Beyond SVD

STA561 (Probabilistic Machine Learning) Final Project

Ilan Man
Department of Statistical Science
Duke University
ilanman@gmail.com

December 4, 2015

1 Introduction

Singular value decomposition plays an important role in high-quality statistical computations and in schemes for data compression and least squares solutions. Many modern computational algorithms are based on singular value computations because the problem of computing the eigenvalues of a general matrix is well-conditioned. However, traditional SVD on a large matrix can be very computationally expensive.

This paper introduces the concept random sampling to approximate SVD calculations. We outline two randomized SVD algorithms and an important lemma which provides a theoretical error bound for mapping a set of points in high dimensional space onto a lower dimensional subspace. We finish with some empirical results showing the speed up gained from using randomized methods vs. classical SVD.

2 Singular Value Decomposition

2.1 Matrix Factorization

Before reviewing SVD, we motivate by discussing matrix factorizations. Matrix factorizations underpin many matrix manipulations commonly found in machine learning and statistical algorithms. Formally, matrix factorization is an operation on a matrix A such that it is factored into two smaller matrices, R and Q:

$$\underset{n\times m}{\mathrm{A}} = \underset{n\times k}{R} \times \underset{k\times m}{Q^T} \tag{1}$$

Where $k \leq \min(n, m)$. Since RQ^T may not be exactly equal to A, this results in a residual term, ε that we seek to minimize. Mathematically, we want $||A - RQ^T||_F \leq \varepsilon$. This norm, known as the Frobenius norm, is a common matrix norm, where $||A||_F^2 = \mathrm{trace}(A^TA)$.

One approach is to use partial-QR decomposition. This is an iterative algorithm where A factors into QR such that R is *upper triangular*. This makes it much easier to multiply against, and reduces the computational expense. At iteration t, we get:

$$A^{(t)} = Q^{(t)}R^{(t)} + E^{(t)}$$

where $E^{(t)}$ is an error residual that satisfies $||E^{(t)}||_F \leq \varepsilon$. The more steps that we take in this algorithm the closer $||E^{(t)}||_F$ get to the theoretical ε. More iterations also results in higher operational complexity.

There are multiple ways to factorize a matrix, each providing an advantage and disadvantage [1]. This paper focuses on singular value decomposition.

2.2 SVD algorithm

We begin by stating the **SVD Theorem**: *If A is an $m \times n$ matrix, then A has a singular value decomposition.*

The SVD of a real $m \times n$ matrix A is a factorization of the form $A = U\Sigma V^T$ where:

- U is an m x k orthogonal matrix of left singular vectors of A
- Σ is an k x k diagonal matrix of ordered singular values of A, i.e. $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_k$.

- V^T is an k x n orthogonal matrix of right singular vectors of A

$$\Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_k \end{bmatrix}$$

Computing the SVD is a fundamental linear algebra objective with a wide range of applications. Two of the most common are low-rank approximations and solutions to least squares. We briefly discuss these two topics.

2.2.1 Low Rank Approximation

For high-dimensional data, we often want to simplify the data so that traditional machine learning and statistical techniques can be applied. However, crucial information intrinsic in the data should not be removed under this simplification. A widely used method for this purpose is to approximate the data matrix, A, with a matrix of lower rank.

Formally, we seek:

$$\min_B ||A - B||_F : \text{rank}(B) = k, \text{where } k \leq \text{rank}(A) \tag{2}$$

Note that the minimization objective has a very similar structure to that above for matrix factorizations.

2.2.2 Least Squares Approximation

Solving equations of the form $A\mathbf{x} = \mathbf{b}$ is an extremely common task. We want to minimize:

$$||A\mathbf{x} - \mathbf{b}||_2 \tag{3}$$

The best minimizer is $\mathbf{x} = (A^TA)^{-1}A^T\mathbf{b}$, the so-called normal equations. To solve this, one might form the matrix $K = A^TA$ and use a Cholesky or QR decomposition. Note that the most expensive part of this is computing A^TA, making this prohibitive in some cases. Another approach is using QR factorization, typically via the Householder algorithm. This approach, while more numerically stable, is roughly twice as expensive as solving the normal equations [2].

SVD is the most numerically stable solution of these methods, especially when A is rank-deficient, because it exposes eigenvalues. However it can be much more computationally expensive.

It's application to various problems motivates why technicians are focused on coming up with faster SVD routines.

2.2.3 Comments

Note the $m \times n$ matrix A can be decomposed into a sum of r rank-one matrices:

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T \tag{4}$$

Moreover, by the Eckart-Young theorem, the best 2-norm or Frobenius-norm approximation of rank k ($0 \leq k \leq n$) to A is given by

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$$

And in fact,

$$||A - A_k||_F = \sigma_{k+1}$$

As mentioned above, SVD exposes the eigenvalues of the matrix A. Eigenvalues are found by solving the characteristic polynomial, which is inherently an iterative process, because the roots cannot be found in polynomial time. Specifically, the computational cost of performing SVD is $O(nm \times \min(n, m))$. This becomes very expensive for large, dense matrices.

One solution is to compute a truncated-SVD. This is equivalent to selecting the top k singular values from Σ and associated singular vectors.

Another solution to the compute a partial-QR decomposition on the large matrix A. Then on the resulting factor, compute an SVD. This approach scales with $O(kmn)$.

Note that each of these approaches requires random access to memory, i.e. the matrix must be stored in RAM. If the matrix is stored out of core, the methods mentioned above can become very slow. This motivates new techniques for factorizing lage A.

3 Randomized SVD

3.1 Motivation

As mentioned earlier, traditional SVD algorithms can be computationally expensive for large scale problems. SVD also assumes that the matrix can fit in RAM. This is not well suited for missing or noisy data, matrices that are very large or distributed/parallel computation.

To combat these and other items, randomized matrix algorithms were developed. A low-rank approximation can be obtained by randomly sampling columns of A according to a probability distribution that depends on the Euclidean norms of the columns. We describe the algorithm below.

3.2 RVSD Algorithm

The following is a generic formulation of the randomized SVD, along with the corresponding Python code, to show how trivial implementation is:

Given: $m \times n$ matrix A and a desired rank k:

1. Draw an $n \times k$ Gaussian random matrix Ω with elements $\omega_{ij} \overset{iid}{\sim} N(0,1)$

   ```
   Omega = np.random.randn(m, k)
   ```

2. Compute $H = A \times \Omega$

   ```
   H = np.dot(A,Omega)
   ```

3. Construct $Q \in R^{m\times k}$ with columns forming an orthonormal basis for the range of H

   ```
   Q, R = np.qr(H)
   ```

4. Form the $k \times n$ matrix, $B_k = Q^T A$.

   ```
   B = np.dot(Q.T,A)
   ```

5. Return B_k

We are interested in comparing the error $\varepsilon_k = ||A - B_k||_F$ to the theoretically minimum error σ_{k+1}.

3.2.1 Comments

RSVD methods allow us to reorganize the calculations to exploit matrix arithmetic and distributed computer networks. As a result, these methods are well suited for parallel implementation. The bottleneck in these calculations is $A\Omega$ which ends up being very parallelizable.

This, **fixed-rank** algorithm, assumes that we know the target rank, k, of our matrix. In some applications, this makes sense, such as genomic identification, where practioners have an idea of the effective population size. In other applications, a low rank k is not known in advance (or perhaps may not exist). The number of columns, l, that the algorithm needs in order to minimize ε is slightly larger than k. We call this discrepency the oversampling parameter p. You can think of this as a fudge parameter, usually set at 5 or 10. Therefore we write $k = l + p$ in the algorithm above.

To set a reasonable value for l, we can run a search algorithm that finds some l corresponding to the smallest error. This problem formulation is known as the **fixed-precision** problem, because we are solving for a precision in ε, rather than a rank.

3.2.2 Numerical vs. Statistical perspective

One of the criticisms of the fixed-precision problem, as stated by numerical analysts, is that the error they seek to minimize is the training, or in-sample error. This doesn't provide useful information about the generalization, or out-of-sample error, which is the key metric we typically want to minimize. More useful would be to find some lower rank l such that we can minimize the out-of-sample error. This way of thinking is motivated by the idea that our data matrix, A, is produced by some data-generating process. We seek to find a way to approximate this process in a lower dimensional space. This is the way a statistician would go about solving the problem. However, we'll set aside this discussion for another paper.

3.3 Johnson-Lindenstrauss

Upon seeing the RSVD algorithm, a natural question to ask is how are we confident that we can map our original matrix A onto a smaller matrix B_k in a way that doesn't disrupt the structure of our data? We want to project onto a lower-dimensional subspace, while reducing all distances by the same common factor, leaving the relative ordering of distances unchanged. Furthermore, we want to do this in polynomial time. The following lemma provides the theoretical basis for this:

Johnson-Lindenstrauss Lemma:
Let $\varepsilon \in (0, \frac{1}{2})$. Let $Q \in R^d$ be a set of n points and $k = \frac{20\,log\,n}{\varepsilon^2}$. There exists a map $f : \mathbb{R}^d \to \mathbb{R}^k$ such that for all $u, v \in Q$:

$$(1-\varepsilon)||u-v||^2 \leq ||f(u)-f(v)||^2 \leq (1+\varepsilon)||u-v||^2$$

Two key observations:

- For some dimension $k < n$, with high probability, there exists a mapping that does not change the pairwise distance between any two points by more than a small factor $(1 \pm \varepsilon)$
- k is randomly found, in polynomial time

This result allows us to apply random projections in the RSVD algorithm and not be concerned with maintaining the relationship of our original data. A proof of the lemma can be found here [3].

Ultimately, a projection from $\mathbb{R}^d$ to $\mathbb{R}^k$ takes $O(kd)$ time. For large, d, even this could be slow. As a result there have been recent advances in speeding up this process; one such advance is the Fast JL-transform [4].

3.4 Power Scheme

Section 3.2 outlined the generic RSVD algorithm. One enhancement is to include power iterations. This involves repeated multiplication of the original matrix A and computing the QR factorization of the result. We can think of the problem as follows:

$$A^k = U\Sigma^k V^T$$

Recall that we can write A as $\sum_{i=1}^{n} \lambda_i u_i v_i^t$. Calculating A^k is analogous to raising λ_i to the kth power. We can imagine $\lambda_i > 1$ being magnifide, and $\lambda_i < 1 \to 0$. Visually:

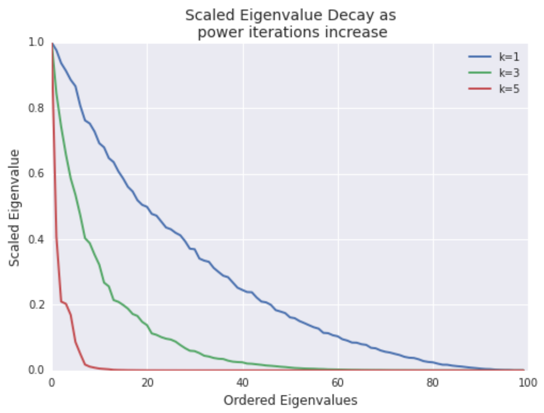

Figure 1: Separating Eigenvalues with power iterations

We can compute A^2, A^3, etc... and perform QR factorization on the resulting matrix which tends to be more accurate than performing it on A, because our important eigenvalues are amplified. This saves a lot of computational expense.

An enhancement to the algorithm is as follows (replacing k with $l + p$, the oversampling parameter):

1. Draw an $n \times (l + p)$ Gaussian random matrix Ω with elements $\omega_{ij} \overset{iid}{\sim} N(0, 1)$

   ```
   Omega = np.random.randn(m, l + p)
   ```

2. For $i : 1, ..., q$

   ```
   for i in range(iters):
   ```

   (a) Compute $H = A \times \Omega$

   (b) Compute $H = (AA^T)^q H$, by iterating q times

   ```
   H = np.dot(A,np.dot(A.T,H))
   ```

3. Compute a QR factorization of H. Q is an $n \times (l + p)$ matrix whose columns form an orthonormal basis for H.

   ```
   Q, R = np.qr(Omega)
   ```

4. Form the $(l + p) \times n$ matrix, $B_k = Q^T A$.

   ```
   B = np.dot(Q.T,A)
   ```

5. Return B_k

This algorithm requires $2q + 1$ times as many matrix–vector multiplies as the one without power iterations, but is far more accurate in situations where the singular values of A decay slowly [5].

3.5 Empirical Results

Below we present empirical results comparing the RSVD algorithm outlined in this paper to Python's built-in SVD library, in `numpy`. We tested using a dense matrix of random Gaussian elements:

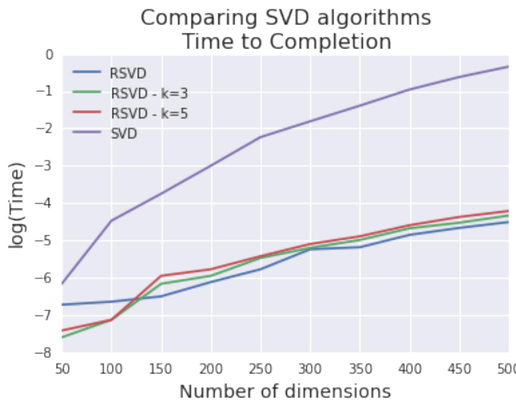

Figure 2: RSVD computes much faster

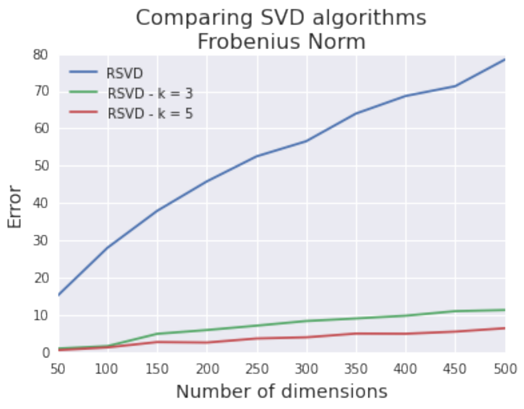

Figure 3: Using power iterations improves error

The speed-up gained from using Randomized algorithms is orders of magnitude faster than the out of the box svd method. And using a modest power iteration, the error is minimal. The benefits of using Randomized methods is readily apparent.

4 Conclusion

Classical SVD is a very useful and ubiquotous algorithm for solving least squares and low-rank approximations. There are downsides, however, specifically when the matrix being factorized is very large. If we know that there exists a latent rank in a smaller subspace, we can use randomized algorithms to efficiently approximate a lower rank. We discussed a randomized SVD algorithm, and showed how trivial it is to code in Python. Then we highlighted an important theoretical finding that provides error bounds on this low dimensional projection. Finally, we showed how power iterations increases our accuracy many fold.

There are many more Randomized techniques, including Adaptive Randomized SVD, which I hope to explore in further research.

References

[1] http://www.dcsc.tudelft.nl/bdeschutter/pub/rep/02012.pdf

[2] http://math.uchicago.edu/ may/REU2012/REUPapers/Lee.pdf

[3] http://ttic.uchicago.edu/ gregory/courses/LargeScaleLearning/lectures/jl.pdf

[4] https://www.cs.princeton.edu/ chazelle/pubs/FJLT-sicomp09.pdf

[5] http://arxiv.org/pdf/0909.4061v2.pdf