Distributed and Scalable PCA in the Cloud

Arun Kumar
Department of Computer Sciences
University of Wisconsin-Madison

Nikos Karampatziakis, Paul Mineiro,
Markus Weimer, Vijay K Narayanan
Microsoft Cloud and Information Services Lab

Abstract

Principal Component Analysis (PCA) is a popular technique with many applications. Recent randomized PCA algorithms scale to large datasets but face a bottleneck when the number of features is also large. We propose to mitigate this issue using a composition of structured and unstructured randomness within a randomized PCA algorithm. Initial experiments using a large graph dataset from Twitter show promising results. We demonstrate the scalability of our algorithm by implementing it both on Hadoop, and a more flexible platform named REEF.

1 Introduction

PCA, and the related Singular Value Decomposition (SVD), is a popular tool to compress high-dimensional data to fewer dimensions and uncover hidden patterns [16, 21]. Given a dataset $\mathbf{X} \in \mathbb{R}^{n \times p}$ with SVD $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, the product $\mathbf{U}\mathbf{\Sigma}$ are the *principal components* and $\mathbf{V}$ are called the *loadings*. Recent randomized algorithms for PCA and SVD can scale to datasets with large n in a distributed setting [9, 15, 17]. If the data are distributed close to a k-dimensional subspace ($k \ll p$), these algorithms need only two passes over $\mathbf{X}$ to compute the top k singular values. However, they require $O(pk)$ memory, which could be a bottleneck for datasets that are both "tall and wide", e.g., from text and graph domains. Karampatziakis and Mineiro [10] recently proposed the HashPCA algorithm to mitigate this bottleneck by interleaving a structured projection to lower dimension $d \ll p$ with unstructured randomness within a randomized PCA algorithm [9]. HashPCA needs only $O(dk)$ memory, and the parameter d can be chosen based on the underlying hardware.

We provide an initial demonstration of the utility and scalability of HashPCA. We discuss distributed implementations of HashPCA on the MapReduce/Hadoop framework and a new distributed computation platform named Retainable Evaluator Execution Framework (REEF) [5]. REEF provides an abstraction on top of the cluster resource manager YARN [23] to make it easier to write distributed applications. Implementing machine learning algorithms on REEF offers key systems-level advantages over Hadoop to improve performance, scalability, and robustness – viz. caching, sophisticated aggregation, locality-awareness, and fault-awareness. We demonstrate some of these advantages using HashPCA on a real tall and wide dataset – a user-follower relationship graph from Twitter.

Background and Related Work Our work builds upon Halko et al.'s recent algorithm for truncated SVD [9]. Other scalable algorithms have also been proposed for SVD on datasets that are either tall or wide [8, 15, 17, 18]. Hashing has been effectively used to approximate high dimensional feature maps in many applications [19, 20, 24]. It is typically implemented by representing all features as strings from an alphabet Σ and then using a hash function $h : \Sigma^* \to \{1, \ldots, d\}$ to map each original feature to a hashed feature. HashPCA utilizes feature hashing as a sparsity-preserving structured randomness to enable PCA on sparse datasets that are both tall and wide.

There is increasing interest in platforms for writing distributed applications to process "Big Data". Hadoop [25] is an open-source system that implements the MapReduce programming model [6]. Spark [26], Asterix [3], and Stratosphere [12] provide data flow-based programming models coupled with parallel runtimes, while GraphLab [13], and Pregel [14] provide graph-parallel programming models. In contrast, REEF decouples the programming model from the runtime.

Algorithm 1 HashPCA: Truncated PCA with hashing

Inputs: $\mathbf{X}_{n\times p}, \mathbf{H}_{p\times d}, k, d$ ▷ $\mathbf{H}$ not materialized

1: $\mathbf{\Omega}_{d\times k} \leftarrow$ Random Gaussian matrix ($\Omega_{ij} \sim \mathcal{N}(0,1)$)

2: $\mathbf{Y}_{d\times k} \leftarrow (\mathbf{XH})^\top(\mathbf{XH})\ \mathbf{\Omega}\ /\ n$ ▷ First data-parallel pass

3: $\mathbf{Q}_{d\times k} \leftarrow$ Gram-Schmidt Orthonormalization of the column space of $\mathbf{Y}$. ▷ $O(dk)$ space

4: $\mathbf{Z}_{d\times k} \leftarrow (\mathbf{XH})^\top(\mathbf{XH})\ \mathbf{Q}\ /\ n$ ▷ Second data-parallel pass

5: $\mathbf{\Upsilon}\tilde{\mathbf{\Sigma}}_\mathbf{1}^4\mathbf{\Upsilon}^\top \leftarrow$ Spectral decomposition of $\mathbf{Z}^\top\mathbf{Z}$ ▷ $\mathbf{Z}^\top\mathbf{Z} \in \mathbb{R}^{k\times k}$

6: **return** ($\tilde{\mathbf{V}}_\mathbf{1} = \mathbf{Z\Upsilon}(\tilde{\mathbf{\Sigma}}_\mathbf{1}^2)^\dagger, \tilde{\mathbf{\Sigma}}_\mathbf{1}$) ▷ $\Sigma^\dagger$ is the Moore-Penrose pseudo-inverse of Σ

2 Scalable PCA on Tall and Wide Datasets

Let $\mathbf{X} \in \mathbb{R}^{n\times p}$ be the data matrix. We assume that the data are centered.[1] The SVD $\mathbf{X} = \mathbf{U\Sigma V}^\top$ yields orthogonal matrices $\mathbf{U} \in \mathbb{R}^{n\times n}$ and $\mathbf{V} \in \mathbb{R}^{p\times p}$, while $\mathbf{\Sigma} \in \mathbb{R}^{n\times p}$ is a diagonal matrix with entries $\mathbf{\Sigma}_{ii} = \sigma_i$ in non-ascending order. Truncating by utilizing the top k singular values yields the bast rank-k approximation of $\mathbf{X}$ ($k <= min(n,p)$) in the Frobenius norm [7], leading to $\tilde{\mathbf{X}} = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^\top$ where $\mathbf{U}_k \in \mathbb{R}^{n\times k}$, $\mathbf{\Sigma}_k \in \mathbb{R}^{k\times k}$, and $\mathbf{V}_k \in \mathbb{R}^{p\times k}$. Any SVD algorithm can compute $\mathbf{V}_k$ and $\mathbf{\Sigma}_k$. However, for large n or p, only randomized SVD algorithms are practical. Randomized SVD algorithms work in two phases. In the first phase they employ a random projection matrix $\mathbf{\Omega}$ as a way to probe the range of $\mathbf{X}$ (or the covariance $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$). This requires at least one pass over the data, which can be streamed along the examples since $\frac{1}{n}\mathbf{X}^\top\mathbf{X\Omega} = \frac{1}{n}\sum_{i=1}^n x_i x_i^\top\mathbf{\Omega}$, where $x_i^\top$ is the i-th example (row) of $\mathbf{X}$. Next, they orthogonalize the image of $\mathbf{\Omega}$ under the data and project onto that basis in the second pass. Although these algorithms have been adapted for PCA before [8, 18], to the best of our knowledge, they assume that the orthogonalization step can be done efficiently. This is possible if either n or p is not too large, but not both. When $p < n$, this step uses $O(pk)$ memory and has time complexity $O(pk^2)$. It is efficient if p is modest, say, 10^6 on current commodity hardware. However, for large datasets that also have large p, e.g., the adjacency matrix of an online social network, this step might become impractical.[2]

HashPCA (Algorithm 1), employs structured randomness to reduce the number of features from p to d (where $d \ll p$) so that randomized SVD algorithms become viable. Thus, both the space complexity and communication cost (in a distributed setup) are reduced from $O(pk)$ to $O(dk)$. Although not materialized, we represent the structured randomness as a matrix $\mathbf{H} \in \mathbb{R}^{p\times d}$. For sparse data, e.g. text or social graph data, hash based structured randomness [24] is computationally convenient and empirically effective. Conceptually, this scheme multiplies the data by a hashing matrix $\mathbf{H} \in \mathbb{R}^{p\times d}$ which is determined by two hash functions $h : \{1,\ldots,p\} \to \{1,\ldots,d\}$ and $\xi : \{1,\ldots,p\} \to \{\pm 1\}$, with $H_{ij}|_{\xi,h} = \xi(i)1_{h(i)=j}$. Other choices of structured random projections such as subsampled fast real transforms (e.g. Walsh-Hadamard, Hartley) can be used, and should be used if the data vectors are dense.

3 Scalable Distributed Implementations

In Algorithm 1, steps 2 and 4 are data-intensive and can be trivially parallelized along the rows of $\mathbf{X}$. We now discuss the implementation of HashPCA on two distributed data processing frameworks: MapReduce/Hadoop and the Retainable Evaluator Execution Framework (REEF).

MapReduce/Hadoop We perform one MapReduce job per pass, coordinated by a centralized Driver program. The dataset $\mathbf{X}$ resides on HDFS, and is partitioned row-wise (one example per line). In the first pass, each Mapper initializes a partial sum for $\mathbf{Y}$ in the `setup()` function, and uses a given random seed to initialize $\mathbf{\Omega}$. The `map()` function parses an example x_i, hashes it and accumulates the partial covariance ($\mathbf{H}^\top x_i(\mathbf{H}^\top x_i)^\top\mathbf{\Omega}$) into $\mathbf{Y}$. Each Mapper emits its partial sum $\mathbf{Y}$ as part of a single key-value pair in its *cleanup()* function. The Reducer aggregates all partial sums to obtain the projected covariance matrix. The Driver computes the orthonormalization and writes $\mathbf{Q}$ to HDFS. The second job is similar to the first, except that each Mapper also reads $\mathbf{Q}$ along with the data blocks, and the driver computes the spectral decomposition.

[1] Uncentered data requires a simple rank-one modification to Algorithm 1 for additional $O(d)$ space.

[2] However, we are also looking at adopting some interesting recent ideas on distributed QR factorization [4].

Figure 1: On the left: (A) High-level architecture of a distributed application on REEF. (B) Data-parallel computation for HashPCA on REEF. On the right: A portion of the Twitter embedding showing a geographically coherent "British" cluster (top left) and a topically coherent "News" cluster (middle and bottom right).

Retainable Evaluator Execution Framework (REEF) REEF is a framework that aims to make it easy to implement scalable, fault-tolerant runtime environments for a range of computational models. It provides an abstraction atop the cluster resource manager YARN to enable hardware and software resources to be *retained* across applications. Thus, REEF decouples the lifetimes of resources from the lifetimes of applications. REEF's abstraction consists of the following – a *Driver* process that orchestrates control flow, *Activities*, which perform the data processing and computations (generalizing Mappers and Reducers), *Evaluators* that act as containers on which Activities are run, and *Services*, which are objects (e.g., cache, connections, etc.) that are retained by an Evaluator across Activities that run on it. The control flow is centralized in the Driver, which consists of user-supplied event handlers for events such as system startup, allocation of Evaluators, start and completion of Activities as well as various failure scenarios.

Our implementation of HashPCA on REEF is similar to the one on Hadoop (Figure 1(B)). The dataset is pre-partitioned into M blocks on HDFS. The job Driver controls both passes and performs the orthonormalization and spectral decomposition. Based on the number of available machines (a user-given parameter), the Driver requests N Evaluators. The Driver then executes one Activity per Evaluator. Each Activity reads and processes $\frac{M}{N}$ data blocks. The Activities essentially perform the same computations as the Mappers in our Hadoop implementation. Upon completion, the Activities notify the Driver, which then starts the aggregation, akin to the Reducer. In our current implementation, each Activity sends its partial sum directly to the Driver through a messaging API that relies on the Netty framework. If the message is too large for Netty to handle (specifically, larger than 1MB [1]), the Activities write the partial sum matrices to HDFS. The Driver then reads the matrices from HDFS to perform the aggregation. The second pass is performed in a similar manner.

Our experience with implementing HashPCA on both REEF and Hadoop highlighted four systems-level advantages of REEF. *(1) Caching*: Hadoop has high I/O overhead since it reads data from disk in both passes. In REEF, we implemented a simple cache on the Evaluator to store as many parsed and hashed examples as possible in memory in the first pass so that the second pass is faster. *(2) Aggregation*: REEF enables us to configure aggregation trees of different fan-outs and also control how much data is read and when during aggregation. *(3) Locality-aware Scheduling*: Unlike Hadoop's fixed greedy scheduling policy to allocate Mappers to blocks, REEF enables us to exploit more factors such as data locality, network topology, node capacities, etc. *(4) Fault-Awareness*: Unlike Hadoop's a fixed fault tolerance policy of restart recovery, REEF enables us to adopt alternative fault handling mechanisms that could potentially improve performance.

Currently, we have only implemented the Caching functionality and plan to incorporate the others in the future as REEF matures. Still, as our experiments will show, our implementation on REEF provided better performance and scalability than Hadoop, We believe this is a promising first step for building more scalable machine learning algorithms on REEF.

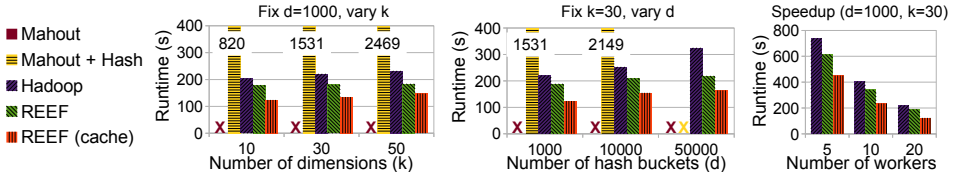

Figure 2: Plots of runtimes as we vary k and d, fixing one at a time. Mahout crashed for all these settings, while Mahout + Hash worked (hashing time excluded). Also shown are the speedups.

4 Experiments

We now present some initial experimental results that validate the utility of HashPCA as well as test the performance and scalability of our distributed implementations.

Dataset and Setup We use a large public graph data set derived from a 2009 crawl of Twitter [11]. We constructed the adjacency matrix, where entry (i, j) indicates if a user i follows user j. We removed users who followed < 5 or > 300 other users, yielding 18 million examples. The dimensionality (p) is 29 million. The dataset is in plain text Vowpal Wabbit format [2] and is 17GB in size. The experiments were run on a 23-node cluster on the Azure cloud service with Ubuntu 12.04.2 VMs. Each instance is of class Medium (A2) with 3.5GB RAM, 10GB disk, and 2 virtual AMD Opteron 4171 HE cores. We installed Apache Hadoop 2.0.4 and used default replication of 3.

Quality of HashPCA Results We run HashPCA on the Twitter graph data with parameters $k = 30$ and $d = 1.3$ million. The loadings form an "interest fingerprint" operator, which projects any user into a latent space based upon the set of users who are followed by that user. We can explore the impact of any single user on this latent representation by projecting a singleton follower set containing only that user. We did this for the 2000 most popular Twitter accounts in the dataset, and then used t-SNE [22] to project down to two dimensions for visualization. A portion of the visualization is shown in Figure 1. The figure shows both geographically coherent and topically coherent clusters, which indicate that PCA has captured interesting structure from the graph.

Performance and Scalability We now compare the performance and scalability of our distributed implementations. We also present results for an open-source toolkit for SVD and PCA from Apache Mahout. We expect Mahout to crash since this dataset's dimensionality is too large for its implementation. Hence, we hash the dataset as a pre-processing step for Mahout. We pre-partitioned the dataset into 20 blocks on HDFS. Thus, Hadoop spawns 20 Mappers and REEF uses 20 Evaluators. We vary the two main parameters – number of PCA dimensions (k), and number of hash buckets (d), and also vary the number of compute nodes. Figure 2 shows the results.

As expected, Mahout crashed since p is too large (we verified that in its implementation, a Reducer accumulates blocks of size $O(pk)$ in memory, which is about 2.3GB per block without Java object overheads). HashPCA mitigates this bottleneck by bringing the memory needed from $O(pk)$ down to $O(dk)$. Mahout works on the hashed dataset, but is still slower than our Hadoop implementation (between 400% to 1000%). Our REEF implementation is marginally faster than Hadoop, due to lower framework overheads (start-up, function calls, etc.). However, REEF with caching is 40% to 100% faster than Hadoop. The runtimes for our HashPCA implementations increases slowly with k and d, since the I/O cost of scanning is dominant. We also see near-linear speedups for both Hadoop and REEF – about 3.3x as the number of nodes is quadrupled. REEF with caching has a higher speedup of 3.6x since the whole dataset fits in the cache across 20 nodes in the second pass.

5 Conclusion and Ongoing Work

We presented a new, scalable algorithm for PCA that combines structured and unstructured randomness within a randomized PCA algorithm to scale to tall and wide datasets. We implemented the algorithm on MapReduce/Hadoop and REEF and discussed how REEF's primitives provided greater efficiency and scalability. We provided an initial validation of the benefits of HashPCA using a real dataset from Twitter. Our ongoing work is on incorporating new REEF APIs for aggregation trees and scheduling (apart from caching) as well as adding fault-awareness into distributed PCA.

References

[1] The netty project. http://netty.io.

[2] Vowpal wabbit. http://hunch.net/~vw.

[3] S. Alsubaiee et al. Asterix: an open source system for"big data" management and analysis. *VLDB (Demo)*, 2012.

[4] D. F. G. Austin R. Benson and J. Demmel. Direct qr factorizations for tall-and-skinny matrices in mapreduce architectures. IEEE BigData, 2013.

[5] B.-G. Chun et al. Reef: Retainable evaluator execution framework. *VLDB (Demo)*, 2013.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. OSDI, 2004.

[7] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[8] N. Halko, P.-G. Martinsson, Y. Shkolnisky, and M. Tygert. An algorithm for the principal component analysis of large data sets. *SIAM Journal on Scientific Computing*, 33(5):2580–2594, 2011.

[9] N. Halko, P.-G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[10] N. Karampatziakis and P. Mineiro. Combining Structured and Unstructured Randomness in Large Scale PCA. *ArXiv e-prints*, Oct. 2013.

[11] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[12] M. Leich et al. Applying stratosphere for big data analytics. BTW, 2013.

[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *VLDB*, 2012.

[14] G. Malewicz et al. Pregel: a system for large-scale graph processing. ACM SIGMOD, 2010.

[15] D. Okanohara. redsvd (software). *https://code.google.com/p/redsvd*, 2010.

[16] K. Pearson. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2.11:559–572, 1901.

[17] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[18] V. Rokhlin, A. Szlam, and M. Tygert. A randomized algorithm for principal component analysis. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1100–1124, 2009.

[19] Q. Shi, J. Petterson, G. Dror, J. Langford, A. Smola, and S. Vishwanathan. Hash kernels for structured data. *The Journal of Machine Learning Research*, 10:2615–2637, 2009.

[20] S. Sonnenburg and V. Franc. Coffin: A computational framework for linear svms. In *Proceedings of the 27th International Conference on Machine Learning*, Haifa, Israel, 2010.

[21] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

[22] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.

[23] V. Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. *SOCC*, 2013.

[24] K. Q. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. J. Smola. Feature hashing for large scale multitask learning. *CoRR*, abs/0902.2206, 2009.

[25] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. HotCloud'10, 2010.