta + \alpha \left(\frac{e^{-\hat{x}_{uij}}}{1 + e^{-\hat{x}_{uij}}} \cdot \frac{\partial}{\partial \Theta} \hat{x}_{uij} + \lambda_\Theta \Theta \right)$$

where in the case of matrix factorization

$$\frac{\partial}{\partial \theta} \hat{x}_{uij} = \begin{cases} (h_{if} - h_{jf}) & \text{if } \theta = w_{uf} \\ w_{uf} & \text{if } \theta = h_{if} \\ -w_{uf} & \text{if } \theta = h_{jf} \\ 0 & \text{else} \end{cases}$$

3 Distributed Bayesian Personalized Ranking - DBPR

Our implementation of BPR distributes the data and the user matrix across the cluster. This allows the algorithm to scale both in the number of positive ratings and the number of users to model. The item matrix cannot be distributed around the cluster and must sit in memory in every machine for reasons that will be clear soon. However, this is not a major drawback since we normally face situations where the number of items is significantly smaller than the number of users (consider for instance an online retailer that needs to recommend a fixed number of products to millions of users).

Algorithm 1 Distributed Bayesian Personalized Ranking

```
1: procedure DBPR(Array[Tuple](user, item) $feedback$)
2:     randomly initialize item matrix $H$
3:     broadcast $H$ to all workers
4:     randomly initialize user matrix $W$
5:     group ($W$, $feedback$) by user and distribute across cluster
6:     for $i = 1 \rightarrow numIterations$ do
7:         for $worker$ in parallel do
8:             for $j = 1 \rightarrow numSamples$ do
9:                 sample triplet from local $D_S$
10:                update distributed $W$ and local $H$
11:            end for
12:        end for
13:        reduce $H$ from workers and compute new $H$ in driver as the average
14:        broadcast $H$ to all workers
15:    end for
16:    return ($W$, $H$)
17: end procedure
```

The algorithm can be seen in Algorithm 1. First, the ratings (feedback in the form $(user, item)$) is grouped by user and distributed across the workers, so that every machine is responsible of updating certain user vectors. Considering that we may have an enormous number of users and feedback, and that we need to sample negative elements for every user to effectively learn its ranking, it makes sense to parallelize this bulky stochastic gradient descent process, with the advantage that different machines can focus on different users with no overlap. Note that since we will sample many negative items per user, and since each worker is responsible of many users, we will have that each machine will update most of the item vectors (in expectation) after SGD. Hence, it is not really advantageous to come up with an optimized distributed update of item blocks such as that performed by ALS.

The fact that the item matrix is not distributed makes it necessary for the algorithm to reconvene from time to time to average the different item matrix updates performed by the workers. Hence, DBPR runs in an iterative way, where at the end of each iteration a reduce process is performed to compute the new averaged item matrix, which is then sent back to every worker in the next iteration.

Hence, for each iteration we will have a communication overhead in the network of $O(m \cdot |I|k)$, where m is the number of machines and k the number of latent factors. We will see however that the algorithm does not need many iterations for convergence. Nevertheless, since this communication cost might be dominant, it is in the best of our interest to perform as many SGD iterations in each worker before reducing and recomputing the item matrix after each iteration.

Inside each worker, the number of samples from D_S to perform SGD over is a parameter of the algorithm. In order to obtain samples in a fast way, DBPR does not check that the negative item is actually an item that the user did not view. Doing this in an effective way would require extra and costly data structures (i.e., many expensive lookups), while the probability of a collision is normally negligible (normally recommender systems deal with thousands of items, while users have interactions with only a tiny fraction of them). Furthermore, the action of selecting two positive items i and j and sampling them as $i >_u j$ does not break any assumption of the algorithm since we have no information about which item the user might actually prefer. In our experiments, checking that the negative sampling is actually negative had no effect at all in both convergence rate and final results.

4 Evaluation

In order to evaluate our results, we compare learning with DBPR to the default ALS recommender provided in Spark.

4.1 Dataset

We utilize the *Million Song Dataset Challenge* dataset [Ber+11]. This dataset contains the listening history of 110,000 users with the objective of recommending songs that the users like. Although the challenge is originally a prediction one (trying to predict what songs the users actually listened to, hidden in a holdout set), we use the original training data of the challenge for both training our model and creating a separate test set to evaluate our recommendations. The original data contains the number of times each song was played by the user, and although this information could be used to better learn interest, we restrict ourselves to the original setting of the paper where the feedback consists of just the presence of an interaction with an item.

Similarly to the evaluation procedure followed in [Ren+09], we draw a subsample from the original dataset such that every user has at least 10 items ($\forall u \in U : |I_u^+| \geq 10$) and each item has at least 10 users ($\forall i \in I : |U_i^+| \geq 10$).

4.2 Evaluation Methodology

As evaluation methodology, we utilize AUC (area under the curve) as a metric over a test set S_{test}. For each user, we remove a positive entry from I_u^+ and add it to S_{test}, while the rest of entries are used for training in S_{train}. DBPR is then run over S_{train} and the resulting personalized ranking is evaluated with the average AUC statistic over all users:

$$\text{AUC} = \frac{1}{|U|} \sum_u \frac{1}{|E(u)|} \sum_{(i,j) \in E(u)} \delta(\hat{x}_{ui} > \hat{x}_{uj})$$

where δ is the indicator function and

$$E(u) = \{(i,j) | (u,i) \in S_{test} \wedge (u,j) \notin (S_{test} \cup S_{train})\}$$

The metric is measuring, for each item we know a user has interacted with, what fraction of times it would be ranked before all the elements he or she did not interact with. Hence, random guessing would return an AUC of 0.5, while the highest score achievable is 1.

4.3 Results and discussion

In Figure 1 we can see the results of the algorithm using different numbers of samples from D_S per worker and iteration. This parameter is critical since it heavily affects the number of iterations needed until convergence. We utilize 4 workers in Spark standalone mode in all our experiments. We can see how a regime of 20,000 samples per worker and iteration takes a long time to converge, although it obtains decent results from iteration 1. On the other hand, increasing the number of samples per worker and iteration to 2,000,000 makes the algorithm achieve convergence in the first iteration, practically becoming parallel stochastic gradient descent [Zin+10]. Although we anticipated, due to the interdependency of users and items in collaborative filtering models, that different workers would need gradient updates of the items in different machines, it might well be that the independence assumptions of BPR effectively break this interdependency. Further studies would be needed to check if this is always the case, which would allow the algorithm to run in just one iteration.

In any case, it is clear that by setting a high enough number of samples, we can achieve convergence in a small number of iterations, thus allowing for fewer H matrices sent through the network. Also, we note that BPR consistenly obtains a better AUC result (around 0.73) than ALS upon convergence.

We also provide a cursory investigation into the effect of the latent dimension of the factorization on the final AUC metric, as can be seen in Figure 2. In particular, we fix a number of iterations, 10, fix a size of $D_S = 10,000$, and examine the effect on final AUC as a function of varying k. Note that BPR peaks at a slightly higher k, but still within the realm of what is considered standard in industrial settings. Both ALS and BPR need a big enough number of factors to explain the variability in the data, while we also note similar overfitting behavior when the rank is increased too much.

Finally, due to the fact that we tested our distributed version on one multi-core machine, we do not present any empirical studies of communication times in comparison to ALS, and we leave that to further studies. In particular, the authors are interested in comparing computation time versus $|D_S|$ and the number of partitions (for BPR) against the number of blocks in ALS in a cluster setting. It is to expect however that ALS will run faster and with less network usage due to the fact that the inner loop can be distributed across both the users and items with minimal communication cost.

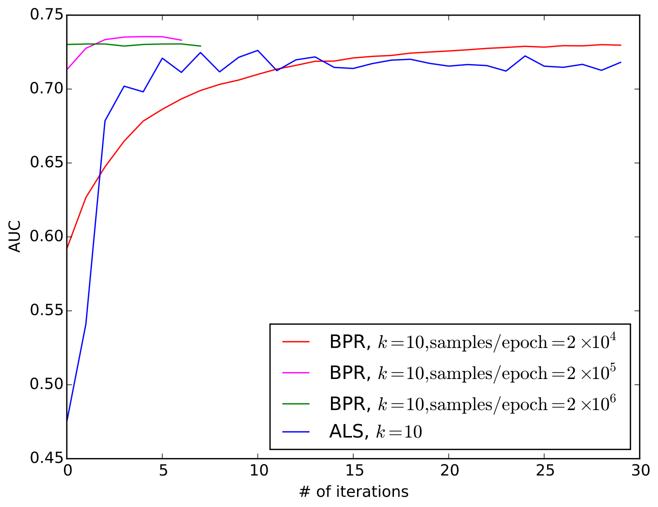

Figure 1: Test AUC (Area Under Curve) as a function of k (latent dimension) and the number of considered ratings per iteration and worker

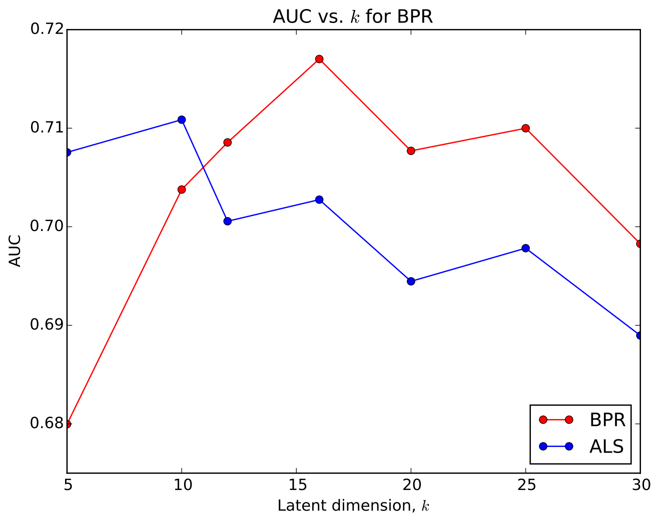

Figure 2: Test AUC (Area Under Curve) as a strict function of k (latent dimension)

5 Conclusion

We have developed a distributed version of Bayesian Personalized Ranking, a model-agnostic recommendation framework that directly learns a personalized ranking for every user from implicit feedback. We have implemented it by using an underlying matrix factorization model and allowing different workers to perform sampling and stochastic gradient descent in a parallel way. Furthermore, the algorithm easily scales to millions of users due to the fact that the user matrix is distributed across the cluster. Our version of BPR obtains better results than Spark's default ALS recommender in an offline evaluation. However, it is important to note that our model needs the item matrix to be present in memory in every worker, which is a very reasonable assumption but does not facilitate high scalability in the number of items.

References

[HKV08] Yifan Hu, Yehuda Koren, and Chris Volinsky. “Collaborative filtering for implicit feedback datasets”. In: *Data Mining, 2008. ICDM’08. Eighth IEEE International Conference on*. Ieee. 2008, pp. 263–272.

[Ren+09] Steffen Rendle et al. “BPR: Bayesian personalized ranking from implicit feedback”. In: *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*. AUAI Press. 2009, pp. 452–461.

[Zin+10] Martin Zinkevich et al. “Parallelized stochastic gradient descent”. In: *Advances in neural information processing systems*. 2010, pp. 2595–2603.

[Ber+11] Thierry Bertin-Mahieux et al. “The Million Song Dataset”. In: *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*. 2011.