Scalable Similarity-Based Neighborhood Methods with MapReduce

Sebastian Schelter Christoph Boden Volker Markl

Technische Universität Berlin, Germany
firstname.lastname@tu-berlin.de

ABSTRACT

Similarity-based neighborhood methods, a simple and popular approach to collaborative filtering, infer their predictions by finding users with similar taste or items that have been similarly rated. If the number of users grows to millions, the standard approach of sequentially examining each item and looking at all interacting users does not scale. To solve this problem, we develop a MapReduce algorithm for the pairwise item comparison and top-N recommendation problem that scales linearly with respect to a growing number of users. This parallel algorithm is able to work on partitioned data and is general in that it supports a wide range of similarity measures. We evaluate our algorithm on a large dataset consisting of 700 million song ratings from Yahoo! Music.

Categories and Subject Descriptors

H.4.m [**Information Systems Applications**]: Miscellaneous

Keywords

Scalable Collaborative Filtering, MapReduce

1. INTRODUCTION

Today's internet users face an ever increasing amount of data, which makes it constantly harder and more time consuming to pick out the interesting pieces of information from all the noise. This situation has triggered the development of recommender systems: intelligent filters that learn about the users' preferences and figure out the most relevant information for them.

With rapidly growing data sizes, the processing efficiency and scalability of the systems and their underlying computations becomes a major concern. In a production environment, the offline computations necessary for running a recommender system must be periodically executed as part of larger analytical workflows and thereby underly strict time and resource constraints. For economic and operational reasons it is often undesirable to execute these offline computations on a single machine: this machine might fail and with growing data sizes constant hardware upgrades might be necessary to improve the machine's performance to meet the time constraints. Due to these disadvantages, a single machine solution can quickly become expensive and hard to operate.

In order to solve this problem, recent advances in large scale data processing propose to run data-intensive, analytical computations in a parallel and fault-tolerant manner on a large number of commodity machines. Doing so will make the execution independent of single machine failures and will furthermore allow the increase of computational performance by simply adding more machines to the cluster, thereby obviating the need for constant hardware upgrades to a single machine. Another economic advantage of such an approach is that the cluster machines can be temporally rented from a cloud computing infrastructure provider.

When applied to recommender systems, this technical approach requires the rephrasing of existing algorithms to enable them to utilize a parallel processing platform. Such platforms are able to run on a cluster of up to several thousand machines and to store and process amounts of data that were previously considered unmanageable. They typically employ a shared-nothing architecture together with a parallel programming paradigm and store the data in replicated partitions across the cluster. They provide the desired horizontal scalability when the number of machines in the cluster is increased. Furthermore they relieve the programmer from having to cope with the complicated tasks of scheduling computation, transferring intermediate results and dealing with machine failures.

We rephrase and scale out the similarity-based neighborhood methods, a standard approach in academic literature [22]. They have the advantage of being simple and intuitive to understand, as they are directly inspired by recommendation in everyday life, where we tend to check out things we heard about from like-minded friends or things that seem similar to what we already like. They capture local associations in the data which increases serendipity [22] and they are necessary as part of ensembles to reach optimal prediction quality [5]. The item-based variants [23] of the neighborhood methods are highly stable and allow computation of recommendations for new users without the need to rebuild the model. Additionally, they are able to provide instant justifications for their recommendations by presenting the list of neighbor items and the ratings the user already gave to these as explanation. Due to these properties, neighbor-

RecSys'12, September 9–13, 2012, Dublin, Ireland.

hood methods are often preferred in industrial use cases [19, 1, 24, 7], although alternative approaches such as the latent factor models are superior in the task of predicting ratings.

We improve the scalability of the similarity-based neighborhood methods by rephrasing the underlying algorithm for pairwise comparisons to MapReduce [8], a popular parallel programming paradigm that has originally been proposed by Google. We demonstrate our approach using Apache Hadoop [2], a widely used, open source platform which implements the MapReduce paradigm.

We already contributed an implementation of the approach presented here to **Apache Mahout** [3], an open-source library of scalable data mining algorithms, where it forms the core of the distributed recommender module.

In this paper, we provide the following contributions:

- We introduce an algorithmic framework that allows scalable neighborhood-based recommendation on a parallel processing platform.
- We describe how to implement a variety of similarity measures in a highly efficient manner in our framework.
- We discuss how to apply selective down-sampling to handle scaling issues introduced by the heavy tailed distribution of user interactions commonly encountered in recommendation mining scenarios.
- We present experiments on various datasets with up to 700 million user interactions.

This paper is organized as follows: After a brief introduction to the MapReduce paradigm for parallel processing, we describe the algorithmic challenges of our approach in Section 2 and related work in Section 3. We describe and in detail derive our algorithm in Section 4. Finally, we evaluate our solution on various datasets in Section 5.

1.1 MapReduce

MapReduce [8], which is inspired by functional programming, has become a popular paradigm for data-intensive parallel processing on shared-nothing clusters.

The data to process is split and stored block-wise across the machines of the cluster in a distributed file system (DFS) and is usually represented as (*key,value*) tuples. In order to efficiently parallelize the computation and offer tolerance against machine failures, data is replicated across the cluster. As the computation tasks should be moved to the data, the runtime system assigns tasks to process data blocks to the machines holding the replicas of these blocks. The computation code is embedded into two functions:

map: $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$
reduce: $(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)$

The data flow in a MapReduce pass is illustrated in Figure 1. At the beginning the **map** function is invoked on the input data in parallel on all the participating machines in the cluster. The output tuples are grouped (partitioned and sorted) by their key and then sent to the reducer machines in the **shuffle** phase. The receiving machines merge the tuples and invoke the **reduce** function on all tuples sharing the same key. The output of that function is written to the distributed file system afterwards.

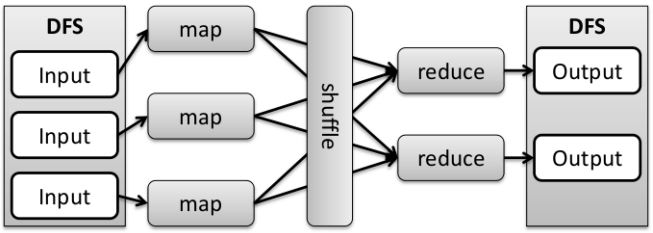

Figure 1: Illustration of the data flow in MapReduce

An optional third function called **combine** can be specified. It is invoked locally after the map phase and can be used to preaggregate the tuples in order to minimize the amount of data that has to be sent over the network, which is usually the most scarce resource in a distributed environment.

combine: $(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)$

In addition, Hadoop offers *initialize* functions that are invoked before the map and reduce functions and the system provides a means to broadcast small files to all worker machines in the cluster via a *distributed cache*.

2. PROBLEM STATEMENT

Let A be a $|U| \times |I|$ matrix holding all known interactions between a set of users U and a set of items I. A user u is represented by his item interaction history $a_{u\bullet}$, the u-th row of A. The top-N recommendations for this user correspond to the first N items selected from a ranking r_u of all items according to how strongly they would be preferred by the user. This ranking is inferred from patterns found in A.

2.1 Computational model

Notation hints: $a_{u\bullet}$ denotes the u-th row of the interaction matrix A, $a_{\bullet i}$ denotes the i-th column of A, $|U|$ denotes the number of users which is equal to the number of rows in A. $foreach\ i \in v$ denotes iteration over the indexes of non-zero entries of a vector v, $foreach\ (i,k) \in v$ denotes iteration over the indexes and the corresponding non-zero values of a vector v.

In order to get a clearer picture of the neighborhood approach, it is useful to express the algorithm in terms of linear algebraic operations. Neighborhood-based methods find and rank items that have been preferred by other users who share parts of the interaction history $a_{u\bullet}$. Let A be a binary matrix with $A_{ui} = 1$ if a user u has interacted with an item i and $A_{ui} = 0$ otherwise. For pairwise comparison between users, a dot product of rows of A gives the number of items that the corresponding users have in common. Similarly, a dot product of columns of A gives the number of users who have interacted with both items corresponding to the columns.

When computing recommendations for a particular user with *User-Based Collaborative Filtering* [21], first a search for other users with similar taste is conducted. This translates to multiplying the matrix A by the user's interaction history $a_{u\bullet}$, which results in a ranking of all users. Secondly, the active user's preference for an item is estimated by computing the weighted sum of all other users' preferences for this item and the corresponding ranking. In our

simple model this translates to multiplying the ranking of all users with A^{T}. This means the whole approach can be summarized by the following two multiplications:

$$r_u = A^{\mathrm{T}}(A\, a_{u\bullet})$$

To exploit the higher stability of relations between items, another variant of the neighborhood methods called *Item-Based Collaborative Filtering* [23] was developed, which looks at items first and weighs their cooccurrences. This approach computes a matrix of item-to-item similarities and allows for fast recommendations as the model does not have to be recomputed for new users. Expressing the item-based approach translates to simply moving the parentheses in our formula, as $A^{\mathrm{T}}A$ gives exactly the matrix of the item cooccurrences:

$$r_u = (A^{\mathrm{T}}A)\, a_{u\bullet}$$

This shows that both user- and item-based collaborative filtering share the same fundamental computational model. In the rest of the paper, we will focus on the more popular item-based variant.

2.2 Sequential approach

The standard sequential approach [19] for computing the item similarity matrix $S = A^{\mathrm{T}}A$ is shown in Algorithm 1.

Algorithm 1: sequential approach for computing item cooccurrences

```
foreach item i do
    foreach user u who interacted with i do
        foreach item j that u also interacted with do
            S_ij = S_ij + 1
```

For each item i, we need to look up each user u who interacted with i. Then we iterate over each other item j from u's interaction history and record the cooccurrence of i and j. We can mathematically express the approach using three nested summations:

$$S = A^{\mathrm{T}}A = \sum_{i=1}^{|I|} \sum_{u=1}^{|U|} \sum_{j=1}^{|I|} A^{\mathrm{T}}_{i,u} \cdot A_{u,j}$$

If we wish to distribute the computation across several machines on a shared-nothing cluster, this approach becomes infeasible as it requires random access to both users and items in its inner loops. Its random access pattern cannot be realized efficiently when we have to work on partitioned data.

Furthermore, at a first glance, the complexity of the item-based approach is quadratic in the number of items, as each item has to be compared with every other item. However, the interaction matrix A is usually very sparse. It is common that only a small fraction of all cells are known[1] and that the number of non-zero elements in A is linear in the number of rows of A. This fact severely limits the number of item pairs to be compared, as only pairs that share at least one interacting user have to be taken into consideration. It decreases the complexity of the algorithm to quadratic in the number of non-zeros in the densest row rather than quadratic in the number of columns. The cost of the algorithm is expressed as the sum of processing the square of the number of interactions of each single user. Unfortunately, collaborative filtering datasets share a property that is common among datasets generated by human interactions: the number of interactions per user follows a heavy tailed distribution which means that processing a small number of 'power users' dominates the cost of the algorithm.

We will develop a parallelizable formulation of the computation to scale out this approach on a parallel processing platform. As we need to utilize a distributed filesystem, our algorithm must be able to work with partitioned input data. Furthermore, it must scale linearly with respect to a growing number of users. We will enable the usage of a wide range of similarity measures and add means to handle the computational overhead introduced by 'power users'.

[1] in the datasets we used for our experiments, this ratio varies from 0.1% to 4.5%

3. RELATED WORK

Most closely related to our work is a MapReduce formulation of item-based collaborative filtering presented by Jiang et al. [17]. However they do not show how to use a wide variety of similarity measures, they do not achieve linear scalability as they undertake no means to handle the quadratic complexity introduced by 'power users' and only present experiments on a small dataset. Another distributed implementation of an item-based approach is used by Youtube's recommender system [7], which applies a domain specific way of diversifying the recommendations by interpreting the pairwise item similarities as a graph. Unfortunately this work does not include details that describe how the similarity computation is actually executed other than stating it uses a series of MapReduce computations walking through the user/video graph. Furthermore, a very early implementation of a distributed item-based approach was applied in the recommendation system of the TiVo set-top boxes [1], which suggests upcoming TV shows to its users. In a proprietary architecture, show correlations are computed on the server side and preference estimation is afterwards conducted on the client boxes using the precomputed correlations.

There have also been several works on parallelizing latent factor models: The recommender system of Google News [6] uses a MapReduce based implementation combining Probabilistic Latent Semantic Indexing and a neighborhood approach with a distributed hashtable that tracks item cooccurrences in realtime. This solution is tailored towards the outstanding infrastructure of Google and might not be practical in other scenarios. Another parallelizable implementation of a latent factor model was presented by Zhou et al. [26], where a factorization of the Netflix dataset using Alternating Least Squares is conducted. Mahout [3] contains a MapReduce port of this approach. Similarly, Gemulla et al. [15] propose a stratified version of Stochastic Gradient Descent for matrix factorization on MapReduce. Due to Hadoop's inability to efficiently execute iterative algorithms, these implementations show unsatisfactory performance. Dataflow systems with explicit iteration support such as Stratosphere [12] or specialized systems for machine learning such as GraphLab [20] will pose a solution for the efficient distributed execution of such algorithms in the near future.

4. ALGORITHM

This section discusses the step-by-step development of our algorithmic framework. We start with showing how to conduct distributed item cooccurrence counting for our simple model that uses binary data. After that we generalize the approach to non-binary data and enable the usage of a wide variety of similarity measures. Finally, we discuss means to sparsify the similarity matrix, conduct batch recommendation and apply selective down sampling to achieve linear scalability with a growing number of users.

4.1 Counting item cooccurrences

In order to scale out the similarity computation from Algorithm 1, it needs to be phrased as a parallel algorithm, to make its runtime speedup proportional to the number of machines in the cluster. This is not possible with the standard sequential approach, as it requires random access to the rows and columns of A in the inner loops of Algorithm 1, which cannot be efficiently realized in a distributed, shared-nothing environment where the algorithm has to work on partitioned data.

We need to find a way of executing this multiplication that is better suited to the MapReduce paradigm and has an access pattern that is compatible to partitioned data. The solution is to rearrange the loops of Algorithm 1 to get the row outer product formulation of matrix multiplication. Because the u-th column of A^{T} is identical to the u-th row of A, we can compute S with only needing access to the rows of A:

$$S = A^{\mathrm{T}}A = \sum_{u=1}^{|U|}\sum_{i=1}^{|I|}\sum_{j=1}^{|I|} A^{\mathrm{T}}_{i,u} \cdot A_{u,j} = \sum_{u=1}^{|U|} a_{u\bullet}(a_{u\bullet})^{\mathrm{T}}$$

Following this finding, we partition A by its rows (the users) and store it in the distributed file system. Each map function reads a single row of A, computes the row's outer product with itself and sends the resulting intermediary matrix row-wise over the network. The reduce function simply has to sum up all partial results, thereby computing a row of S per invocation (Algorithm 2).

This approach allows us to exploit the sparsity of the intermediary outer product matrices by making the map function only return non-zero entries. At the same time we apply a combiner (which is identical to the reducer) on the vectors emitted by the mappers, which makes the system minimize the amount of data that has to be sent over the network. Additionally we only compute the upper triangular half of S, as the resulting similarity matrix is symmetric.

4.2 Generalized similarity computation

Real world datasets contain richer representations of the user interactions than a simple binary encoding. They either consist of *explicit feedback* like numerical ratings that the users chose from a predefined scale or of *implicit feedback* where we count how often a particular behavior such as a click or a page view was observed. We need to be able to choose from a variety of similarity measures for comparing these item interactions, in order to be able to find the one that best captures the relationships inherent in the data. From now on, we drop the assumption that A contains only binary entries and assume that it holds such explicit or implicit feedback data.

Algorithm 2: computing item coocurrences

```
function map(a_u•):
  foreach i ∈ a_u• do
    c ← sparse_vector()
    foreach j ∈ a_u• with j > i do
      c[j] ← 1
    emit(i, c)

function combine(i, c_1, ..., c_n):
  c ← vector_add(c_1, ..., c_n)
  emit(i, c)

function reduce(i, c_1, ..., c_n):
  s ← vector_add(c_1, ..., c_n)
  emit(i, s)
```

Expressing arbitrary similarity measures: We incorporate a wide range of measures for comparing the interactions of two items i and j by integrating three canonical functions into our algorithm. We first adjust each item rating vector via a function **preprocess()**:

$$\widehat{i} = preprocess(i) \quad \widehat{j} = preprocess(j)$$

Next, the second function **norm()** computes a single number from the preprocessed vector of an item:

$$n_i = norm(\widehat{i}) \quad n_j = norm(\widehat{j})$$

These preprocessing and norm computations are conducted in an additional single pass over the data, which starts with A^{T}, applies the two functions and transposes A^{T} to form A.

The next pass over the data is a modification of the approach presented in Section 4.1. Instead of summing up cooccurrence counts, we now compute the dot products of the preprocessed vectors.

$$dot_{ij} = \widehat{i} \cdot \widehat{j}$$

We provide those together with the numbers we computed via the *norm* function to a third function called **similarity()** which will compute a measure-specific similarity value (Algorithm 3).

$$S_{ij} = similarity(dot_{ij}, n_i, n_j)$$

With this approach we are able to incorporate a wide variety of different similarity measures which can be rephrased as a variant of computing a dot product. Note that this technique preserves the ability to apply a combiner in each pass over the data and is therefore highly efficient.

Table 1 describes how to express several common similarity measures through these canonical functions, including cosine, Pearson correlation and a couple of others evaluated by Google for recommending communities in its social network Orkut [24].

Example: The Jaccard coefficient between items i and j (two columns from the interaction matrix A) is computed as the ratio of the number of users interacting with both items to the number of users interacting with at least one of those items. It can easily be expressed by our algorithmic framework, as this example shows:

$$i = \begin{bmatrix}1\\-\\3\end{bmatrix} \quad j = \begin{bmatrix}2\\1\\5\end{bmatrix}$$

Pass 1: We start by having *preprocess* binarize the vectors:

$$\widehat{i} = bin(i) = \begin{bmatrix}1\\0\\1\end{bmatrix} \quad \widehat{j} = bin(j) = \begin{bmatrix}1\\1\\1\end{bmatrix}$$

The second function that is invoked for each of the vectors is *norm*. We let it return the L_1 norm, which gives us the number of non-zero components of each of the binary vectors:

$$n_i = \|\widehat{i}\|_1 = 2 \quad n_j = \|\widehat{j}\|_1 = 3$$

Pass 2: Finally the function *similarity* will be called given the dot product between the preprocessed vectors and their precomputed norms. We have to rearrange the formula of the Jaccard coefficient so that it can be computed from the numbers we have at hand:

$$jaccard(i,j) = \frac{|i \cap j|}{|i \cup j|} = \frac{dot_{ij}}{n_i + n_j - dot_{ij}} = \frac{2}{2+3-2} = \frac{2}{3}$$

Algorithm 3: computing arbitrary item similarities

```
function map(a_u•):
  foreach (i, k1) ∈ a_u• do
    d ← sparse_vector()
    foreach (j, k2) ∈ a_u• with j > i do
      d[j] ← k1 k2
    emit(i, d)

function combine(i, d1, ..., dn):
  d ← vector_add(d1, ..., dn)
  emit(i, d)

function initialize_reducer():
  n ← load_norms()

function reduce(i, d1, ..., dn):
  dots ← vector_add(d1, ..., dn)
  s ← sparse_vector()
  foreach j, d ∈ dots do
    s[j] = similarity(d, n[i], n[j])
  emit(i, s)
```

4.3 Sparsification

In order to be able to handle cases with an enormous number of items, we add means to decrease the density of the similarity matrix S to our final implementation.

To get rid of pairs with near-zero similarity, a similarity threshold can be specified, for which we evaluate a size constraint to prune lower scoring item pairs early in the process [4] and eventually remove all entries from S that are smaller than the threshold. Note however that this threshold is data dependent and must be determined experimentally to avoid negative effects on prediction quality.

Furthermore, it has been shown that the prediction quality of the item-based approach is sufficient if only the top fraction of the similar items is used [23], therefore we add another MapReduce step that only retains these top similar items per item in a single pass over the data.

4.4 Batch recommendation

Although the similarity matrix is usually used to compute recommendations online, some use cases such as generating personalized newsletters require batch recommendation for all users. To achieve that, we need another function called **recommend()** that is invoked with the similarity matrix S and item interaction history $a_{u\bullet}$ of an active user u and returns the top-N items to recommend to that user. A variety of strategies can be applied in the estimation procedure, ranging from simple weighted sum estimation [23], improved by baseline estimates [18], to more advanced techniques that incorporate domain specific knowledge and aim to diversify recommendations [7].

measure	*preprocess*	*norm*	*similarity*
Cosine	$\frac{v}{\|v\|_2}$	-	dot_{ij}
Pearson correlation	$\frac{(v-\bar{v})}{\|v-\bar{v}\|_2}$	-	dot_{ij}
Euclidean distance	-	$\hat{v}^2$	$\sqrt{n_i - 2 \cdot dot_{ij} + n_j}$
Common neighbors	$bin(v)$	-	dot_{ij}
Jaccard coefficient	$bin(v)$	$\|\hat{v}\|_1$	$\frac{dot_{ij}}{n_i + n_j - dot_{ij}}$
Manhattan distance	$bin(v)$	$\|\hat{v}\|_1$	$n_i + n_j - 2 \cdot dot_{ij}$
Pointwise Mutual Information	$bin(v)$	$\|\hat{v}\|_1$	$\frac{dot_{ij}}{\|U\|} \log \frac{dot_{ij}}{n_i n_j}$
Salton IDF	$bin(v)$	$\|\hat{v}\|_1$	$\frac{\|U\| \cdot dot_{ij}}{n_i n_j^2}(-\log \frac{n_i}{\|U\|})$
Log Odds	$bin(v)$	$\|\hat{v}\|_1$	$\log \frac{\frac{\|U\| \cdot dot_{ij}}{n_i n_j^2}}{1 - \frac{\|U\| \cdot dot_{ij}}{n_i n_j^2}}$
Log-likelihood ratio [10]	$bin(v)$	$\|\hat{v}\|_1$	$2 \cdot (H(dot_{ij}, n_j - dot_{ij}, n_i - dot_{ij}, \|U\| - n_i - n_j + dot_{ij}) - H(n_j, \|U\| - n_j) - H(n_i, \|U\| - n_i))$

Table 1: expressing measures with the canonical functions

If the similarity matrix fits into the memory of a single mapper instance, the most efficient way of embedding the recommendation computation is by executing a *broadcast join* [9] of the users' item interaction histories and the sparsified similarity matrix. As shown in Algorithm 4, the similarity matrix is broadcasted to all worker machines in the cluster via the distributed cache and the recommendations are computed in a map-only job over the users' interaction histories. Such a job is highly efficient as no reducer is required, which obviates the need for the *shuffle* phase and its associated sorting and network overhead.

In cases with an extreme number of items, the similarity matrix might not fit into the mappers' memory any more. In such a case a less performant *repartition join* [13] has to be used where the items each user has interacted with and their corresponding rows from the similarity matrix are sent over the network to a reducer that joins them after receival.

Algorithm 4: batch recommendation for all users

```
function initialize_mapper():
  S ← load_similarity_matrix()

function map(u, a_u•):
  r ← recommend(S, a_u•)
  emit(u, r)
```

4.5 Linear scalability with a growing user base

Recall that our goal is to develop an algorithmic framework that scales linearly with respect to a growing user base. As described in Section 2.2, the cost of the item-based approach is dominated by the densest rows of A, which correspond to the users with the most interactions. This cost, which we express as the number of item cooccurrences to consider, is the sum of the squares of the number of interactions of each user.

The number of interactions per user usually follows a heavy tailed distribution as illustrated in Figure 3 which plots the ratio of users with more than n interactions to the number of interactions n on a logarithmic scale. Therefore, there exists a small number of 'power users' with an unproportionally high amount of interactions. These drastically increase the runtime, as the cost produced by them is quadratic with the number of their interactions.

If we only look at the fact whether a user interacted with an item or not, then we would intuitively not learn very much from a 'power user': each additional item he interacts with will cooccur with the vast amount of items he already preferred. We would expect to gain more information from users with less interactions but a highly differentiated taste. Furthermore, as the relations between items tend to stabilize quickly [22], we presume that a moderately sized number of observations per item is sufficient to find its most similar items.

Following this rationale, we decided to apply what we call an *interaction-cut*: we selectively down sample the interaction histories of the 'power users'.

We apply this by randomly sampling p interactions from each such user's history, thereby limiting the maximum number of interactions per user in the dataset to p. Note that this sampling is only applied to the small group of 'power users', it does not affect the data contributed by the vast majority of non-'power users' in the long tail. Capping the effort per user in this way limits the overall cost of our approach to $|U|\ p^2$. We will experimentally show that a moderately sized p is sufficient to achieve prediction quality close to that of unsampled data. An optimal value for p is data dependent and must be determined by hold-out tests.

5. EVALUATION

In this section we present the results of a sensitivity analysis for the interaction-cut and conduct an experimental evaluation of our parallel algorithm on a large dataset[1].

We will show that the prediction quality achieved by using an interaction cut quickly converges to the prediction quality achieved with unsampled data. Subsequently, we will analyze the relationship between the size of the interaction-cut, the achieved quality and the runtime for the similarity computation in our large dataset. After that we will study the effects on the runtime speedup if we add more machines to the Hadoop cluster as well as the scaling behavior with a growing user base.

Prediction was conducted with weighted sum estimation enhanced by baseline estimates [18]. In a preprocessing step that has negligible computation cost, we estimate global user and item biases b_u and b_i that describe the tendency to deviate from the average rating μ. This gives us the simple

[1]code to repeat our experiments is available at http://github.com/dima-tuberlin/publications-ssnmm

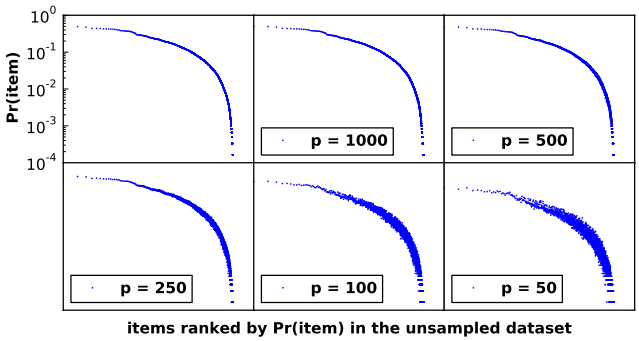

Figure 2: sensitivity of the probability of interaction with an item to an interaction-cut of size p in the Movielens dataset

baseline prediction $b_{ui} = \mu + b_u + b_i$ for the rating of a user u to an item i. To finally predict the rating r_{ui}, we use the normalized weighted sum over the user's ratings to the k most similar items of i, incorporating the baseline predictions:

$$r_{ui} = b_{ui} + \frac{\sum_{j \in S^k(i,u)} S_{ij}(A_{uj} - b_{uj})}{\sum_{j \in S^k(i,u)} S_{ij}}$$

5.1 Effects of the interaction-cut

We conducted a sensitivity analysis of the effects of the interaction-cut. We measured the effect on the probability of interaction with an item and on prediction quality for varying p on the **Movielens**[2] dataset consisting of 1,000,209 ratings that 6,040 users gave to 3,706 movies.

For our first experiment, we ranked the items by their probability of interaction as shown in the plot in the top left corner in Figure 2. Next, we applied the interaction-cut by sampling down the ratings of users whose number of interactions in the training set exceeded p and repeated this for several values of p. In the remaining plots of Figure 2, we retained the order of the items found in the unsampled data and plotted their probabilities after applying the interaction-cut for a particular p. We see that for $p \geq 500$ there is no observable distortion in the ranking of the items, which is a hint that this distribution is independent of the data omitted by sampling down the interactions of the 'power users'.

In our second experiment, we computed the prediction quality (by mean average error) achieved by a particular p by randomly splitting the rating data into 80% training and 20% test set based on the number of users. For this experiment, we additionaly used the **Flixster**[3] dataset consisting of 8,196,077 ratings from 147,612 users to 48,794 movies. Again, we applied the interaction-cut by sampling down the ratings of users whose number of interactions in the training set exceeded p and used the 80 most similar items per item for rating prediction. For these small datasets, the tests were conducted on a single machine using a modified version of Mahout's [3] *GenericItemBasedRecommender*.

Figure 4 shows the results of our experiments. Note that the right-most data points are equivalent to the prediction quality of the unsampled dataset. In the Movielens dataset we see that for $p > 400$ the prediction quality converges to the prediction quality of the unsampled data, in the Flixster

[2]http://www.grouplens.org/node/73

[3]http://www.cs.sfu.ca/~sja25/personal/datasets/

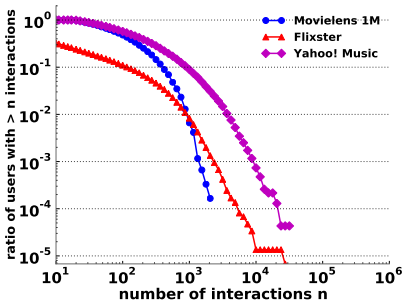

Figure 3: long tailed distribution of the number of interactions per user in various datasets

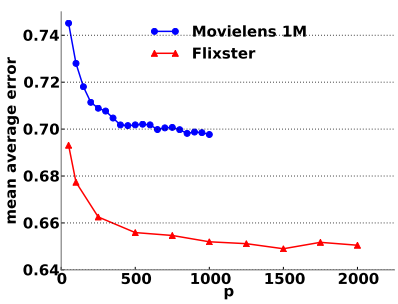

Figure 4: effects of an interaction-cut of size p on prediction quality in various small datasets

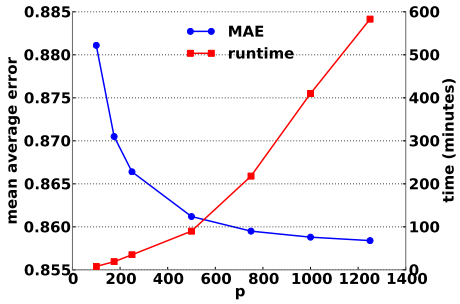

Figure 5: prediction quality and runtime for an interaction-cut of size p in the Yahoo! Music dataset

dataset this happens at $p > 750$. There is no significant decrease in the error for incorporating more interactions from the 'power users' after that. This confirms our expectation that we can compute recommendations based on the user data from the long tail and only samples from the 'power users' without sacrificing prediction quality.

5.2 Parallel computation with MapReduce

The following experiments were conducted on a Hadoop cluster with a MapReduce implementation of our approach. The cluster consisted of six machines running Apache Hadoop 0.20.203 [2] with each machine having two 8-core Opteron CPUs, 32 GB memory and four 1 TB disk drives. The experiments for showing the linear speedup with the number of machines were run on Amazon's computing infrastructure, where we rented *m1.xlarge* instances, 64-bit machines with 15 GB memory and eight virtual cores each.

5.2.1 Prediction quality

To test our approach in a demanding setting, we used a very large ratings dataset[1] which represents a snapshot of the Yahoo! Music community's preferences for various songs that were collected between 2002 and 2006. The data consists of 717,872,016 ratings that 1,823,179 users gave to 136,736 songs.

We used the 699 million training ratings provided in the dataset to compute item similarities and measured the prediction quality for the remaining 18 million held out ratings. We computed the 50 most similar items per item with Pearson correlation as similarity measure with a threshold of 0.01. Figure 5 shows the results we got for differently sized interaction cuts. We see that the prediction quality converges for $p > 600$, similar to what we have observed for the smaller datasets in Section 5.1. We additionally measured the root mean squared error and observed the same behavior, the prediction quality converged to an error of 1.16 here. We see the expected quadratic increase in the runtime for a growing p, which is weakened by the fact there is a quickly shrinking number of users with more than p interactions (from Figure 3 we know for example that only approximately 9% of the users have more than 1000 interactions).

The computational overhead introduced by the 'power users' is best illustrated when we compare the numbers for $p = 750$ and $p = 1000$: we see a decrease of only 0.0007 in the mean average error, yet the higher value of p accounts for a nearly doubled runtime. We conclude that we are able to achieve the convergence of the prediction quality in this large dataset with a p that is extremely low compared to the overall number of items and thereby results in a computation cost that is easily manageable even by our small Hadoop cluster.

In order to conduct a comparison to a single machine implementation, we tried the *item-based kNN* recommender of MyMediaLite [14], the *GenericItemBasedRecommender* provided by Apache Mahout [3] and the *ItemRecommender* from LensKit [11]. Unfortunately, not one of these was able to complete the computation due to problems with memory handling, although we ran them on a machine which had 48 GB of RAM available.

Based on our findings, we chose to set p to 600 for the following scalability experiments.

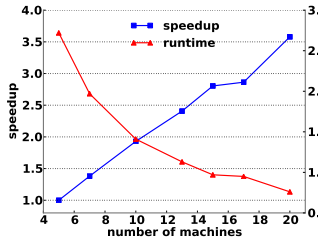

Figure 6: speedup for a growing number of machines in Amazon EC2

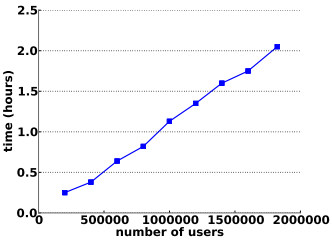

Figure 7: linear runtime increase for a growing user base

5.2.2 Linear speedup with the number of machines

The major promise of parallel processing platforms is seamless horizontal scale out by simply adding more machines to the cluster. This requires the computation speedup to be proportional to the number of machines. To experimentally evaluate this property of our algorithm, we made use of the *ElasticMapReduce* computing infrastructure provided by Amazon, which allows us to run our algorithm on a customly sized Hadoop cluster. We repeatedly ran the similarity computation with an increasing number of cluster machines. Figure 6 shows the linear speedup as expected by us. With 15 machines we were able to reduce the runtime of the similarity computation to less than one hour.

[1]R2 - Yahoo! Music User Ratings of Songs with Artist, Album, and Genre Meta Information, v. 1.0, http://webscope.sandbox.yahoo.com/

5.2.3 Linear scale with a growing number of users

Finally, we evaluate the scaling behaviour of our approach in the case of a rapid growth of the number of users. The Yahoo! Music dataset is already partitioned into several files, with each file containing approximately 77 million ratings given by 200,000 unique users. In order to simulate the growth of the user base, we used an increasing number of these files as input to our parallel algorithm and measured the duration of the similarity computation. Figure 7 shows the algorithm's runtime when scaling from 200,000 to 1,8 million users. We see a perfectly linear increase in the runtime which confirms the applicability of our approach in scenarios with enormously growing user bases.

As the speedup with the number of machines as well as the runtime for a growing number of users scale linearly, we can counter such growth by simply adding more machines to the cluster to keep the computation time constant.

6. CONCLUSIONS

We showed how to build a scalable, neighboorhood-based recommender system based on the MapReduce paradigm. We rephrased the underlying pairwise comparison to run on a parallel processing platform with partitioned data and described how a wide variety of measures for comparing item interactions easily integrate into our method. We introduced a down sampling technique called interaction-cut to handle the computational overhead introduced by 'power users'.

For a variety of datasets, we experimentally showed that the prediction quality quickly converges to that achieved with unsampled data for moderately sized interaction-cuts. We demonstrated a computation speedup that is linear in the number of machines on a huge dataset of 700 million interactions and showed the linear scale of the runtime with a growing number of users on that data.

In future work we intend to explore how our method could be used to scale out recommendation approaches that incorporate similarity computations on large networks [16, 25].

7. ACKNOWLEDGMENTS

A special thanks goes to Ted Dunning, Zeno Gantner and Moritz Kaufmann. The research leading to these results has received funding from the European Union (EU) in the course of the project 'ROBUST' (EU grant no. 257859) and used data provided by 'Yahoo! Academic Relations'.

8. REFERENCES

[1] K. Ali and W. van Stam. Tivo: Making show recommendations using a distributed collaborative filtering architecture. KDD, 2004.

[2] Apache Hadoop, http://hadoop.apache.org.

[3] Apache Mahout, http://mahout.apache.org.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. WWW, pp. 131–140, 2007.

[5] R. M. Bell and Y. Koren. Lessons from the netflix prize challenge. *SIGKDD Newsl.*, 9:75–79, 2007.

[6] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. WWW, pp. 271–280, 2007.

[7] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath. The youtube video recommendation system. RecSys, pp. 293–296, 2010.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, 2008.

[9] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA - a high performance dataflow database machine. VLDB, pp. 228–237, 1986.

[10] T. Dunning. Accurate methods for the statistics of surprise and coincidence. ACL, 19:61–74, 1993.

[11] M. D. Ekstrand, M. Ludwig, J. A. Konstan, and J. T. Riedl. Rethinking the recommender research ecosystem: reproducibility, openness, and lenskit. RecSys, pp. 133–140, 2011.

[12] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning Fast Iterative Data Flows. *PVLDB*, 2012.

[13] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the system software of a parallel relational database machine GRACE. VLDB, pp. 209–219, 1986.

[14] Z. Gantner, S. Rendle, C. Freudenthaler, and L. Schmidt-Thieme. Mymedialite: a free recommender system library. RecSys, pp. 305–308, 2011.

[15] R. Gemulla, E. Nijkamp, P. Haas, and Y. Sismannis. Large-scale matrix factorization with distributed stochastic gradient descent. KDD, pp. 69–77, 2011.

[16] M. Jamali and M. Ester. Trustwalker: a random walk model for combining trust-based and item-based recommendation. KDD, pp. 397–406, 2009.

[17] J. Jiang, J. Lu, G. Zhang, and G. Long. Scaling-up item-based collaborative filtering recommendation algorithm based on hadoop. *SERVICES*, pp. 490 –497, 2011.

[18] Y. Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. *ACM Trans. KDD*, 4:1:1–1:24, 2010.

[19] G. Linden, B. Smith, and J. York. Amazon.com recommendations: item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003.

[20] Y. Low and J. Gonzalez and A. Kyrola and D. Bickson and C. Guestrin and J. Hellerstein. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB*, 2012.

[21] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. CSCW, pp. 175–186, 1994.

[22] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor. *Recommender Systems Handbook*. 2011.

[23] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. WWW, pp. 285–295, 2001.

[24] E. Spertus, M. Sahami, and O. Buyukkokten. Evaluating similarity measures: a large-scale study in the orkut social network. KDD, pp. 678–684, 2005.

[25] P. Symeonidis, E. Tiakas, and Y. Manolopoulos. Product recommendation and rating prediction based on multi-modal social networks. RecSys, pp. 61–68, 2011.

[26] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. AAIM, pp. 337–348, 2008.