Principal Component Analysis Implementation on Husky

Liu Jie
Department of Computer Science
and Engineering
The Chinese University of Hong Kong
1155047039@link.cuhk.edu.hk

Bao Ergute
Department of Computer Science
and Engineering
The Chinese University of Hong Kong
1155076717@link.cuhk.edu.hk

***Abstract*—Principal component analysis (PCA) is a classical data analysis technique that finds linear transformations of data that remain the maximal amount of variance. PCA is an important tool in many areas including dimensionality reduction, image processing, data visualization and information retrieval. We implemented several PCA algorithms on the Husky platform, which is developed mainly for in-memory large scale data mining, and adopted a few optimizations to achieve scalability in distributed settings. We present three methods of computing PCA in this report: Basic-PCA (computing eigen decomposition of covariance matrix), SVD-PCA (using singular value decomposition), scalable-PCA, or sPCA (based on Probabilistic PCA).**

Keywords—Husky, principal component analysis.

I. Introduction

Many current machine learning algorithms were designed for centralized computing system, making it difficult to make sense of big data. In other words, increasing scale of data highlights the need for designing distributed machine learning algorithms. Husky [2] offers a simple yet expressive set of object interaction primitives, and can serve as a platform for developing distributed algorithms with comparable efficiency as low-level codes. We developed PCA libraries on Husky to avoid common challenges appearing in distributed settings, such as handling failures, balancing load, etc.

PCA is not only a useful tool in areas like image processing, data visualization, but also a key step in many other machine learning algorithms which do not perform well with high dimensional data, since it reduces the dimensionality of data. We implemented three PCA algorithms on Husky: Basic-PCA (eigen decomposition), SVD-PCA (singular value decomposition), sPCA (Probabilistic PCA). We will lay stress on sPCA since Probabilistic PCA promises the best theoretical scalability. Our experiments showed that PCA algorithms in popular libraries, such as Mahout on MapReduce and MLlib on Spark do not scale well to support high-dimensionality data analysis, and sPCA on Husky can outperform these two libraries.

The rest of this paper is organised as follows: In Section 2, we present different PCA methods. In Section 3, we presents our experimental evaluation. Section 4 concludes the paper.

II. Analysis of PCA Algorithms

In this section, we analyze different methods of computing the principle components of a given dataset represented as a matrix, and also present their implementations on Husky.

We represent the given dataset as matrix Y of size $N \times D$. That means that the dataset has N samples and every sample has D features. A PCA algorithm obtains d principal components ($d \leq D$) that explain the most variance (and hence information) of the data in Y. Original matrix Y can be mapped on the principal components using the following formula: $X = Y * C$, where C is a transformation matrix of size $D \times d$.

A. Basic-PCA

One of the simplest way to compute principal components is to compute the eigen decomposition of the covariance matrix.

```
Algorithm 1 Basic-PCA (Matrix Y, int N, int D, int d)
1: $Ym = columnMean(Y)$
2: $Yc = Y.rowwise - Ym$
3: for all Yi in Yc.rows do
4:     $cov\_mat += Yi^T * Yi$
5: end for
6: $C = EigenSolver(cov\_mat).sort().submatrix()$
```

This method is implemented in MLlib on Spark, however, it is not suitable for large datasets due to high computational cost and high communication cost. The computational cost is dominated by the computation of covariance matrix, which is O(N*D*min(N, D)). In addition to that, this method generates a dense covariance matrix of size $D \times D$ which can account for high communication cost. It still works for small dimensional data. For example, given a dataset of size 3000×120, we prefer basic-PCA than scalable-PCA which we will mention later, since computing transformation matrix C directly is more efficient than using EM method in this case.

As shown in algorithm 1, the basic-PCA algorithm is only partially distributed. Computation of covariance matrix can be separated into a number of parallel tasks, but the eigen decomposition cannot. We compute Eigenvectors and Eigenvalues using EigenSolver in Eigen/Eigenvalues module. The implementation of this algorithm can now be found in Husky/examples folder.

B. Singular Value Decomposition

Let $A \in R^{m \times n} (m \geq n)$ have the singular value decomposition $A = U \sum V^T$. Then the symmetric matrix $M = A^T A$ has eigen values ${\sigma^2}_1 \geq {\sigma^2}_2 \geq \cdots \geq {\sigma^2}_n$ corresponding to the eigen vectors v_i, $i = 1, 2, \cdots, n$. Now let's get into our implementation of SVD on Husky. First, we read the input matrix $A \in R^{m\times n}$ line by line. If $m \geq n$, then we use the method in Basic-PCA to compute $M = A^T A$. Then we collect the matrix M in one machine and call *Eigen* to compute the eigen values $\sigma^2{}_1 \geq \sigma^2{}_2 \geq \cdots \geq \sigma^2{}_n$ and corresponding eigen vectors v_i, $i = 1, 2, \cdots, n$ of M. Then matrix V consists of the right singular vectors we want. Then we can obtain u_i, $i = 1, 2, \cdots, n$ by the formula $u_i = (1/\sigma_i)Av_i$. If $m < n$, we need to compute $M = AA^T$ first and collect it in a single machine. Then we call *Eigen* to compute the eigen values $\sigma^2{}_1 \geq \sigma^2{}_2 \geq \cdots \geq \sigma^2{}_m$ and corresponding eigen vectors u_i, $i = 1, 2, \cdots, m$ of M. At last, we obtain $v_i = (1/\sigma_i)A^T u_i$. As we can see, when both m and n are large, neither $M \in R^{n\times n}$ nor $M \in R^{m\times m}$ can fit in the memory of a single machine. So our implementation of SVD are only suitable for tall and skinny matrices or short and fat matrices.

```
input  : $A \in R^{m\times n}$
output: $u_i, i = 1, 2, \cdots, r$
        $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r$
        $v_i, i = 1, 2, \cdots, r$
if $m \geq n$ then
    $M = A^T * A$ in parallel;
    $v_i$ = ComputeEigenVectors($M$), $i = 1, 2, \cdots, r$;
    $\sigma^2{}_i$ = ComputeEigenValues($C$), $i = 1, 2, \cdots, r$;
    $u_i = (1/\sigma_i)Av_i$ in parallel;
else
    $M = A * A^T$ in parallel;
    $u_i$ = ComputeEigenVectors($M$), $i = 1, 2, \cdots, r$;
    $\sigma^2{}_i$ = ComputeEigenValues($M$), $i = 1, 2, \cdots, r$;
    $v_i = (1/\sigma_i)A^T u_i$ in parallel;
end
```

C. SVD-PCA

Singular Value Decomposition has some connections with PCA. As we can see v_i s are exactly the principal components we wanted in PCA. When $m < n$, the computation for principal components of the Basic-PCA encounters a bottleneck, i.e. $M = A^T * A$ may not fit in the memory when n is big. However, when A is short and fat we can obtain its right singular vectors using SVD. Obviously, the right singular vectors are the principal components we want in PCA. In conclusion, when the input matrix is short and fat, we can use SVD to compute the principal components.

D. scalable-PCA

Websites, social networks, and sensors generate massive amount of data every day, and some datasets may have millions of dimensions. Our experiments showed that Spark fails to process datasets of high dimensionality. That is the motivation for implementing scalable-PCA (sPCA) on Husky.

Scalable-PCA is based on Probabilistic PCA, so we present Probabilistic PCA in some details first. Probabilistic PCA (PPCA) is a probabilistic approach, in which PCA is presented as a latent (unobserved) variable model that seeks a linear relation between a D-dimensional observed data vector **y** and a d-dimensional latent variable **x**. The model is defined by:

$$\mathbf{y} = \mathbf{C} * \mathbf{x} + \mu + \varepsilon \tag{1}$$

The motivation is that, with $d < D$, the latent variables will offer a more parsimonious explanation of the dependencies between the observations. $\mathbf{x} \sim N(\mathbf{0}, \mathbf{I})$, and $\varepsilon \sim N(\mathbf{0}, \varphi)$. Equation (1) induces a corresponding Gaussian distribution for the observations $\mathbf{y} \sim N(\mu, CC^T + \varphi)$. The model parameters can be determined by maximum-likelihood, although because there is no closed-form analytic solutions for $\mathbf{C}$ and φ, their values can be obtained via EM method.

The **x**-conditional probability distribution over **y**-space is given by

$$\mathbf{y}|\mathbf{x} \sim N(\mathbf{C}\mathbf{x} + \mu, ss * \mathbf{I})$$

The marginal distribution for the observed data **y** is obtained by integrating out the latent variables:

$$\mathbf{y} \sim N(\mu, \mathbf{M}),$$

where the observation covariance model is specified by $\mathbf{M} = \mathbf{C}\mathbf{C}^T + ss * \mathbf{I}$. The corresponding loglikelihood is then

$$L = -\frac{N}{2}\{d\ln(2\pi) + \ln \mid \mathbf{M} \mid + tr(\mathbf{M}^{-1}\mathbf{S})\},$$

where S is the sample covariance matrix of the observations $\{\mathbf{y}_n\}$. Estimates for $\mathbf{C}$ and ss can be obtained by iterative maximization of L using EM algorithm, which is a well-known method to optimize the likelihood of the models when a closed form solution does not exist. The pseudo code of sPCA is shown below:

```
Algorithm 2 sPCA (Matrix Y, int N, int D, int d)
1: $C = normrnd(D, d)$
2: $ss = normrnd(1, 1)$
3: Ym = meanJob(Y)
4: ss1 = FnormJob(Y)
5: while not STOP_CONDITION do
6:     $M$ = $C^T$ * C + ss * I
7:     CM = C * $M^{-1}$
8:     Xm = Ym * CM
9:     $\{X_tX, Y_tX\}$ = $Y_tXJob$(Y, Ym, Xm, CM)
10:    $X_tX$ += $ss * M^{-1}$
11:    $C = Y_tX/X_tX$
12:    $ss2 = trace(X_tX * C^T * C)$
13:    $ss3$ = $ss3Job$(Y, Ym, CM, C)
14:    $ss = (ss1 + ss2 - 2 * ss3)/N/D$
15: end while
```

The function *normrnd*(r,c) gives a random matrix of size $r \times c$ with Normal Distribution. The algorithm initializes the transformation matrix C and the variance ss with random values. At each iteration, it improves C and ss until it reaches the STOP_CONDITION. The time complexity is O(NDd), which proves itself a potential candidate for performing PCA for large datasets.

We adopted several optimizations proposed by Elgamal in his sPCA paper [1]. First, Sparse matrices can achieve a small disk and memory footprint by storing only non-zero elements, but subtracting mean from original data would make many elements non-zero. Therefor we do not subtract the mean and keep original matrix Y and mean vector Ym in two separate data structures. We propagate the mean throughout the different matrix operations.

$$Yc * C = (Y - Ym) * C = Y * C - Ym * C$$

Second, we note that storing and exchanging X is expensive due to its large size, so we redundantly compute X at each job that consumes it as input. This approach trades communication cost with computation cost.

III. Performance evaluation

We present the comparison of sPCA on Husky and MLlib-PCA on Spark. The experiment result is shown below. We can see that MLlib-PCA is not capable of processing datasets of 10000 or more dimensions, while sPCA can do it without problems. Note that if the number of features is below one thousand, basic-PCA and MLlib-PCA is more suitable for that task.

Input($N \times D$)	d	SPCA	MLlib
$353 \times 2K$	300	80sec	225sec
$353 \times 2K$	100	22sec	170sec
$353 \times 2K$	10	6sec	120sec
64×47236	50	315sec	fail
20242×47236	10	100sec/iteration	fail

Both SPCA and Basic-PCA was implemented and run on Husky. Mlib-PCA was run on Spark. All tests were run under the same environment (20 workers, 20 threads).

IV. Conclusion

In this report, we analyzed different methods for computing principal components of an input matrix in a distributed setting. Current PCA algorithm on Spark has significant computation and communication bottlenecks, and sPCA we implemented on Husky can do much better comparing to MLlib-PCA when faced with datasets with high dimensions ($D > 1k$).

Acknowledgment

The authors would like to thank Professor James Cheng for his encouragement and guidance, and thank all the senior teammates in Husky Team for their patience and assistance. We wish to spend another summer with this great team.

References

[1] T. Elgamal, M. Yabandeh, A. Aboulnaga, W. Mustafa, and M. Hefeeda. spca: Scalable principal component analysis for big data on distributed platforms. In *SIGMOD*, pages 79–91, 2015.

[2] F. Yang, J. Li, and J. Cheng. Husky: Towards a more efficient and expressive distributed computing framework. *PVLDB*, 9(5):420–431, 2016.