Randomized Numerical Linear Algebra and its applications

Haoyang Li, supervised by Ery Arias-Castro

June 2020

1 Abstract

This thesis mainly studies some algorithms in randomized numerical linear algebra, from simple randomized inner product and randomized matrix multiplication to more complicated ones including randomized low rank matrix approximation with random projection and randomized low rank approximation for positive semidefinite matrix with sketch method. These algorithms contributed to improving the efficiency of some linear algebra algorithms or saving computer memory by sacrificing limited accuracy. This thesis also provides some analysis about the error bounds for these algorithm. Finally, this thesis shows the application of randomized low rank matrix approximation in Principle Component Analysis and Support Vector Machine.

Here is the link for my code on github:
https://github.com/HaoyangLi0507/Randomized-Numerical-Linear-Algebra.git

2 Randomized Matrix Multiplication

2.1 Randomized Inner Product

We know that the inner product of two vectors a and b in R^n is $a^\top b = \sum_{i=1}^n a_i b_i$. The running time for this process is $O(n)$. When we deal with very large vectors and the variance of their elements are not large, it is reasonable to choose a small sample independently and uniformly at random with replacement from the population of the product $a_i b_i$ for $i \in \{1, \ldots, n\}$ to estimate the value of $a^\top b$.

Algorithm 1: Randomized inner product

```
Data: vector $a, b \in R^n$ and sampling size $c$.
Result: $x^\top y$ as an approximation for $a^\top b$, $x, y \in R^c$.
Initialize $xy = 0$;
for $i$ in 1 to $c$ do
    pick a number $k \in \{1, \ldots, n\}$ independently and uniformly at random with replacement;
    set $x_i = \sqrt{\frac{n}{c}} a_k$ and $y_i = \sqrt{\frac{n}{c}} b_k$;
    set $xy = xy + x_i y_i$
end
Return $xy$;
```

Now we have an algorithm with running time $O(c)$. Analysis for this algorithm is simple. We can see that

$$E\left[x^\top y\right] = a^\top b,$$

because

$$\begin{aligned} E\left[x^\top y\right] &= E\left[\sum_{i=1}^c x_i y_i\right] \\ &= \sum_{i=1}^c E\left[\frac{1}{n}\sum_{j=1}^n \sqrt{\frac{n}{c}} a_j \sqrt{\frac{n}{c}} b_j\right] \\ &= \sum_{i=1}^c \frac{1}{c} E\left[\sum_{j=1}^n a_j b_j\right] \\ &= \sum_{i=1}^c \frac{1}{c} E\left[a^\top b\right] \\ &= a^\top b. \end{aligned}$$

2.2 Randomized Matrix Multiplication

Taking a closer look at the matrix multiplication, we can see that for some $A \in R^{m\times n}$ and $B \in R^{n\times p}$,

$$AB = \sum_{i=1}^n A_{*i} B_{i*}.$$

This sum is in $R^{m\times p}$. Similar to the randomized inner product, a reasonable randomized approximation for the exact product AB can be the sum of c ($c \ll n$) rescaled terms randomly sampled from n terms in $\sum_{i=1}^{n} A_{*i}B_{i*}$ independently and with replacement, according to a probability distribution set $\{p_k\}_{k=1}^{n}$.

$$AB \approx \sum_{t=1}^{c} \frac{1}{cp_{i_t}} A_{*i_t}B_{i_t*}$$

To write this approximation in matrix multiplication form, we can use sampling matrix S. Let $S \in R^{n\times c}$ be a matrix that $S_{i_t t} = \frac{1}{\sqrt{cp_{i_t}}}$ and other elements equal to 0. Let $X = AS$ and $Y = S^\top B$,

$$XY = ASS^\top B \approx AB.$$

Algorithm 2: Randomized Matrix Multiplication [2]

```
Data: input matrix $A \in R^{m\times n}$ and $B \in R^{n\times p}$, positive integer $c \ll n$, probability
      distribution $\{p_k\}_{k=1}^{n}$.
Result: an approximation to AB.
initiate $XY$ be an empty $m \times n$ matrix;
for $t$ in 1 to $c$ do
    Randomly select $i_t$ from $[1, n]$ with probability $p_{i_t}$;
    Calculate $tempo$ = rescaled outer product $\frac{1}{\sqrt{cp_{i_t}}}A_{*i_t} \times \frac{1}{\sqrt{cp_{i_t}}}B_{i_t*}$;
    $XY = XY + tempo$;
end
Return $XY$;
```

Note that S is not explicitly created in this algorithm. This algorithm has time complexity $O(mcp)$ while the exact matrix multiplication's is $O(mnp)$.
The following lemma and theorem are from [2].

Lemma 2.1. *Let X and Y be constructed as described above, then for an arbitrary pair of (i,j), we have $E\left[(XY)_{ij}\right] = (AB)_{ij}$ and $Var\left[(XY)_{ij}\right] \le \frac{1}{c}\sum_{k=1}^{n} \frac{A_{ik}^2 B_{kj}^2}{p_k}$.*

Proof. For $t \in \{1, 2, \dots, c\}$, define $R_t = \frac{A_{ii_t}B_{i_tj}}{cp_{i_t}}$, which is the (i, j) term of $\frac{A_{*i_t}B_{i_t*}}{cp_{i_t}}$ and A_{*i_t} and B_{i_t*} are the randomly chosen column and row. We can see that $(XY)_{ij} = \sum_{t=1}^{c} R_t$. Then since we can choose A_{*i_t} and B_{i_t*} from all the n pairs in the sum for the exact result, we can see that $E[R_t] = \sum_{k=1}^{n} p_k \frac{A_{ik}B_{kj}}{cp_k} = \frac{1}{c}\sum_{k=1}^{n} A_{ik}B_{kj} = \frac{1}{c}(AB)_{ij}$. Thus

$$E\left[(XY)_{ij}\right] = E\left[\sum_{t=1}^{c} R_t\right] = \sum_{t=1}^{c} E[R_t] = \sum_{t=1}^{c} \frac{1}{c}(AB)_{ij} = (AB)_{ij}.$$

For variance, we know that $Var[R_t] = E[R_t^2] - E[R_t]^2$. As $E[R_t]^2 \ge 0$, $Var[R_t] \le E[R_t^2]$.

$$E[R_t^2] = \sum_{k=1}^{n} p_k \frac{A_{ik}^2 B_{kj}^2}{c^2 p_k^2} = \frac{1}{c^2}\sum_{k=1}^{n} \frac{A_{ik}^2 B_{kj}^2}{p_k}$$

$$Var[(XY)_{ij}] = \sum_{t=1}^{c} Var[R_t] \le \sum_{t=1}^{c} E[R_t] = \frac{1}{c}\sum_{k=1}^{n} \frac{A_{ik}^2 B_{kj}^2}{p_k}$$

$\square$

Next we discuss the choice of the probability distribution set $\{p_k\}_{k=1}^n$ and we bound the error in terms of Frobenius norm. If we use identical probability for every pair of A's column and B's row, the approximation might behave poorly because it might miss the pair having the multiplication result with relatively large Frobenius norm. This is important especially when dealing with cases like A has an outstandingly large column and the corresponding row in B is not outstandingly small.

Thus we use the probability set

$$p_k = \frac{\|A_{*k}\|_2 \|B_{k*}\|_2}{\sum_{k'=1}^n \|A_{*k'}\|_2 \|B_{k'*}\|_2}.$$

Note that if we use uniform probability, which is $p_k = \frac{1}{n}$ for all $k \in \{1, \ldots, n\}$, the error bound will be

$$E\left[\|AB - XY\|_F^2\right] = \frac{n}{c}\sum_{k=1}^n \|A_{*k}\|_2^2 \|B_{k*}\|_2^2 .$$

Computing the weights can be more precise in general cases (especially when dealing with the matrices with outstanding columns), but the computation of those weights also costs time.

Theorem 2.2. *If we construct X and Y as described previously and approximate AB by XY, and we choose*

$$p_k = \frac{\|A_{*k}\|_2 \|B_{k*}\|_2}{\sum_{k'=1}^n \|A_{*k'}\|_2 \|B_{k'*}\|_2} \tag{1}$$

for all $k \in \{1, \ldots, n\}$, then

$$E\left[\|AB - XY\|_F^2\right] \leq \frac{1}{c}(\sum_{k=1}^n \|A_{*k}\|_2 \|B_{k*}\|_2)^2.$$

Proof. We know that

$$\begin{aligned} E\left[\|AB - XY\|_F^2\right] &= \sum_{i=1}^m \sum_{j=1}^n E\left[\left\|(AB)_{ij} - (XY)_{ij}\right\|_F^2\right] \\ &= \sum_{i=1}^m \sum_{j=1}^n [Var(\left\|(AB)_{ij} - (XY)_{ij}\right\|_F) - E\left[\left\|(AB)_{ij} - (XY)_{ij}\right\|_F\right]^2]. \end{aligned}$$

According to lemma 2.1, we know that $E\left[\left\|(AB)_{ij} - (XY)_{ij}\right\|_F\right]^2 = 0$. Thus

$$\begin{aligned} E\left[\|AB - XY\|_F^2\right] &= \sum_{i=1}^m \sum_{j=1}^n Var(\left\|(AB)_{ij} - (XY)_{ij}\right\|_F) \\ &\leq \sum_{i=1}^m \sum_{j=1}^n \left[\frac{1}{c}\sum_{k=1}^n \frac{A_{ik}^2 B_{kj}^2}{p_k}\right] \end{aligned}$$

If we use uniform probability, which is $p_k = \frac{1}{n}$ for $k \in \{1, \ldots, n\}$,

$$E\left[\|AB - XY\|_F^2\right] \le \frac{1}{c}\sum_{k=1}^{n}\frac{1}{p_k}\|A_{*k}\|_2^2\|B_{k*}\|_2^2 \quad = \frac{n}{c}\sum_{k=1}^{n}\|A_{*k}\|_2^2\|B_{k*}\|_2^2$$

If we choose p_k according to (1),

$$\begin{aligned}E\left[\|AB - XY\|_F^2\right] &\le \frac{1}{c}\sum_{k=1}^{n}[(\frac{\sum_{k'=1}^{n}\|A_{*k'}\|_2\|B_{k'*}\|_2}{\|A_{*k}\|_2\|B_{k*}\|_2}) * (\sum_{i=1}^{m}\sum_{j=1}^{n}A_{ik}^2B_{kj}^2)] \\ &= \frac{1}{c}\sum_{k=1}^{n}[(\sum_{k'=1}^{n}\|A_{*k'}\|_2\|B_{k'*}\|_2) * (\|A_{*k}\|_2\|B_{k*}\|_2)] \\ &= \frac{1}{c}(\sum_{k=1}^{n}\|A_{*k}\|_2\|B_{k*}\|_2)^2.\end{aligned}$$

To see why we choose p_k, we can use Lagrange multiplier to solve the system that

$$\text{minimize} \sum_{i=1}^{m}\sum_{j=1}^{n}\left[\frac{1}{c}\sum_{k=1}^{n}\frac{A_{ik}^2B_{kj}^2}{p_k}\right] \text{ subject to} \sum_{k=1}^{n}p_k = 1.$$

The specific proof is in [2] under section 4.1. □

2.3 Numerical Experiment

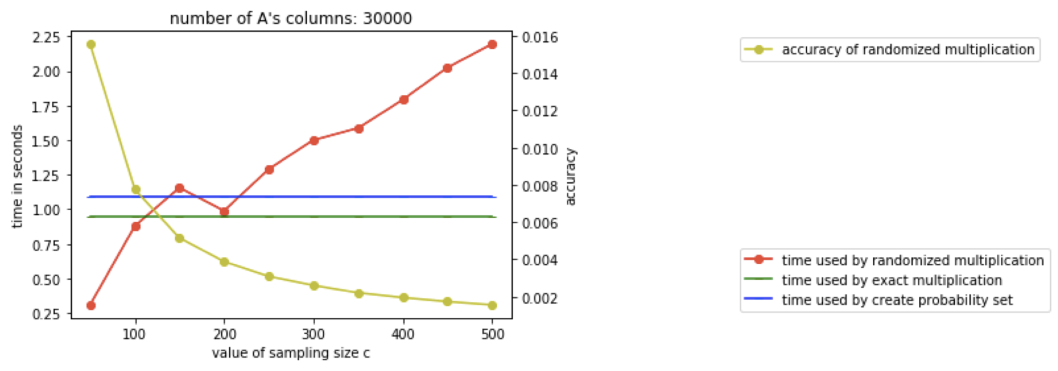
In this section, I did some numerical experiments in Python to see the actual behaviors of this randomized matrix multiplication in terms of accuracy and efficiency.

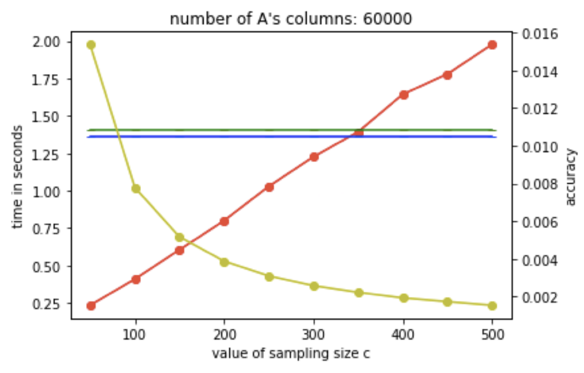
I randomly generated a series of matrices A and B with increasing sizes. Every element of A and B is between 0 and 1. I fixed $m = 700$ and set $n \in \{3 \times 10^4, 6 \times 10^4, \ldots, 6 \times 10^5\}$. I use the probability set described above and $c = 50, 100, 150, ..., 500$ to do the experiments. For each value of c, I repeated the experiment for 40 times. I took the average time used to measure the efficiency and took the average of $\frac{\|AB-XY\|_F}{\|AB\|_F}$ to measure accuracy.

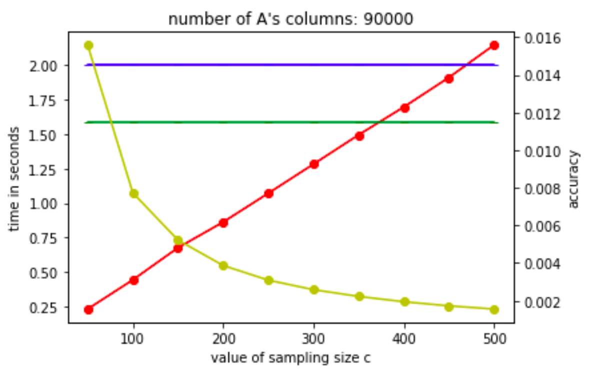
Remark 1. *From my experiments (some of the results are not shown), I found that this algorithm can improve the efficiency greatly with the pair of original matrices having a large n and a relatively small m and p. This is actually kind of trivial because we are sampling from A's n columns (and corresponding rows in B), not from A's m rows or B's p columns, and when we use a small c, we can save a lot of time for the calculation.*

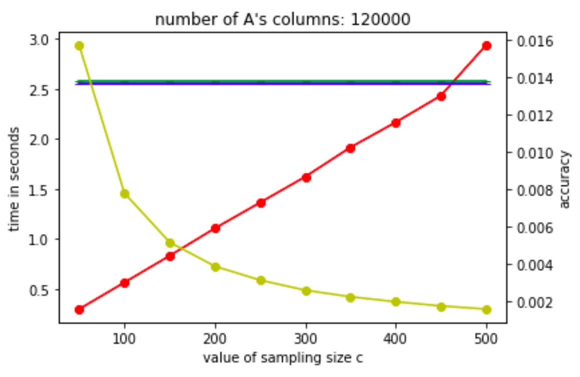
Remark 2. *From the experiments, we can see that computation time of this algorithm increases approximately linearly with the increase of c. The errors decrease fast at first few c values and then more and more slowly, approximately according to a log function. In my experiments, the elbows for accuracy are at around 300 to 400 for c, no matter how large n is.*

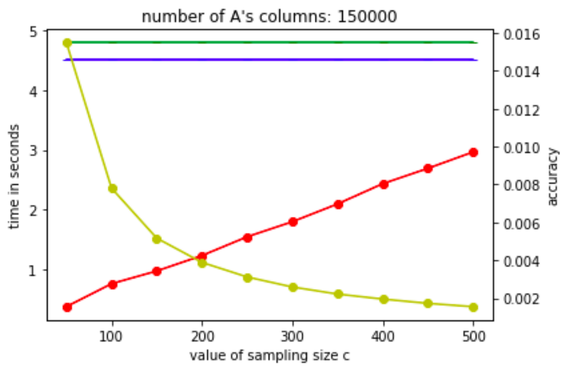
Remark 3. *We can also see that calculating the probability set $\{p_k\}$ is also time consuming. When dealing with small n (like below 24000) or very large n (like above 57000), sometimes the computer even used more time to calculate the probability set than to calculate exact matrix multiplication. I believe one of the reason is that my code is not optimized. But users should pay attention to the efficiency of calculating probability set. This is also why sometimes uniform probability is a good choice when users are not strict about accuracy. We can see that the Randomized Matrix Multiplication algorithm itself runs pretty fast, especially for small c values.*

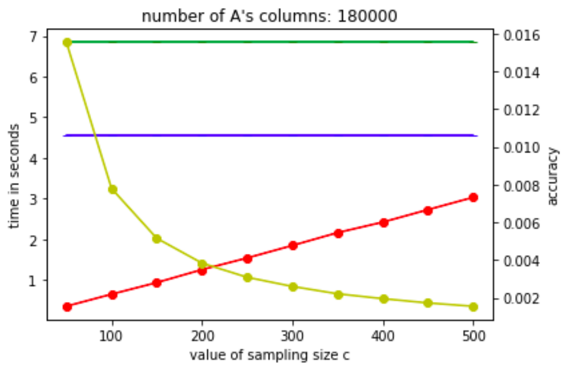

number of A's columns: 30000
time in seconds
accuracy
value of sampling size c
accuracy of randomized multiplication
time used by randomized multiplication
time used by exact multiplication
time used by create probability set

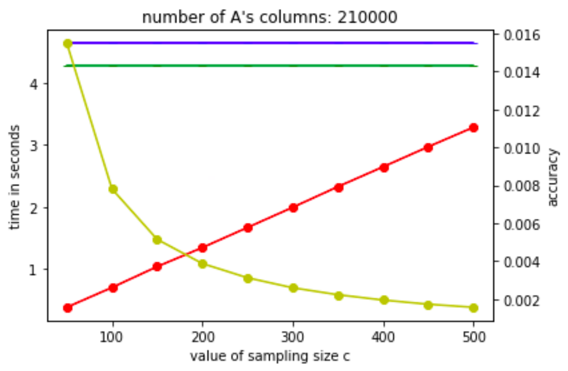

number of A's columns: 60000
time in seconds
accuracy
value of sampling size c

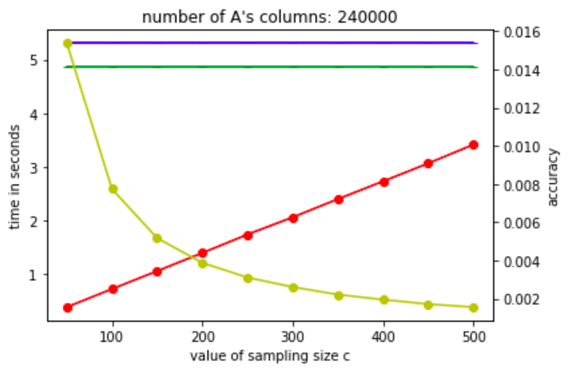

number of A's columns: 90000
time in seconds
accuracy
value of sampling size c

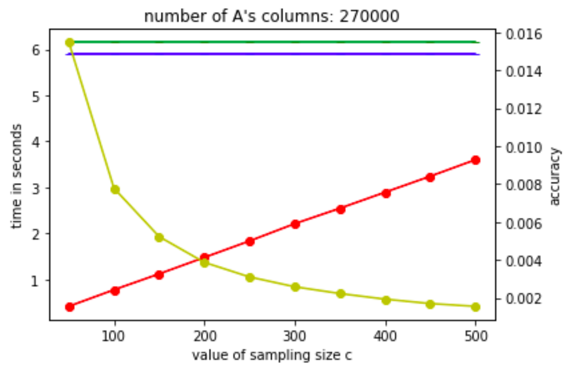

number of A's columns: 120000
time in seconds
accuracy
value of sampling size c

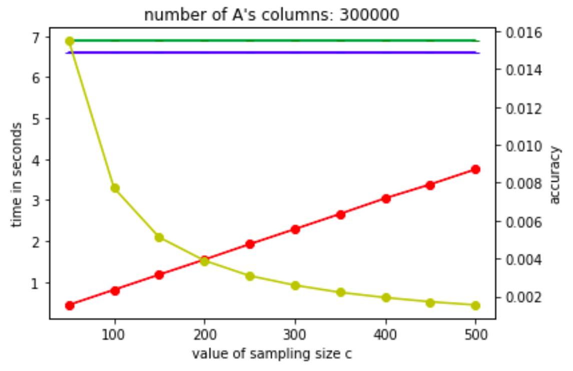

number of A's columns: 150000
time in seconds
accuracy
value of sampling size c

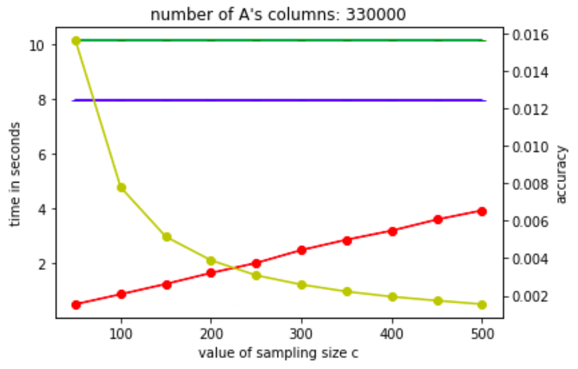

number of A's columns: 180000
time in seconds
accuracy
value of sampling size c

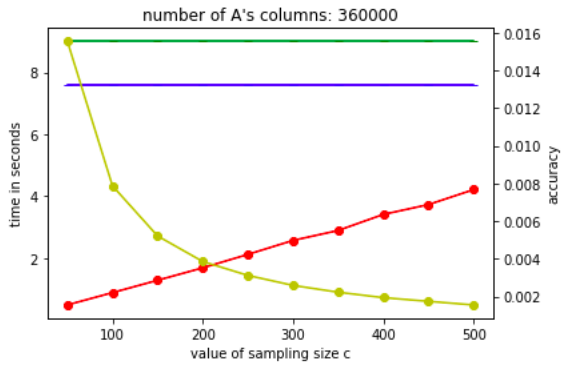

number of A's columns: 210000
time in seconds
accuracy
value of sampling size c

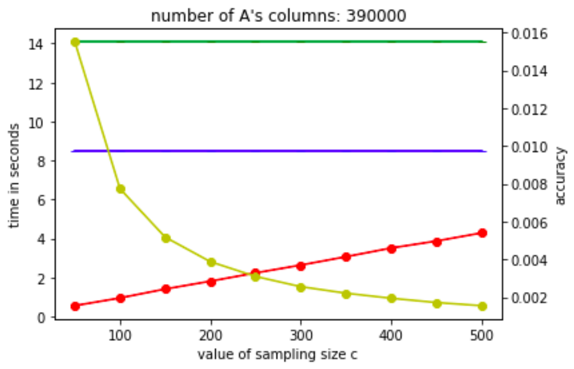

number of A's columns: 240000
time in seconds
accuracy
value of sampling size c

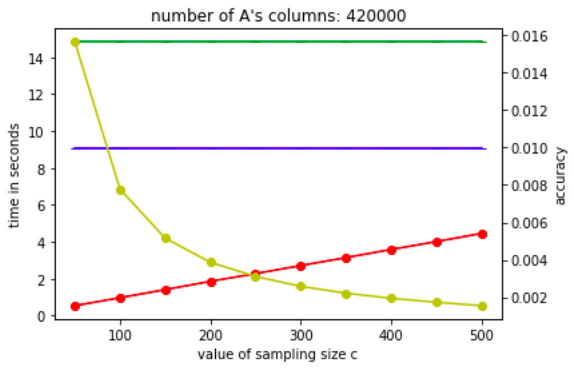

number of A's columns: 270000
time in seconds
accuracy
value of sampling size c

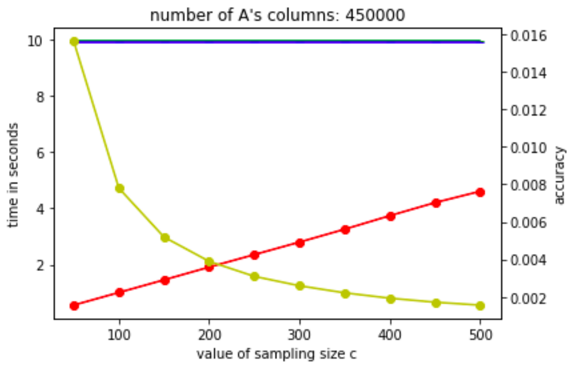

number of A's columns: 300000
time in seconds
accuracy
value of sampling size c

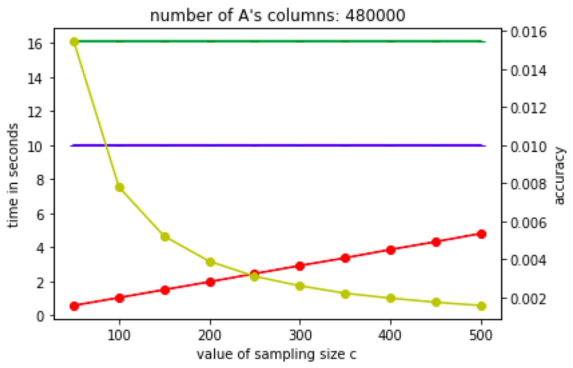

number of A's columns: 330000
time in seconds
accuracy
value of sampling size c

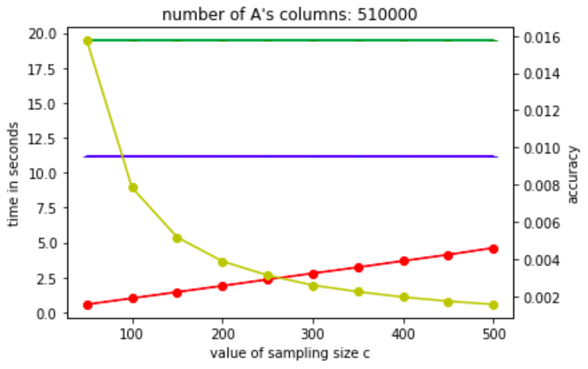

number of A's columns: 360000
time in seconds
accuracy
value of sampling size c

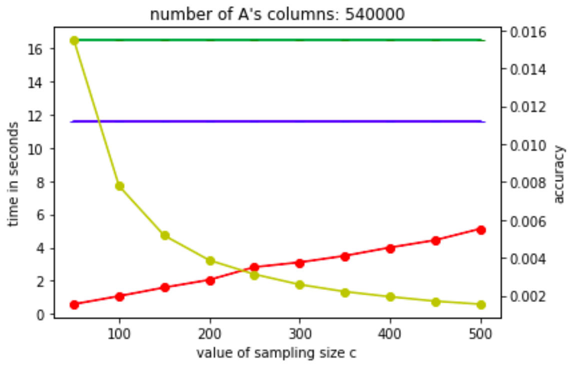

number of A's columns: 390000
time in seconds
accuracy
value of sampling size c

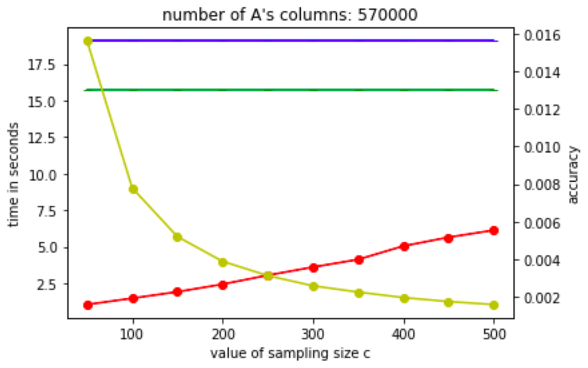

number of A's columns: 420000
time in seconds
accuracy
value of sampling size c

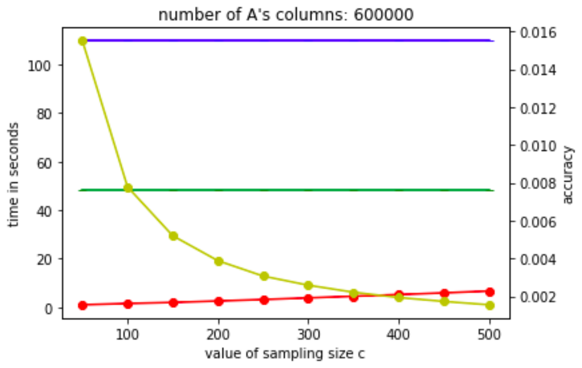

number of A's columns: 450000
time in seconds
accuracy
value of sampling size c


number of A's columns: 480000
time in seconds
accuracy
value of sampling size c


number of A's columns: 510000
time in seconds
accuracy
value of sampling size c


number of A's columns: 540000
time in seconds
accuracy
value of sampling size c


number of A's columns: 570000
time in seconds
accuracy
value of sampling size c


number of A's columns: 600000
time in seconds
accuracy
value of sampling size c

3 Randomized Low Rank Matrix Approximation

3.1 Singular Value Decomposition

In low rank matrix approximation, one popular method is Singular Value Decomposition (SVD). SVD can decompose the original matrix $A \in R^{m\times n}$ into three matrices:

$$A = U\Sigma V^\top,$$

where $U \in R^{m\times m}$, $\Sigma \in R^{m\times n}$, $V \in R^{n\times n}$. Suppose $m > n$. U and V are orthogonal matrices. Columns of U are left singular vectors of A and columns of V are right singular vectors of A. The only nonzero elements in Σ are those on its diagonal $(\Sigma_{11}, \Sigma_{22}, ..., \Sigma_{nn})$ and they are A's singular values. Let σ_1, σ_2,..., σ_n be A's singular values with $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_n$, $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ be the corresponding left and right singular vectors, we can also write

$$A = \sum_{i=1}^{n} u_i \sigma_i v_i^\top.$$

We know that the best rank k approximation in Frobenius norm for A is

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

In matrix notation, we let Σ_k be the $k \times k$ matrix with top k singular values of A in descending order, U_k and V_k be the matrices of left and right singular vectors corresponding to the singular values in Σ_k,

$$A_k = U_k \Sigma_k V_k^\top. \tag{2}$$

In some of the following algorithms, we compare the computed approximating matrix with A_k to analyze the accuracy of the approximating matrix.

Computing SVD can be accurate but expensive. Therefore, we have following randomized algorithms to compute the approximation faster but with some loss of accuracy.

3.2 Low Rank Matrix Approximation with Random Projection

3.2.1 Intuition and Algorithm

One way of doing randomized low rank matrix approximation is using the idea of random projection, which can be done in two stages. The first stage is to approximate the range of input matrix A by multiplying A by a randomly created matrix Ω (for example a Guassian random matrix) with much less columns. This would work because in a high-dimensional space there exist much more almost orthogonal directions than orthogonal directions, and vectors with random directions might be sufficiently close to orthogonal (from [1]). Therefore, multiplying A by Ω can be seen as approximately projecting A onto a lower dimensional space. Then we find an orthonormal basis Q of this product and Q should have same number of columns with Ω. We will have Q = orthonormal basis of $A\Omega$,

$$A \approx QQ^\top A.$$

The next stage is to use this Q to help perform SVD. We set $B = Q^\top A$ and B will have fewer rows than A. Then we can do

$$U_B \Sigma_B V_B^\top = B.$$

Let Σ_{B_k} be the matrix of B's top k singular values and U_{B_k}, V_{B_k} be the matrices of corresponding left and right singular vectors,

$$U = QU_{B_k}$$

$$A \approx U \Sigma_{B_k} V_{B_k}^\top.$$

This two-stage algorithm idea comes from [5]. A complete introduction of this random projection and random sampling based approximation can be found in [5].

In this algorithm, we choose the random matrix Ω according to randomized Hadamard transformation (this algorithm comes from [2]). We can also use Fourier Transformation or other structured randomized transformations here. See [5] section 4.6 for more details.

Algorithm 3: Randomized Low Rank Matrix Approximation [2]

```
Data: $A \in R^{m\times n}$, a rank parameter $k \ll min(m,n)$, an error parameter $\epsilon \in [0, \frac{1}{2})$.
Result: $\tilde{U}_k \in R^{m\times k}$ as an approximation to the $U_k$ in (2).
Let $c$ be a value such that $c \geq c_0 \frac{k \ln n}{\epsilon^2}(\ln \frac{k}{\epsilon^2} + \ln\ln n)$;
Initialize $S$ be an empty matrix;
for $t$ in 1 to $c$ do
    Select a column $e_i$ from $n \times n$ identity matrix $I_n$ uniformly at random (i.i.d. with
      replacement), multiplied by $\sqrt{\frac{n}{c}}$, append it to $S$;
end
Let $D$ be a diagonal matrix, $D_{ii} = 1$ with probability $\frac{1}{2}$ and $D_{ii} = -1$ with probability $\frac{1}{2}$;
Let $H$ be a $n \times n$ normalized Hadamard transform matrix;
Let $\Omega = DHS \in R^{m\times c}$;
Compute a orthonormal basis $Q$ of $A\Omega$;
Let $B = Q^T A$;
Compute $U_B, \Sigma_B, V_B = SVD(B)$ and take top $k$ singular values and corresponding left
  and right singular vectors $U_{B_k}, \Sigma_{B_k}, V_{B_k}$;
Return $\tilde{U}_k = QU_{B_k}$;
```

Note that c_0 in this algorithm is an integer. In [2], the choice of c_0 is not mentioned. In my experiments, I use 0.01. Also if we fix n, we can see that $c = O(k \ln k)$. This algorithm has time complexity $O(mnc)$ (from [2]).

3.2.2 Analysis about Randomized Low Rank Matrix Approximation

Section 6 in [2] provides the following theorem for analyzing the error bounds of this algorithm.

Theorem 3.1. *Let $A \in R^{m\times n}$, k be a rank number, $\epsilon \in [0, \frac{1}{2})$. If we choose $c \geq c_0 \frac{k \ln n}{\epsilon^2}(\ln \frac{k}{\epsilon^2} + \ln\ln n)$, we can have*

$$\left\| A - \tilde{U}_k \tilde{U}_k^T A \right\|_F \leq (1+\epsilon)\|A - A_k\|_F$$

with probability at least 0.85 and $\tilde{U}_k$ is the matrix returned by the Randomized Low Rank Matrix Approximation.

The complete proof of this theorem is quite long and can be found in [2].

Remark 4. *The probability for finding an approximation as good as the theorem stated is at least 0.85. However, if we run this algorithm for m times, the probability of not finding such a good approximation is at most* 0.15^m*, which approaches to 0 fast. Also comparing the approximation with the original matrix is relatively not too time consuming (O(mn) for comparing the Frobenius norm). Thus running a few times of this algorithm and picking the best approximation will produce a good approximation efficiently with a very large probability.*

3.2.3 Numerical Experiment

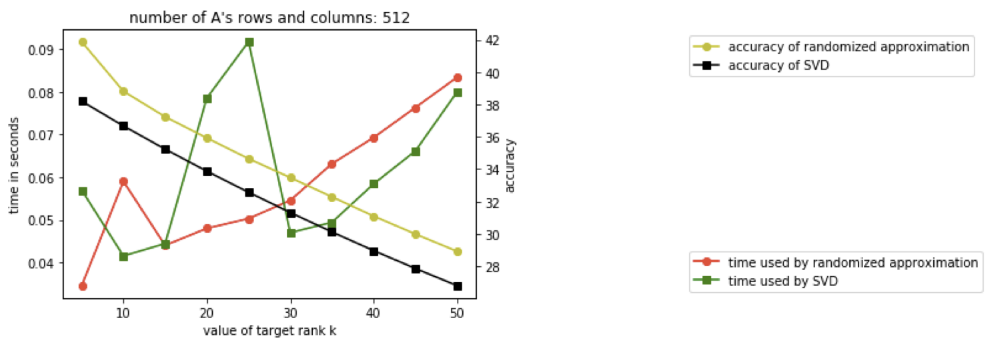
To see the behaviors of this randomized low rank approximation algorithm, I randomly generated a series of matrices A for experimenting. For simplicity, I used square matrices. Also I only implemented the random Hadamard transformation in a simplified way that multiplying the Hadamard transformation matrix whose size is a power of 2. So when I experimented with different sizes, I used $\{2^9, 2^{10}, \ldots, 2^{13}\}$ as the numbers of columns and rows of A. Then I also experimented with different sparcity of A, which is defined by $\frac{\text{number of nonzero elements in } A}{\text{number of } A\text{'s elements}}$. For both size related experiments and sparcity related experiments, I used target rank $k \in \{5, 10, \ldots, 50\}$. I chose $c_0 = 0.005$ and $\epsilon = 0.25$. For accuracy, I used $\left\|\tilde{A} - A\right\|_F$ as error, where $\tilde{A}$ was the approximation gotten from the randomized low rank matrix approximation algorithm. For each approximation, I repeated 15 times and took the average of running time and errors.

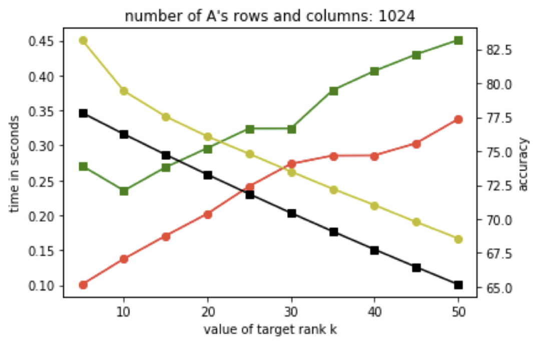
Remark 5. *For the experiments with different sizes of A but same sparcity (*$= 1$*), we can see that running time of both randomized low rank matrix approximation and SVD increases approximately linearly with the increase of k. The randomized method behaves better in terms of efficiency when the size of A is large. The error of randomized method decreases fast for the first few k and then it also decreases approximately linearly with the increase of k.*

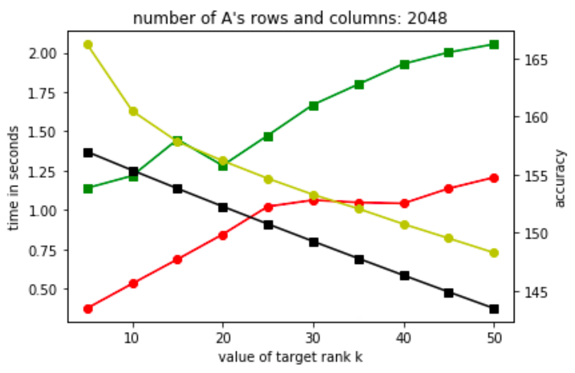
Remark 6. *We can see that in the experiments with same size but different sparcity, when the matrix is very sparse, the errors of both exact SVD and randomized low rank matrix approximation increases as the k increases. I think one of the reason can be the ill condition of input matrix A. But this should arise attention. Also compared with the running time with different input matrices' sizes, it does not change much with the change of input matrices' sparcity.*

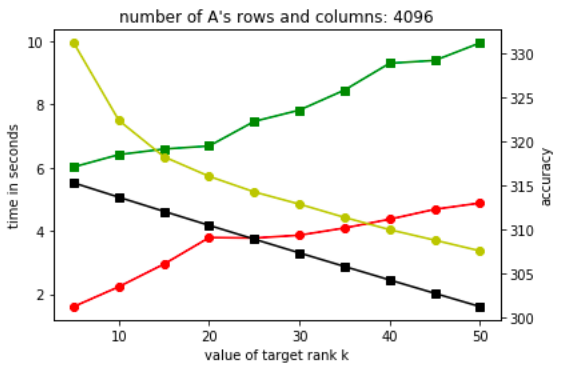
Remark 7. *Note that generating random matrix also costs time. Generating a* 8192×8192 *random matrix with Python function numpy.random.rand() uses 1.13s, which cannot be ignored. However, generating random number from normal distribution in Python is almost free as it cost so little time, like less than 0.06s for the* 8192×8192 *random matrix.*

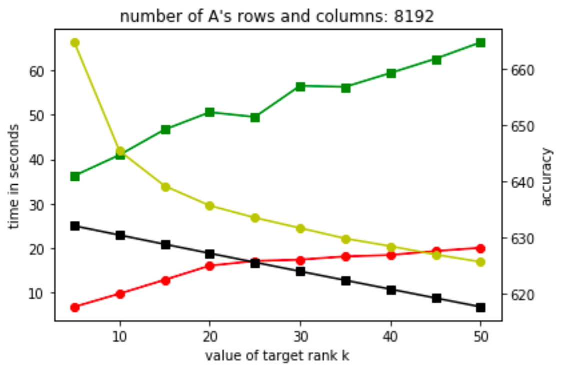
The following figures show the results of experiment with different sizes but same sparcity ($= 1$).


number of A's rows and columns: 512
time in seconds
accuracy
value of target rank k
accuracy of randomized approximation
accuracy of SVD
time used by randomized approximation
time used by SVD

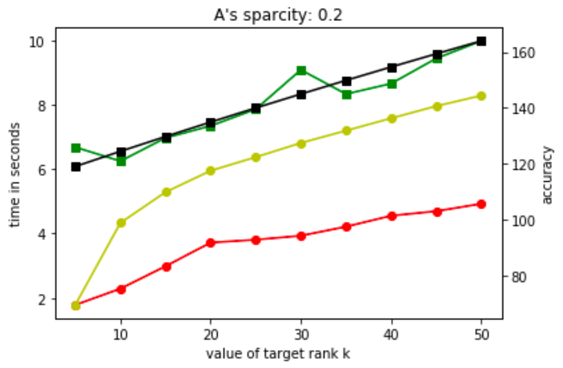

number of A's rows and columns: 1024
time in seconds
accuracy
value of target rank k

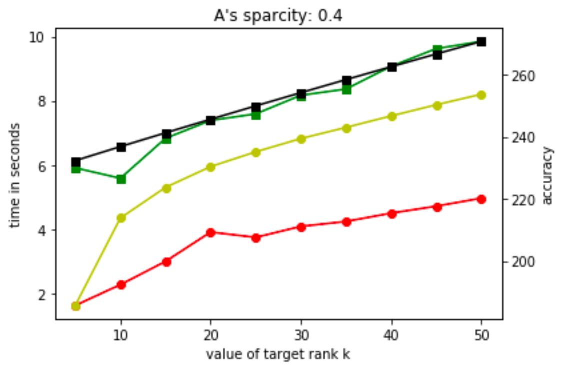

number of A's rows and columns: 2048
time in seconds
accuracy
value of target rank k

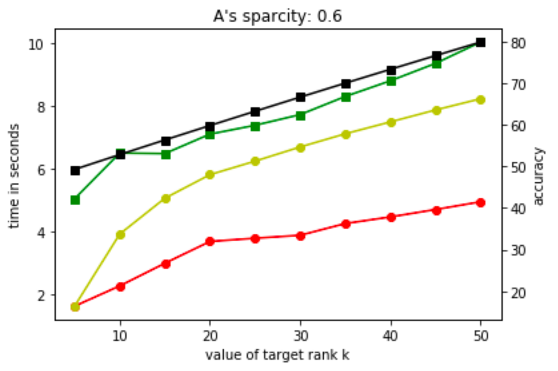

number of A's rows and columns: 4096
time in seconds
accuracy
value of target rank k

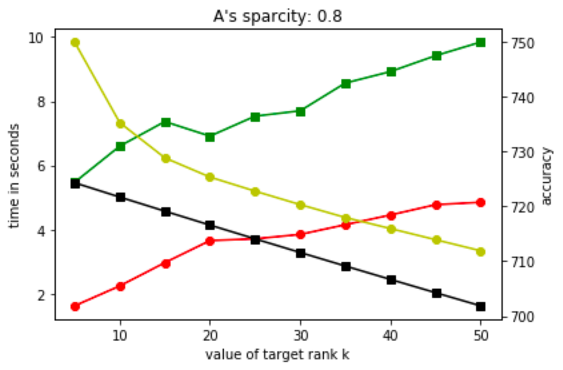


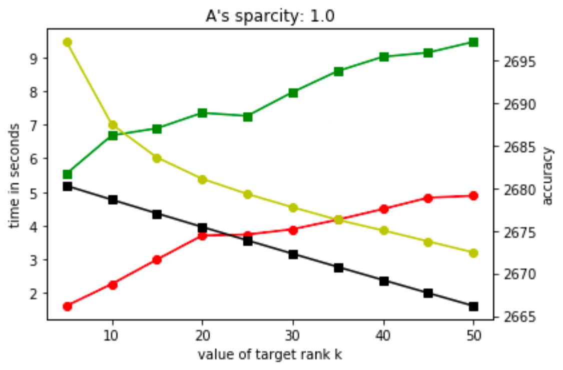
The following figures show the results of experiment with same size (4096) but different sparcity.











3.3 Low Rank Approximation for Positive Semidefinite Matrix

3.3.1 Introduction

Positive semidefinite matrix (PSD matrix) is an important kind of matrix in today's data world. For example, when we want to analyze a group of data like stock price, we might want to consider about the correlation between each of the instance in the dataset. The matrix with all the correlation values is a PSD matrix. If we want to use Principle Component Analysis or other methods to do some analysis about this matrix, a low rank approximation can help save a lot of time and memory. For another example, the gram matrix in Support Vector Machine is also a PSD matrix and a low rank approximation for gram matrix can speed up the Support Vector Machine algorithm. Therefore, in many cases finding a low rank approximation for the PSD matrix can be helpful.

3.3.2 Algorithms and Analysis

A thesis ([4]) I read provides one way to approximate a PSD matrix A. It is different from the one we introduced before that using the idea of random projection: let $S \in R^{n\times l}$ be a sampling matrix with $l \ll n$ and let $C = AS$ and $W = S^\top AS$, we use

$$A \approx CW^\dagger C^\top$$

($W^\dagger$ represent the Moore-Penrose pseudoinverse of W) as the PSD sketch of A with rank at most l.

This PSD sketch has several different kinds of extensions, regarding different choices of sampling matrix S. The first one is Nystrom extension, which samples the columns from A with unifrom probability (i.i.d.). Note that this extension needs to use the coherence of the matrix and I will talk more about this in analysis section. The second one is Gaussian extension, which uses $n \times l$ sampling matrix S with each entry coming from normal distribution. The thesis ([4]) also provides other extensions, like the one using Fourier transformation, in section 6.

Algorithm 4: Randomized Low Rank Matrix Approximation for PSD Matrix [4]

```
Data: PSD matrix $A \in R^{n\times n}$, sampling number $l$.
Result: $\tilde{A}$ as an approximation to $A$.
Nystrom: pick $l$ columns from identity matrix $I_n$ to form sampling matrix $S$;
Gaussian: create a $n \times l$ sampling matrix $S$ by randomly generating each of its elements
  from normal distribution;
Let $C = AS$ and $W = S^\top AS$;
Return $\tilde{A} = CW^\dagger C$;
```

3.3.3 Error Bounds for This Sketch Method

The thesis ([4]) provides a complete analysis of this sketch method's behaviors in terms of accuracy and efficiency. I will just state the theorem about the accuracy here and the proof can be found in the thesis.

Theorem 3.2. *With Nystrom extension, if we choose $l \geq 2\mu\epsilon^{-2}k\log(\frac{k}{\delta})$, we can have*

$$\left\|A - CW^\dagger C^\top\right\|_F \leq \|A - A_k\|_F + (\frac{\sqrt{2}}{\delta\sqrt{1-\epsilon}} + \frac{1}{(1-\epsilon)\delta^2})Tr(A - A_k)$$

with probability at least $1 - 4\delta$. μ is the measure of coherence of the matrix, which is defined as $\mu(A) = \frac{n}{k}max_i(P_A)_{ii}$, where P_A is the projection onto the range of A.

Theorem 3.3. *With Gaussian extension, if we choose $l \geq (1+\epsilon^2)k$, we can have*

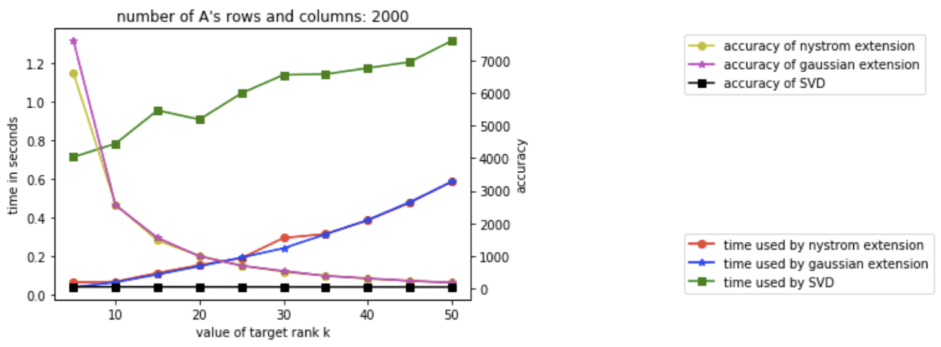
$$\left\|A - CW^\dagger C^\top\right\|_F \leq \|A - A_k\|_F + c_0\sqrt{\|A - A_k\|_2\, Tr(A - A_k)} + c_1\|A - A_k\|_2 + \frac{c_2}{\sqrt{k}}Tr(A - A_k)$$

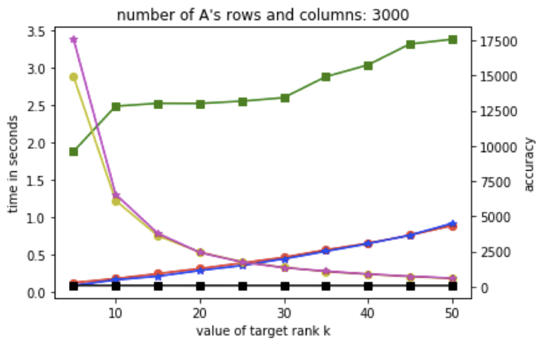
with probability at least $1 - 2k^{-1} - 4e^{\frac{-k}{\epsilon^2}}$. c_0, c_1, c_2 are integers relating to ϵ. $c_0 = 11\epsilon + 544\epsilon^2$, $c_1 = 815\epsilon^2$, and $c_2 = 91\epsilon$.

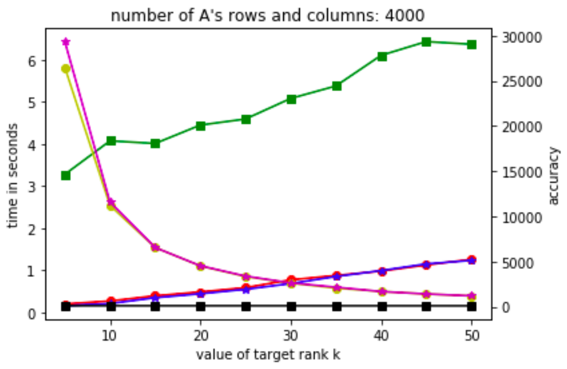
3.3.4 Numerical Experiment

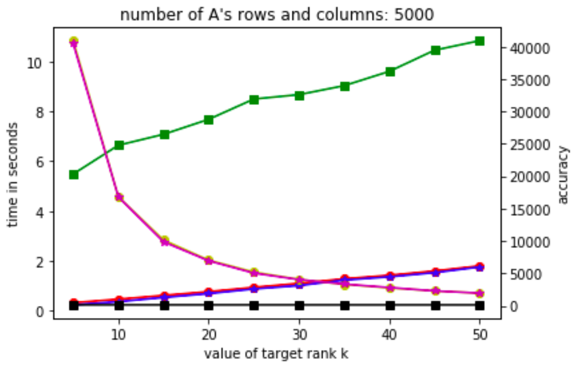
To see the behaviors of this randomized low rank approximation algorithm for PSD matrix, I randomly generated a series of matrices A for experimenting. When I experimented with different sizes, I used $\{2000, 3000, \ldots, 8000\}$ as the numbers of columns and rows of A. Then I also experimented with different sparcity of A, which is defined by $\frac{\text{number of nonzero elements in } A}{\text{number of } A\text{'s elements}}$. For both size related experiments and sparcity related experiments, I used target rank $k \in \{5, 10, \ldots, 50\}$. I choosed $\delta = 0.1$ and $\epsilon = 0.9$. For accuracy, I used $\left\|\tilde{A} - A\right\|_F$ as error, where $\tilde{A}$ was the approximation gotten from the randomized low rank matrix approximation for PSD matrix algorithm. For each approximation, I repeated 10 times and took average of running time and errors.

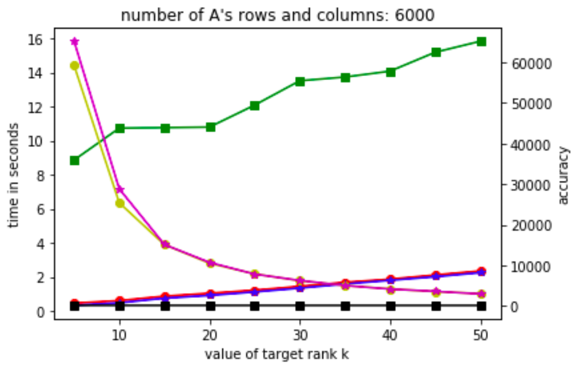
The following figures show the results of experiment with different sizes but same sparcity ($= 1$).

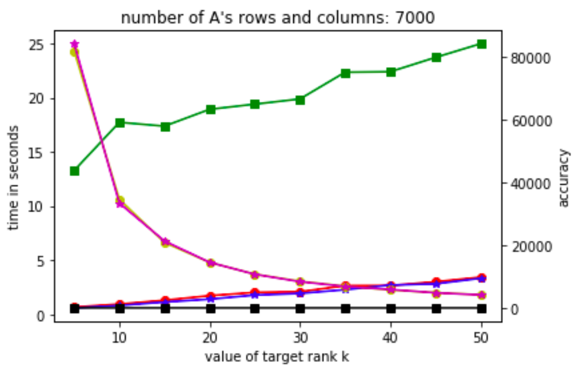


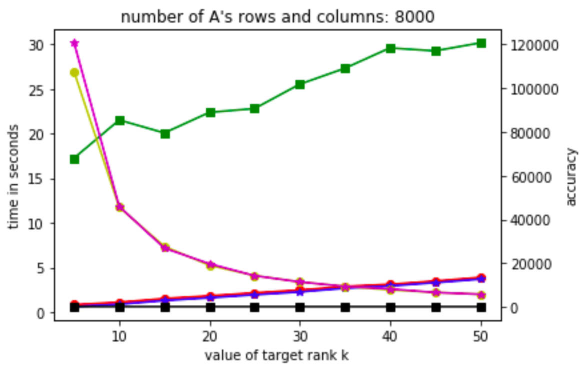



number of A's rows and columns: 4000
time in seconds
accuracy
value of target rank k

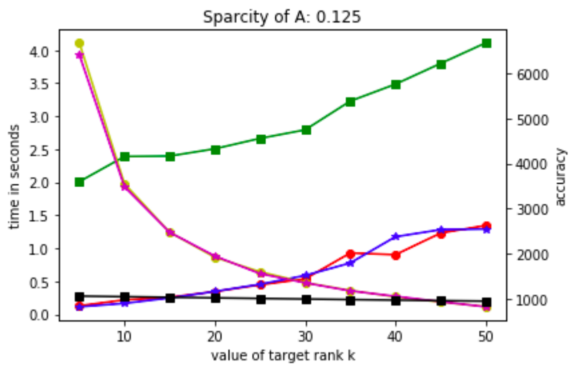

number of A's rows and columns: 5000
time in seconds
accuracy
value of target rank k

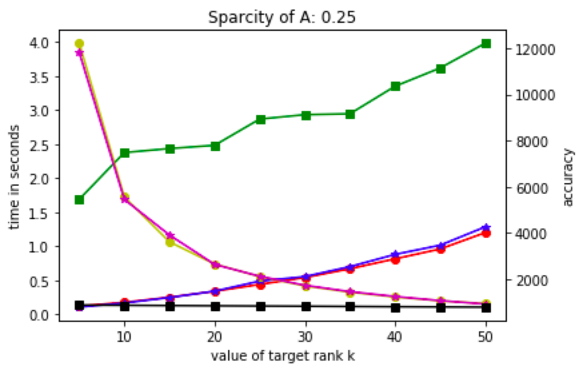

number of A's rows and columns: 6000
time in seconds
accuracy
value of target rank k

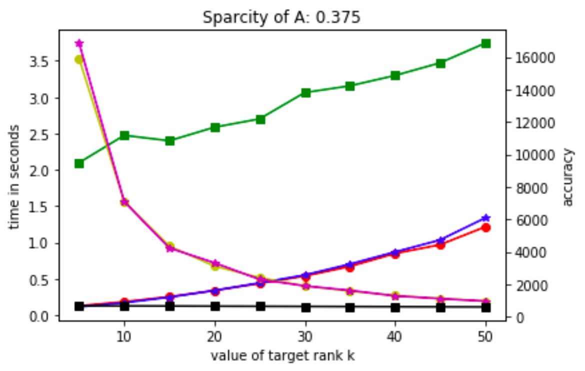


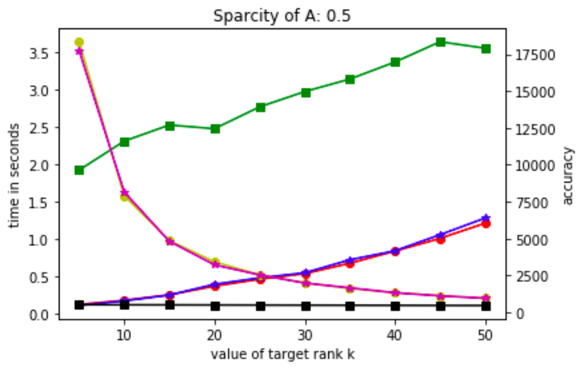


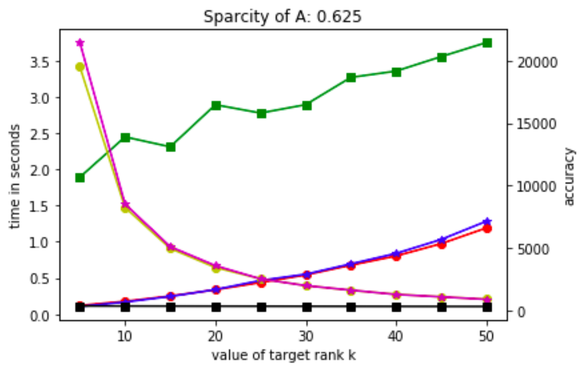
The following figures show the results of experiment with same sizes (3000) but different sparcity (= 1).

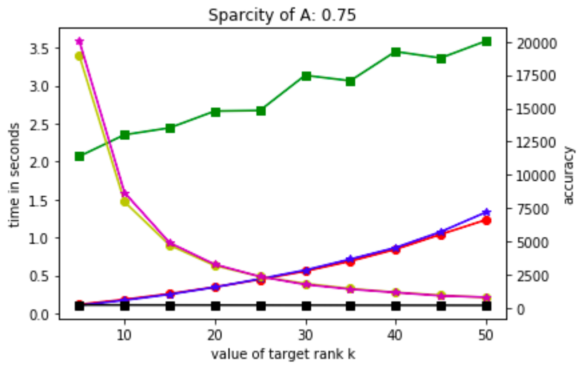


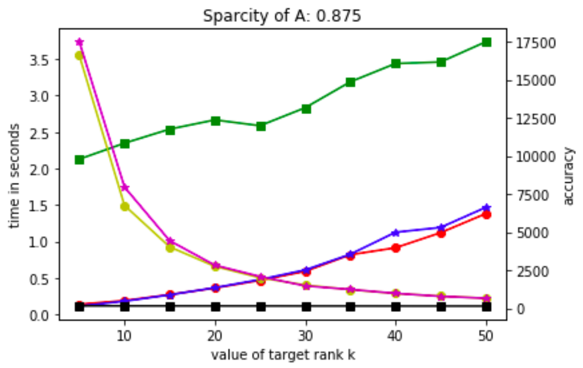



Sparcity of A: 0.375
time in seconds
accuracy
value of target rank k


Sparcity of A: 0.5
time in seconds
accuracy
value of target rank k


Sparcity of A: 0.625
time in seconds
accuracy
value of target rank k





Remark 8. *First from the experiments with different sizes but same sparcity, we can see that the randomized low rank approximation for PSD matrix with both extensions did run faster than SVD. The behaviors of both extensions are almost the same in terms of efficiency and accuracy. According to the range of l given in the theorem, Gaussian extension requires a smaller l than Nystrom extension, especially for some large k. However, when I used the relatively small l for Gaussian extension, the error was large. Therefore, I used the same l for both of the algorithms. Running time of the randomized methods increases almost linearly with the increase of k, and the error decreases approximately according to a log function as k increases. The elbows of accuracy are at around 35 for all sizes and sparcity of A.*

3.4 Randomized Principle Component Analysis

3.4.1 Introduction

In data science and machine learning, when we do classification, we usually have many variables for data. On the one hand, if we want to visualize the data, it is impossible for us to visualize the data with more than three variables because we live in a three-dimensional world and we don't know how to plot the data in a higher-dimensional space. On the other hand, higher dimension and rank mean harder to processe the data, causing memory or efficiency issues. But simply dropping some variables is not ideal because it ignores the relationship between the variables. Therefore, we have Principle Component Analysis (PCA) here to help us create a few new variables, which are the combination of original variables, and find the most important ones, i.e. the ones explain the variance of our data best.

A common way to calculate PCA is first centered the original matrix A so that each row of A has mean $= 0$. Then we calculate the covariance matrix $A^\top A$ and calculate

$$U, \Sigma, V^\top = \mathrm{SVD}(\mathrm{A}^\top \mathrm{A}).$$

Suppose the diagonal elements of Σ are in the descending order and the vectors in U are arranged according to the order of the diagonal elements of Σ. The top k vectors in U are the k principle directions that we want. Then we can transform A by calculating AU_k, where U_k is the matrix of the k principle directions.

I will introduce 2 ways of calculating PCA, one of them using randomness, in this section.

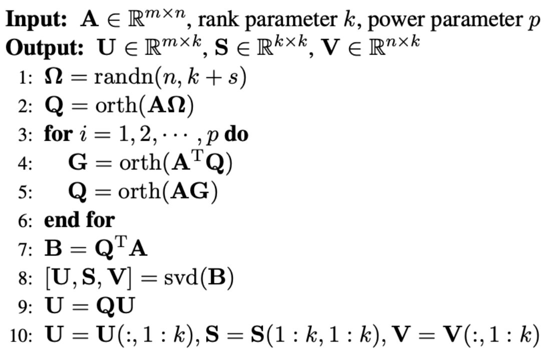
The first one is from my supervisor Ery Arias-Castro. Without calculating SVD of $A^\top A$, we calculate the SVD of A: $U\Sigma V^\top = \text{SVD(A)}$. We can see that

$$A^\top A = V\Sigma U^\top U\Sigma V^\top = V\Sigma V^\top.$$

Therefore, if we order the rows of $V^\top$ according to the order of sorted singular values, $V^\top$'s top k rows are actually principle directions in transpose. Let V_k be the matrix with the top k principle directions. We can transform A by calculating AV_k, which has same effect as the method described above.

The next algorithm is from [3] that actually the top k left singular vectors of A are the k principle components we want (before rescaled by sigular values). Therefore, finding a approximation to the top k singular vectors with a randomized method can be an efficient way to calculate an approximation of exact PCA.

$$A_k V_k = U_k \Sigma_k V_k^\top V_k = U_k \Sigma_k$$



```
Input: $\mathbf{A} \in \mathbb{R}^{m\times n}$, rank parameter $k$, power parameter $p$
Output: $\mathbf{U} \in \mathbb{R}^{m\times k}$, $\mathbf{S} \in \mathbb{R}^{k\times k}$, $\mathbf{V} \in \mathbb{R}^{n\times k}$
 1: $\mathbf{\Omega} = \text{randn}(n, k+s)$
 2: $\mathbf{Q} = \text{orth}(\mathbf{A\Omega})$
 3: for $i = 1, 2, \cdots, p$ do
 4:     $\mathbf{G} = \text{orth}(\mathbf{A}^\text{T}\mathbf{Q})$
 5:     $\mathbf{Q} = \text{orth}(\mathbf{AG})$
 6: end for
 7: $\mathbf{B} = \mathbf{Q}^\text{T}\mathbf{A}$
 8: $[\mathbf{U}, \mathbf{S}, \mathbf{V}] = \text{svd}(\mathbf{B})$
 9: $\mathbf{U} = \mathbf{QU}$
10: $\mathbf{U} = \mathbf{U}(:, 1:k), \mathbf{S} = \mathbf{S}(1:k, 1:k), \mathbf{V} = \mathbf{V}(:, 1:k)$
```

Figure 1: Algorithm from [3]

3.4.2 Experiment

I used the data found on Kaggle.com ([1] under Data Reference). The data is used for classification and has 7352 rows and 561 columns, which means 561 variables. I used PCA to reduce the dimension to 2 with 2 functions described above on Python. These two algorithms were implemented by myself so the code might run slowly because my code was not optimized.

The run time for PCA with calculating A's SVD was 0.16s, and randomized PCA only took 0.03s. However, $\frac{\|\text{result from PCA}-\text{result from randomized PCA}\|}{\|\text{result from PCA}\|} = 2$, which means the result from randomized algorithm was relatively far from the result from exact PCA. In this special case the randomized algorithm is not that optimal, but I will do more experiments on this part and make updates on my github.

3.5 Low Rank Approximation for PSD Matrix Used in Support Vector Machine

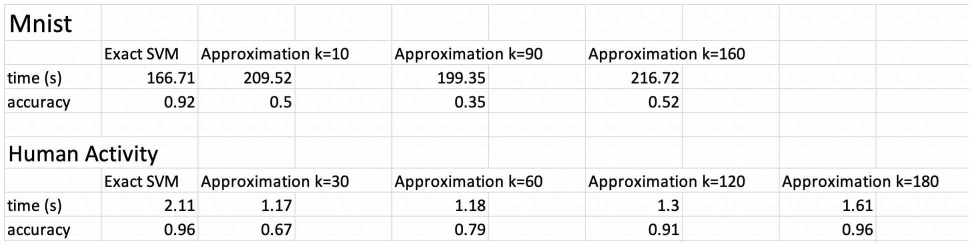
3.5.1 Introduction

Support Vector Machine(SVM) is a popular machine learning algorithm for classification. In the running process, SVM with linear kernel needs to calculate based on a gram matrix, which is a PSD matrix. Therefore, inspired by [6], I think probably replacing the gram matrix by a low rank approximation can help speed up SVM's calculation process and save memory space when dealing with super large dataset.

3.5.2 Experiment

I used two datasets from Kaggle.com to run the experiments ([1] and [2] under Data Reference). Mnist contains data about images, trying to label the numbers in the images. Human Activity Recognition contains data about human, trying to classifying human's activities according to the data. Mnist data has 42000 rows and 785 columns (one of the column is for y value). I splitted the Mnist data into train and test datasets. Train set has 80% of the whole dataset while test has the rest 20%. I standardized the data before using it and I recorded the accuracy of the model by the percentage of correct classifications in test dataset. I used gaussian extension with different k values to do the experiments. The time I recorded included the time used by approximation.

Remark 9. *From the result we can see that for Mnist data, low rank approximation did a bad job that costing more time and having a lower accuracy than standard SVM. But for Human Activity, it did help increase the runing speed of SVM while keep a relatively good accuracy. My naive idea is that low rank approximation is not that suitable for image recognition problem, but I will do more experiments about this and update it on my github.*



Mnist

	Exact SVM	Approximation k=10	Approximation k=90	Approximation k=160
time (s)	166.71	209.52	199.35	216.72
accuracy	0.92	0.5	0.35	0.52

Human Activity

	Exact SVM	Approximation k=30	Approximation k=60	Approximation k=120	Approximation k=180
time (s)	2.11	1.17	1.18	1.3	1.61
accuracy	0.96	0.67	0.79	0.91	0.96

Figure 2: Result from SVM

4 Data Reference

[1]: mnist-svm-m4, Nikhil Sai
https://www.kaggle.com/jnikhilsai/data

[2]: Human Activity Recognation with Smartphones, UCI Machine Learning
https://www.kaggle.com/uciml/human-activity-recognition-with-smartphones

References

[1] Ella Bingham and Heikki Mannila. Random projection in dimensionality deduction: Applications to image and text data. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 245–250, 2001.

[2] Petros Drineas and Michael W Mahoney. Lectures on randomized numerical linear algebra. *The Mathematics of Data*, 25:1, 2018.

[3] Xu Feng, Yuyang Xie, Mingye Song, Wenjian Yu, and Jie Tang. Fast randomized pca for sparse data. *arXiv preprint arXiv:1810.06825*, 2018.

[4] Alex A Gittens. *Topics in Randomized Numerical Linear Algebra.* PhD thesis, California Institute of Technology, 2013.

[5] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[6] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *Advances in Neural Information Processing Systems*, pages 1177–1184, 2008.

5 Acknowledgement

Here I want to thanks my supervisor Ery Arias-Castro. He provided me much help that without him I could not finish this project. I also want to thanks my parents providing me support of doing this.