Dimensionality Reduction Algorithms On Husky

BAO Ergute, LIU Jie
The Chinese University of Hong Kong

***Abstract*—Dimensionality reduction is a powerful tool for retrieving information and gain valuable insights from massive amounts of data. Husky [2] is a distributed computing system designed to handle mixed jobs of coarse-grained transformations, graph computing and machine learning. We implemented three dimensionality reduction algorithms on Husky. The principal Component Analysis(PCA) and Singular Value Decomposition(SVD) are widely used in big data processing. PCA is concerned with explaining the variance structure of a set of variables through a few linear combinations. In other words, PCA can be used to reduce the old set of variables to a new set of fewer variables. While SVD is to find the factorization of matrix. The factorization can be used for matrix approximation, data mining, etc. We implemented both Scalable Probabilistic PCA(SPCA) and Basic-PCA on a distributed platform called Husky. We also implemented SVD on Husky. Then we compared and analyze the performance of SVD, Basic-PCA and SPCA. Our experiment shows that SPCA outperforms SVD and Basic-PCA when the input data set matrix is tall and fat, whereas SVD and Basic-PCA is more suitable than SPCA when the input matrix is tall and skinny.**

***Keywords*—PCA, SVD, distributed platform, Husky.**

I. Introduction

Large amounts of data offer people the opportunity for extracting valuable insights and retrieve information for further usage. We can use dimensionality reduction algorithms to achieve the goal. Moreover, when the data set is big, only distributed platforms are able to handle the data. Husky is a distributed computing system designed to handle mixed jobs of coarse-grained transformations, graph computing and machine learning. The core of Husky is written in C++ so as to leverage the performance of native runtime. For machine learning, Husky supports relaxed consistency level and asynchronous computing in order to exploit higher network/CPU throughput. During this summer research, LIU Jie and me have implemented Basic-principal Component Analysis, Scalable Probabilistic principal Component Analysis and Singular Value Decomposition on Husky. All of the three algorithms are for dimensionality reduction for big data on distributed platforms. In the following sections, we will introduce the three algorithms we have implemented and compare and discuss their performance and limitations.

II. Analysis of algorithms

In this section, we will introduce and compare the three algorithms we've implemented, i.e. Basic-principal Component Analysis(Basic-PCA), Singular Value Decomposition(SVD) and Scalable Probabilistic principal Component Analysis(SPCA).

A. Basic-PCA

PCA is concerned with explaining the variance/covariance structure of a set of variables through a few linear combinations. Given an N by D input data matrix A with N observed data-points, each data-point consists of D potentially correlated variables $x_1, x_2, ..., x_D$. We look for a transformation of the D x_is into d new variables z_is that are uncorrelated to replace the old variables. In other words, we want to find the directions of a set of new axes which minimizes the sum of squared errors when the original data points are projected on to the new set of axes. We can mathematically prove that the new set of d axes have the same directions as the eigen vectors of $A.T * A$. Essentially, our job is to get the eigen vectors of $A^T * A$. Given the N by D matrix A, we first read the matrix line by line on the cluster consisting of several workers. We make each line an object(a basic unit in Husky). Then we want to get the matrix $C = A^T * A$, i.e.$C = \sum_{n=1}^{n=N} (A_n)^T * A_n$. During this step, each object A_n only needs to do ${A_n}^T * A_n$ and broadcast the result(or send it to a specific worker), C is the sum of the result sent by all the workers. Then we just call an external library called *Eigen* to get the eigen vectors of C. The eigen vector corresponding to the largest eigen value of C is the first principal component of A. Similarly, the eigen vector corresponding to the second largest is the second principal component of A, etc. As we've noticed, the limitation of this algorithm is that when D is large, C, which is D by D, can not fit in the memory of a single worker. In other words, this algorithm is only suitable for a skinny(D small) input matrix.

```
input : $A \in R^{m \times n}$
output: $v_i,\ i = 1, 2, \cdots, d$
$C = A^T * A$ in parallel;
$v_i$ = ComputeEigenVectors($C$), $i = 1, 2, \cdots, d$;
```

Algorithm 1: Basic-PCA

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 $C = A^T A$ 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 $C = A^T A$. Then we collect the matrix C 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 C. 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 $u_i = (1/\sigma_i) A v_i$. If $m < n$, we need to compute $C = AA^T$ 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 C. 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 $C \in R^{n\times n}$ nor $C \in R^{m\times m}$ 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
    $C = A^T * A$ in parallel;
    $v_i$ = ComputeEigenVectors($C$), $i = 1, 2, \cdots, r$;
    ${\sigma^2}_i$ = ComputeEigenValues($C$), $i = 1, 2, \cdots, r$;
    $u_i = (1/\sigma_i)Av_i$ in parallel;
else
    $C = A * A^T$ in parallel;
    $u_i$ = ComputeEigenVectors($C$), $i = 1, 2, \cdots, r$;
    ${\sigma^2}_i$ = ComputeEigenValues($C$), $i = 1, 2, \cdots, r$;
    $v_i = (1/\sigma_i)A^T u_i$ in parallel;
end
```

Algorithm 2: SVD

C. SVD for 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. $C = A^T * A$ 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. SPCA for large and sparse matrices

We adopted the algorithm and pseudo code of SPCA [1] from Elgamal's paper and we also did some refinement. SPCA is a scalable version of Probabilist Principal Component Analysis(PPCA). In this probabilistic approach, 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: $y = C * x + \mu + e$, where C is a $D \times d$ transformation matrix (i.e, the columns of C are the principal components), $x \sim N(0, I)$, μ is the vector mean of y, and $e \sim N(0, ss*I)$ is white noise to compensate for errors.[1] Elgamal makes uses of the sparsity and also makes some refinement on the original PPCA to make it scalable. The pseudo code is shown below.

```
input : $Y \in R^{N\times D}$
output: $C \in R^{D\times d}$
$C = normrnd(D, d)$;
$ss = normrnd(1, 1)$;
$Ym = mean(Y)$ in parallel;
$ss1 = FrobeniusNorm(Y)$ in parallel;
while not $STOP_CONDITION$ do
    $M = C^T * C + ss * I$;
    $CM = C * M^{-1}$;
    $Xm = Ym * CM$;
    $XtX, YtX$ = YtXJob$(Y, Ym, Xm, CM)$ in parallel;
    $XtX += ss * M^{-1}$;
    $C = YtX/XtX$;
    $ss2 = trace(XtX * C^T * C)$;
    $ss3$ = ss3Job$(Y, Ym, Xm, CM, C)$ in parallel;
    $ss = (ss1 + ss2 - 2 * ss3)$ ;
end
```

Algorithm 3: SPCA

There are four parts that involves parallel computation in the algorithm. The first two are done only once. The latter two are done in every iteration. In order to speed up the algorithm, our implementation focus on the code inside the while loop. In **YtXJob**, we do the following computation in parallel: $XtX = X^T * X + ss * M^{-1}$, $YtX = Yc^T * X$, where $X = Yc * C * M^{-1}$ and $Yc = Y - Ym$. First, we compute $X = y * C * M^{-1} - Ym * C * M^{-1}$ instead of computing $X = Yc * C * M^{-1}$ to leverage the sparsity in Y. Then we make use of the intermediate matrix X to compute XtX and YtX. Then we compute $YtX = Y^T * X - Ym^T * X$ instead of computing $YtX = Yc^T * X$ directly to leverage the sparsity in Y. In **ss3Job**, we compute $ss3 = \sum_{n=1}^{N} X_n * C^T * Yc_n^T$. Similarly, to leverage sparsity, we do $X_n * (C^T * Y_n^T) - X_n * C^T * Y_m^T$ since $C^T * Y_n^T$ is small and sparse. For the computation not done in parallel, we just do it in every single worker. Although it keeps the worker working intensively, it actually saves the time of communication between workers.

III. Performance

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

Both SPCA and Basic-PCA was implemented and run on Husky. MLlib-PCA was run on Spark. All tests were run under the same environment. The $353 \times 2K$ input matrix is dense, the other 2 input matrices are sparse.

IV. Conclusion

On one hand, our implementation of SPCA is most suitable when the input data set is sparse and fat. Still, our SPCA on Husky outperforms MLlib-PCA of Spark when the input matrix is $353 \times 2K$ and dense. It's mainly because that when D is high, MLlib is doing CPU intensive computation on a single machine while SPCA is running EM method on a cluster. On the other hand, MLlib-PCA and Basic-PCA are better than SPCA when D is small. When D is small, there is no need doing parallel computation of EM method for eigen vectors and thus solving those on a single machine is better.

Acknowledgment

The authors would like to thank Professor James CHENG, his PhD candidates, LIU Jie and all other people working in the lab.

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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, 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.