Randomized Linear Algebra Algorithms for Very Over-constrained Least-squares Problem and its Application in Composed Year Prediction of Songs

Jin Chengneng, Qin Hengle

June 29, 2020

Abstract

We have done some research in the field of randomized linear algebra algorithms, specifically about the very over-constrained least-squares problem. We implement several efficient and relative-error-bounded matrix sketching algorithms, which generate sketching matrixes that are significantly smaller than the original matrix but approximate it well. In experiments, we evaluate the approximated rate of the sketching matrix by matrix multiplication and apply those algorithms to predict the composed year of songs with Least Squares Regression. Moreover, we discuss about how to choose a proper sketching size in practice and propose an improved sketching algorithm.

1 Introduction

Matrix computation, such as singular value decomposition (SVD) and least-squares (LS), plays an important role in industry. However, with the rapid development of big-data machine learning and large-scale statistic, those traditional algorithms which are time and memory expensive, are not efficient enough. To get an relative-error approximated solution in reasonable time, randomized algorithms for large-scale matrix computation have been proposed.

We will pay attention to the fundamental least-squares problems. To solve the problem $min||Ax - b||_2$, $A \in \mathbb{R}^{n\times d}$ and $b \in \mathbb{R}^n$ are given, there are several traditional methods, e.g., Cholesky decomposition, QR decomposition and SVD. The time complexity are all $O(nd^2)$. However, it's too time and memory expensive when $n \gg d$, which is called very over-constrained least-squares problem.

2 Related Work

Michael W. Mahoney [1] has taught a course about randomized linear algebra at UCB and shared his class notes, which contain a great amount of theoretical proofs. Meanwhile, Shusen Wang [2] has provided a practical guide to randomized matrix computation with MATLAB implementation. I also referred materials from Martinsson [3], Dong [4].

3 Algorithms

The least squares regression (LSR)

$$\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2^2, \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{n\times d}$, is ubiquitous in statistics, computer science, electrical engineering and many other fields. When the problem is high-dimensional and $n \gg d$, LSR can be solve efficiently using randomized matrix sketching techniques.

There are two types of methods to solve this problem, direct methods and iterative methods. We will pay more attention to the former. A direct thought is to construct another matrix similar to A but significantly smaller than A, called sketching matrix. A randomized sketch can be generated with a random sampling or random projection procedure.

We will first introduce the concept of matrix sketching and several sketching methods in section 3.1 and then briefly discuss how to apply randomized matrix sketching techniques to large-scale, over-constrained LSR problems. We will also prove that the optimal solution to the subproblem is a very good approximation of the solution to the original problem and it's both relatively easy to construct the subproblem with a sketch and solving the subproblem in section.

3.1 Matrix Sketching

Given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, matrix $\mathbf{C} = \mathbf{AS} \in \mathbb{R}^{m\times s}$ is call a sketch of $\mathbf{A}$, where $\mathbf{S} \in \mathbb{R}^{n\times s}$ is a sketching random projection or column selection matrix. The sketch matrix $\mathbf{C}$ has much smaller dimension than the original matrix $\mathbf{A}$ while still preserving some important properties of $\mathbf{A}$.

The sketching matrix is useful for LSR problem if it has the following property:

Property 1 *(Subspace Embedding). For a fixed $m \times n (m \ll n)$ matrix $\mathbf{A}$ and all m-dimension vector $\mathbf{y}$, the inequality*

$$1 - \epsilon \leq \frac{\left\|\mathbf{y}^T\mathbf{AS}\right\|_2^2}{\left\|\mathbf{y}^T\mathbf{A}\right\|_2^2} \leq 1 + \epsilon, \tag{2}$$

where $\mathbf{S} \in \mathbb{R}^{n\times s} (s \ll n)$ is a sketching matrix, holds with high probability.

This property can be interpreted as the length of any n-dimensional vector $\mathbf{x}$ in the row space of $\mathbf{A}$ foes not change much after sketching, i.e. $\|\mathbf{x}\|_2^2 \approx \|\mathbf{xS}\|_2^2$. With a proper sketching methods and a relatively large s, γ should be near one.

In the following sections, we will discuss two types of matrix sketching techniques: random projection and column selection.

3.1.1 Random Projection

Gaussian Projection The $n \times s$ Gaussian random projection matrix $\mathbf{S}$ is a matrix formed by

$$\mathbf{S} = \frac{1}{\sqrt{s}}\mathbf{G}, \tag{3}$$

where each entry of $\mathbf{G}$ is sampled i.i.d. from $\mathcal{N}(0, 1)$.

The time complexity of Gaussian projection is $O(mns)$. When $s = O(m/\epsilon^2)$, the subspace embedding property with $\gamma = 1 + \epsilon$ holds with high probability. Gaussian projection is easy to implement and the sketch $\mathbf{C}$ is of very high quality. However, the time complexity to perform matrix multiplication is high and the possible sparsity of original matrix $\mathbf{A}$ is destroyed.

Subsampled Randomized Hadamard Transform (SRHT) The subsampled randomized Hadamard transform (SRHT) matrix is defined by

$$\mathbf{S} = \frac{1}{\sqrt{sn}}\mathbf{DH}_n\mathbf{P}, \tag{4}$$

where $\mathbf{D} \in \mathbb{R}^{n\times n}$ is a diagonal matrix with entries uniformly sampled from $\{1, -1\}$, $\mathbf{H}_n \in \mathbb{R}^{n\times n}$ is defined recursively by

$$\mathbf{H}_n = \begin{bmatrix} \mathbf{H}_{n/2} & \mathbf{H}_{n/2} \\ \mathbf{H}_{n/2} & -\mathbf{H}_{n/2} \end{bmatrix}, \qquad \mathbf{H}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},$$

$\mathbf{P} \in \mathbb{R}^{n\times s}$ is a sampling matrix that samples s from n columns.

The matrix multiplication $\mathbf{AS}$ can be performed in $O(mn \log s)$, which makes SRHT more efficient than Gaussian projection. When $s = O(\epsilon^{-2}(m+\log n)\log m)$, SRHT satisfies the subspace embedding property with $\gamma = 1 + \epsilon$ holds with probability 0.99.

Count Sketch The streaming fashion of count sketch is summarized in the Algorithm 3.1.1

Algorithm 1 Count Sketch in the Streaming Fashion

Input: $\mathbf{A} \in \mathbb{R}^{m\times n}$.

Initialize $\mathbf{S}$ to an $m \times s$ all-zero matrix;
for $i = 1$ to n **do**
    Sample l from the set $[s]$ uniformly;
    Sample g from the set $\{1, -1\}$ uniformly;
    Update the l-th column of $\mathbf{S}$ by $\mathbf{s}_{:l} \leftarrow \mathbf{s}_{:l} + g\mathbf{a}_{:i}$;
return $\mathbf{S} \in \mathbb{R}^{m\times s}$.

Note that in the previous procedure, the sketch is calculated directly and the sketching matrix $\mathbf{S}$ is formed implicitly. In fact, sketching matrix $\mathbf{S}$ is a matrix that each row has only one nonzero entry.

The time complexity of count sketch is $O(nnz(\mathbf{A}))$, where $nnz(\mathbf{A})$ is the number of nonzero entries of $\mathbf{A}$. When $s = O(m^2/\epsilon^2)$, the subspace embedding property holds with $\gamma = 1 + \epsilon$ with high probability. The count sketch is very efficient, especially when the original matrix $\mathbf{A}$ is sparse. However, the count sketch requires larger s to attain the same accuracy as with Gaussian projection. Therefore, it would be better to combine the count sketch with Gaussian projection or SRHT.

3.1.2 Column Selection

Uniform Sampling The most efficient way to form a sketch is to uniformly sample columns without seeing the whole data matrix. The performance of uniform sampling is data-dependent, reasonable performance can be attained by uniform sampling when the matrix coherence is small.

Leverage Score Sampling Let $\mathbf{A}$ be an $m \times n$ matrix as before, with rank $\rho < n$ and let $\mathbf{V} \in \mathbb{R}^{n\times\rho}$ denote the right singular vectors of $\mathbf{A}$. The (column) leverage scores of $\mathbf{A}$ can be defined as

$$l_i := \|\mathbf{v}_{i:}\|_2^2, \forall i = 1, \ldots, n. \tag{5}$$

Leverage score sampling is simply to select each columns of $\mathbf{A}$ with probability proportional to its leverage scores. Sometimes each selected column should be scaled by $\sqrt{\rho/sl_i}$. When $s = O(m\epsilon + m\log m)$, the leverage score sampling satisfies the subspace embedding property with $\gamma = 1 + \epsilon$ holds with high probability. However, the computation of the leverage scores is as expensive as computing SVD which is expensive. Therefore, it is not common to use leverage score sampling to sketch the matrix $\mathbf{A}$ in practice. When the leverage scores are near uniform, there is little difference between uniform sampling and leverage score sampling.

3.2 Inexact Solution to LSR

For large-scale high-dimensional LSR problems, any sketching matrix $\mathbf{S} \in \mathbb{R}^{n\times s}$ can be used to solve LSR approximately as long as it satisfies the subspace embedding property. The following LSR problem

$$\widetilde{\mathbf{x}} = \min_{\mathbf{x}} \left\|(\mathbf{S}^T\mathbf{A})\mathbf{x} - \mathbf{S}^T\mathbf{b}\right\|_2^2 \tag{6}$$

where $\mathbf{A} \in \mathbb{R}^{n\times d}$ can be solved in $O(sd^2)$. It is easy to show that $\widetilde{\mathbf{x}}$ is a good solution by subspace embedding property.

Let $\mathbf{D} = [\mathbf{A}, \mathbf{b}] \in \mathbb{R}^{n\times(d+1)}$ and $\mathbf{z} = [\mathbf{x}; -\mathbf{1}] \in \mathbb{R}^{d+1}$, then

$$\mathbf{Dz} = \mathbf{Ax} - \mathbf{b} \text{ and } \Rightarrow \mathbf{S^T Dz} = \mathbf{S^T Ax} - \mathbf{S^T}b$$

According to Property 1 (subspace embedding property), we have

$$1 - \epsilon \le \frac{\left\|\mathbf{z}^T\mathbf{D^T S}\right\|_2^2}{\|\mathbf{z}^T\mathbf{D}^T\|_2^2} \le 1 + \epsilon, \forall \mathbf{z},$$

i.e. for each $\mathbf{z}$,

$$(1 - \epsilon)\|\mathbf{Dz}\|_2^2 \le \left\|\mathbf{S}^T\mathbf{Dz}\right\|_2^2 \le (1 + \epsilon)\|\mathbf{Dz}\|_2^2$$

Let $\mathbf{z} = [\widetilde{\mathbf{x}}; -\mathbf{1}]$, and $[\mathbf{x}^*, -\mathbf{1}]$ separately, we can get the following inequalities

$$(1-\epsilon)\|\mathbf{A}\widetilde{\mathbf{x}} - \mathbf{b}\|_2^2 \leq \left\|\mathbf{S^T A}\widetilde{\mathbf{x}} - \mathbf{S^T b}\right\|_2^2,$$

$$\left\|\mathbf{S^T A}\mathbf{x}^* - \mathbf{S^T b}\right\|_2^2 \leq (1+\epsilon)\left\|\mathbf{A}\mathbf{x}^* - \mathbf{b}\right\|_2^2,$$

where

$$\widetilde{\mathbf{x}} = \min_{\mathbf{x}} \left\|(\mathbf{S}^T\mathbf{A})\mathbf{x} - \mathbf{S}^T\mathbf{b}\right\|_2^2 \text{ and } \mathbf{x}^* = \min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2.$$

Due to the optimality of $\widetilde{\mathbf{x}}$, we have

$$\left\|\mathbf{S^T A}\widetilde{\mathbf{x}} - \mathbf{S^T b}\right\|_2^2 \leq \|\mathbf{A}\mathbf{x}^* - \mathbf{b}\|_2^2.$$

Now, we can say that

$$(1-\epsilon)\|\mathbf{A}\widetilde{\mathbf{x}} - \mathbf{b}\|_2^2 \leq \gamma\left\|\mathbf{A}\mathbf{x}^* - \mathbf{b}\right\|_2^2, \textit{ i.e., } \|\mathbf{A}\widetilde{\mathbf{x}} - \mathbf{b}\|_2^2 \leq (1+\epsilon)^2\left\|\mathbf{A}\mathbf{x}^* - \mathbf{b}\right\|_2^2.$$

With a proper sketching method and a relatively large s, γ should be near one, which indicates $\widetilde{\mathbf{x}}$ gives a good solution.

In fact, if $\mathbf{S}$ is a Gaussian projection matrix, SRHT matrix, count sketch or leverage score sampling matrix, and $s = poly(d/\epsilon)$ for any error parameter $\epsilon \in (0, 1]$, then

$$\|\mathbf{A}\widetilde{\mathbf{x}} - \mathbf{b}\|_2^2 \leq (1+\epsilon)^2 \min_{\mathbf{x}} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2$$

is guaranteed.

4 Experiments

4.1 Dataset

We will use those above algorithms to predict the composed year of a song. The dataset we will use is the subset of Million Song Dataset [5], which contains a million contemporary popular music tracks. The entire dataset is 280 GB, which is too large to be processed in a personal computer. Thus, we will use a subset of the original dataset.

The dataset we use consists of $515,345$ records of songs that were composed during the years from 1922 to 2011. Each record consists of 91 features. The first feature is the year in which the song was composed, and the remaining 90 features are various quantities (float) related to the song audio. We want to solve the prediction problem by least squares, thus it's a very over-constrained least-squares problem, in which $A \in \mathbb{R}^{515345\times 90}$. We will compare the precision and the used time of traditional algorithms and randomized algorithms in the experiments.

4.2 Matrix Multiplication

Firstly, we evaluate how much information are preserved in the sketching matrix of difference sketching size. The analytical solution of least squares problem is $(A^TA)^{-1}A^Tb$. Although we don't use this method directly, due to the time complexity of large-scale matrix inversion, there are still a great number of matrix multiplication. Thus, we represent the approximation rate by the relative error of matrix multiplication, specifically A^TA and A^Tb.

$$\textit{relative error of } A^TA = \frac{\|(A^TS)(S^TA)\|_2^2}{\|A^TA\|_2^2}$$

$$\textit{relative error of } A^Tb = \frac{\|(b^TS)(S^Tb)\|_2^2}{\|A^Tb\|_2^2}$$

The relative error of multiplication with difference sketching matrix sizes are presented in figure 1 and 2, showing that the relative errors of matrix multiplication decrease significantly as the sizes of sketching matrix increase. When the size of sketching matrix reaches 1024, the average relative error of difference sketching algorithm is about 0.05, which is acceptable, with the dimension of original matrix reduction from $515,345$ to $1,024$.

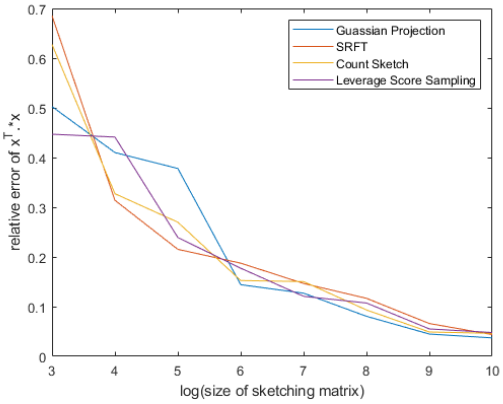

Figure 1: Relative errors of A^TA with different sketching sizes

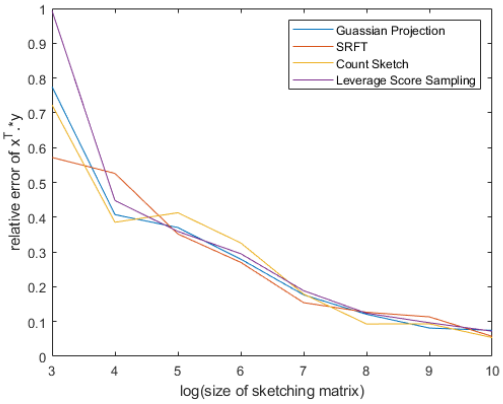

Figure 2: Relative errors of A^Tb with different sketching sizes

4.3 Least Squares Problem

4.3.1 Experiments

Obviously, the problem of predict the composed year of a song with least squares regression is very over-constrained. Thus, we apply the sketching algorithms to this problem. We solve the original LSR problem and get the optimal solution x^* The optimal loss l^*. We also solve the approximated LSR problem with sketching matrix of difference size of sketching matrix. We repeat these randomized linear algebra algorithms for 100 times, and compare the minimal losses and average losses with l^*.

$$l_{min} = \frac{\min l^{(i)}}{l^*}$$

$$l_{avg} = \frac{\sum_i^n l^{(i)}}{nl^*}$$

4.3.2 Results

The l_{min} and l_{avg} of difference sizes of sketching matrix is represented in Figure 3 and 4, showing that l_{min} and l_{avg} decrease significantly as the sizes of sketching matrix increase. When the size of sketching matrix reaches 1024, the l_{min} is about 1.03 and l_{avg} is about 1.05, which are acceptable, with the dimension of original matrix reduction from $515,345$ to $1,024$.

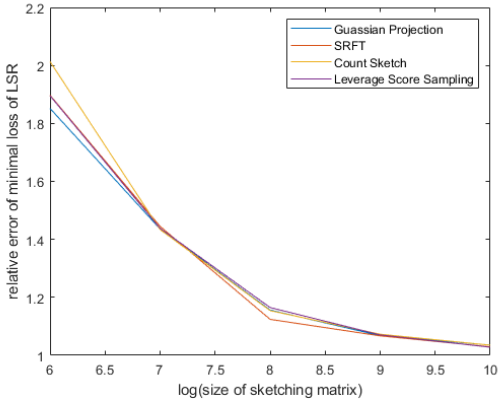

Figure 3: Relative errors of minimal loss of LSR with different sketching size

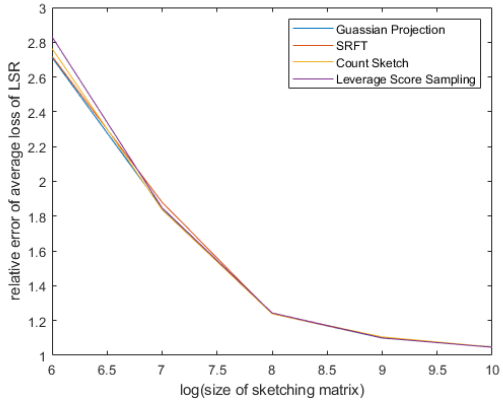

Figure 4: Relative errors of average loss of LSR with different sketching size

4.3.3 Selection of Sketching Size

Naturally, we come to the question that which size of sketching matrix we should choose in practice. If the size of sketching matrix is large, it consumes more time to get solutions of LSR, however, the probability of getting a well-approximated sketching matrix is higher. If the size of sketching matrix is small, the probability of getting a well-approximated sketching matrix is lower, however, it can solve the LSR problem faster, resulting in solving more LSR problems to get a better minimal loss. We compare the minimal losses of different size of sketching matrix from a time perspective. The results are shown in Figure 5. The running time is limited in 5 minutes and the rank of l_{min} is

$$l_{min_{1024}} < l_{min_{512}} < l_{min_{256}} \approx l_{min_8} < l_{min_{16}} < l_{min_{32}} < l_{min_{64}}$$

It shows that when sketching size is large enough, it reaches a small loss for its high-precision approximation of the original matrix, although it only executes several episodes. On the contrary, when sketching size is small enough, it reaches a small loss with a great number of episodes for its effectiveness, although the variance of losses are large in each episode.

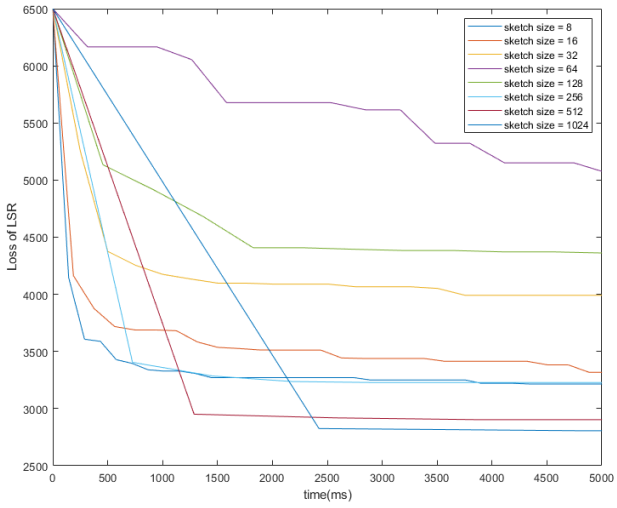

Figure 5: Minimal loss of different sketching size in 5 minutes

4.3.4 An Improved Leverage Score Sampling

In this part, we propose an improved leverage score sampling algorithm which is the fusion algorithm of leverage score sampling algorithm and any other matrix sketching algorithm. The most time-consuming process is getting the right singular vectors of $\mathbf{A}$ by SVD, for $\mathbf{A}$ is a large-scale matrix. To accelerate this process, one effective way is to calculate a sketching matrix of $\mathbf{A}$ and use the right singular vectors of $\mathbf{AS}$ as a substitute. In experiments, we use Gaussian Projection to get an approximated right singular vectors. The experiment results are shown in Figure 6 and 7.

The relative errors of multiplication between Leverage Score Sampling and Approximated Leverage Score Sampling are similar, while there are significant difference of consuming time between Leverage Score Sampling and Approximated Leverage Score Sampling. The consuming time of Leverage Score Sampling is close to a constant, since the SVD of original matrix $\mathbf{A}$ is necessary, regardless of the size of sketching matrix. However, the consuming time of Approximated Leverage Score Sampling is proportional to the sketching size, because the dimension of Gaussian projection matrix depends on the sketching size. Thus, if a small sketching size is chosen in practice, the consuming time of using Approximated Leverage Score Sampling is several times faster than using Leverage Score Sampling.

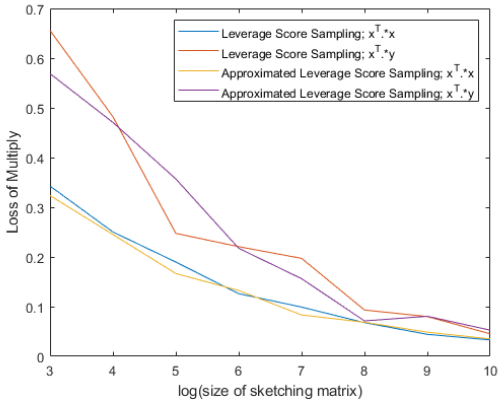

Figure 6: Relative errors of multiplication with different sketching size

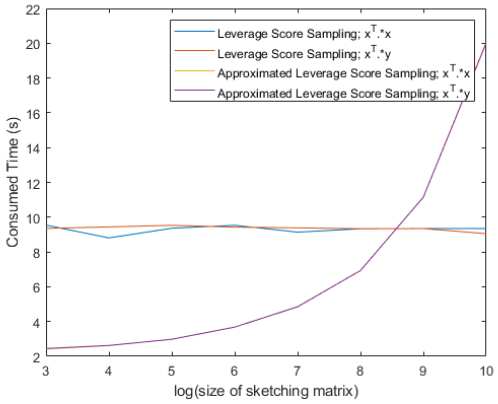

Figure 7: Consuming time of multiplication with different sketching size

5 Conclusion

In this paper, we implement four matrix sketching algorithms, Gaussian Projection, Subsampled Randomized Hadamard Transform(SRHT), Count Sketch and Leverage Score Sampling, which generates sketching matrixes that significantly smaller than the original matrix but approximate it well in a high probability. Moreover, we propose an improved Leverage Score Sampling algorithm via combining Leverage Score Sampling with Gaussian Projection. In experiments, we apply those algorithms to a Least-Squares Problem which is composed year prediction of songs and evaluate the effects of different sketching size.

It is worth mentioning that there are far more than four sketching algorithms mentioned in this paper and there are more techniques for over-constrained Least-squares Problem in the area of Randomized Linear Algebra. We only gain a superficial understanding of this topic and need to learn more in the future.

References

[1] M. W. Mahoney, "Lecture notes on randomized linear algebra," *CoRR*, vol. abs/1608.04481, 2016. [Online]. Available: http://arxiv.org/abs/1608.04481

[2] S. Wang, "A practical guide to randomized matrix computations with MATLAB implementations," *CoRR*, vol. abs/1505.07570, 2015. [Online]. Available: http://arxiv.org/abs/1505.07570

[3] P.-G. Martinsson and J. Tropp, "Randomized numerical linear algebra: Foundations & algorithms," *arXiv preprint arXiv:2002.01387*, 2020.

[4] K. Dong, "Randomized numerical linear algebra for large-scale matrix data," Ph.D. dissertation, Cornell University, 2019.

[5] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[6] E. Liberty, "Simple and deterministic matrix sketching," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2013, pp. 581–588.

[7] R. Witten and E. Candes, "Randomized algorithms for low-rank matrix factorizations: sharp performance bounds," *Algorithmica*, vol. 72, no. 1, pp. 264–281, 2015.

A (Part of) Python codes of matrix sketching

```
def gaussian_proj(s_int, a_mat = None, a_mat_list = None):
    if a_mat is not None and a_mat_list is None:
        m_int, n_int = a_mat.shape
        s_mat = np.random.randn(n_int, s_int) / np.sqrt(s_int)
        sketch_a_mat = np.dot(a_mat, s_mat)
        return sketch_a_mat
    elif a_mat_list is not None and a_mat is None:
        m_int, n_int = a_mat_list[0].shape
        s_mat = np.random.randn(n_int, s_int) / np.sqrt(s_int)
        sketch_a_mat_list = []
        for a_mat in a_mat_list:
            sketch_a_mat = np.dot(a_mat, s_mat)
            sketch_a_mat_list.append(sketch_a_mat)
        return sketch_a_mat_list

def realfft_row(a_mat):
    n_int = a_mat.shape[1]
    fft_mat = np.fft.fft(a_mat, n=None, axis=1) / np.sqrt(n_int)
    if n_int % 2 == 1:
        cutoff_int = int((n_int+1) / 2)
        idx_real_vec = list(range(1, cutoff_int))
        idx_imag_vec = list(range(cutoff_int, n_int))
    else:
        cutoff_int = int(n_int/2)
        idx_real_vec = list(range(1, cutoff_int))
        idx_imag_vec = list(range(cutoff_int+1, n_int))
    c_mat = fft_mat.real
    c_mat[:, idx_real_vec] *= np.sqrt(2)
    c_mat[:, idx_imag_vec] = fft_mat[:, idx_imag_vec].imag * np.sqrt(2)
    return c_mat

def srft(s_int, a_mat = None, a_mat_list = None):
    if a_mat is not None and a_mat_list is None:
        n_int = a_mat.shape[1]
        sign_vec = np.random.choice(2, n_int) * 2 - 1
        idx_vec = np.random.choice(n_int, s_int, replace=False)
        a_mat = a_mat * sign_vec.reshape(1, n_int)
        a_mat = realfft_row(a_mat)
        c_mat = a_mat[:, idx_vec] * np.sqrt(n_int / s_int)
        return c_mat
    elif a_mat_list is not None and a_mat is None:
        n_int = a_mat_list[0].shape[1]
        sign_vec = np.random.choice(2, n_int) * 2 - 1
        idx_vec = np.random.choice(n_int, s_int, replace=False)
        sketch_a_mat_list = []
        for a_mat in a_mat_list:
            a_mat = a_mat * sign_vec.reshape(1, n_int)
            a_mat = realfft_row(a_mat)
            c_mat = a_mat[:, idx_vec] * np.sqrt(n_int / s_int)
            sketch_a_mat_list.append(c_mat)
        return sketch_a_mat_list

def countsketch(s_int, a_mat = None, a_mat_list = None):
    if a_mat is not None and a_mat_list is None:
        m_int, n_int = a_mat.shape
        hash_vec = np.random.choice(s_int, n_int, replace=True)
```

```
        sign_vec = np.random.choice(2, n_int, replace=True) * 2 - 1
        sketch_a_mat = np.zeros((m_int, s_int))
        for j in range(n_int):
            h = hash_vec[j]
            g = sign_vec[j]
            sketch_a_mat[:, h] += g * a_mat[:, j]
        return sketch_a_mat
    elif a_mat_list is not None and a_mat is None:
        m_int, n_int = a_mat_list[0].shape
        hash_vec = np.random.choice(s_int, n_int, replace=True)
        sign_vec = np.random.choice(2, n_int, replace=True) * 2 - 1
        sketch_a_mat_list = []
        for a_mat in a_mat_list:
            sketch_a_mat = np.zeros((a_mat.shape[0], s_int))
            for j in range(n_int):
                h = hash_vec[j]
                g = sign_vec[j]
                sketch_a_mat[:, h] += g * a_mat[:, j]
            sketch_a_mat_list.append(sketch_a_mat)
        return sketch_a_mat_list

def leverage( s_int, a_mat = None, a_mat_list = None):
    if a_mat is not None and a_mat_list is None:
        # calculate leverage vector
        n_int = a_mat.shape[1]
        _ , _, v_mat = np.linalg.svd(a_mat, full_matrices=False)
        lev_vec = np.sum(v_mat ** 2, axis=0)
        # generate sketch matrix by leverage
        a_mat = a_mat.T
        prob_vec = lev_vec / sum(lev_vec)
        idx_vec = np.random.choice(n_int, s_int, replace=False, p=prob_vec)
        scaling_vec = np.sqrt(s_int * prob_vec[idx_vec]) + 1e-10
        sx_mat = a_mat[idx_vec, :] / scaling_vec.reshape(len(scaling_vec), 1)
        return sx_mat.T
    if a_mat_list is not None and a_mat is None:
        # calculate leverage vector
        a_mat = a_mat_list[0]
        n_int = a_mat.shape[1]
        _ , _, v_mat = np.linalg.svd(a_mat, full_matrices=False)
        lev_vec = np.sum(v_mat ** 2, axis=0)
        # generate sketch matrix by leverage
        a_mat = a_mat_list[0].T
        b_mat = a_mat_list[1].T
        prob_vec = lev_vec / sum(lev_vec)
        idx_vec = np.random.choice(n_int, s_int, replace=False, p=prob_vec)
        scaling_vec = np.sqrt(s_int * prob_vec[idx_vec]) + 1e-10
        sx_mat = a_mat[idx_vec, :] / scaling_vec.reshape(len(scaling_vec), 1)
        sy_mat = b_mat[idx_vec, :] / scaling_vec.reshape(len(scaling_vec), 1)
        return [sx_mat.T, sy_mat.T]

def leverage_approx(s_int, a_mat = None, a_mat_list = None):
    if a_mat is not None and a_mat_list is None:
        # calculate leverage vector
        n_int = a_mat.shape[1]
        b_mat = gaussian_proj(s_int, a_mat=a_mat)
        u_mat, sig_vec, _ = np.linalg.svd(b_mat, full_matrices=False)
        t_mat = u_mat.T / sig_vec.reshape(len(sig_vec), 1)
        b_mat = np.dot(t_mat, a_mat)
        lev_vec = np.sum(b_mat ** 2, axis=0)
```

```
        # generate sketch matrix by leverage
        a_mat = a_mat.T
        prob_vec = lev_vec / sum(lev_vec)
        idx_vec = np.random.choice(n_int, s_int, replace=False, p=prob_vec)
        scaling_vec = np.sqrt(s_int * prob_vec[idx_vec]) + 1e-10
        sx_mat = a_mat[idx_vec, :] / scaling_vec.reshape(len(scaling_vec), 1)
        return sx_mat.T
    if a_mat_list is not None and a_mat is None:
        # calculate leverage vector
        a_mat = a_mat_list[0]
        m_int, n_int = a_mat.shape
        b_mat = gaussian_proj(s_int, a_mat=a_mat)
        u_mat, sig_vec, _ = np.linalg.svd(b_mat, full_matrices=False)
        t_mat = u_mat.T / sig_vec.reshape(len(sig_vec), 1)
        b_mat = np.dot(t_mat, a_mat)
        lev_vec = np.sum(b_mat ** 2, axis=0)
        # generate sketch matrix by leverage
        a_mat = a_mat_list[0].T
        b_mat = a_mat_list[1].T
        prob_vec = lev_vec / sum(lev_vec)
        idx_vec = np.random.choice(n_int, s_int, replace=False, p=prob_vec)
        scaling_vec = np.sqrt(s_int * prob_vec[idx_vec]) + 1e-10
        sx_mat = a_mat[idx_vec, :] / scaling_vec.reshape(len(scaling_vec), 1)
        sy_mat = b_mat[idx_vec, :] / scaling_vec.reshape(len(scaling_vec), 1)
        return [sx_mat.T, sy_mat.T]
```

B (Part of) Python codes of algorithm evaluation

```
def evaluate_multiply(s_int, fun_proj):
    # evaluate the error and computation time of x*x.T
    repeat = 10
    err_xx = 0
    time_xx = 0
    for i in range(repeat):
        time_start=time.time()
        c_mat = fun_proj(s_int, a_mat=x_mat)
        time_end=time.time()
        err_xx += np.linalg.norm(xx_mat - np.dot(c_mat, c_mat.T), ord='fro') /
            xx_norm
        time_xx += time_end - time_start
    err_xx /= repeat
    time_xx /= repeat
    print('Approximation error xx for s=' + str(s_int) + ': ' + str(err_xx))
    print('Computation time xx for s='+ str(s_int) + ': ' + str(time_xx))
    # evaluate the error and computation time of x*y.T
    err_xy = 0
    time_xy = 0
    for i in range(repeat):
        time_start=time.time()
        c_mat_list = fun_proj(s_int, a_mat_list=[x_mat, y_mat])
        time_end=time.time()
        c_mat = c_mat_list[0]
        d_mat = c_mat_list[1]
        err_xy += np.linalg.norm(xy_mat - np.dot(c_mat, d_mat.T), ord='fro') /
            xy_norm
        time_xy += time_end - time_start
```

```
    err_xy /= repeat
    time_xy /= repeat
    print('Approximation error xy for s=' + str(s_int) + ': ' + str(err_xy))
    print('Computation time xy for s='+ str(s_int) + ': ' + str(time_xy))
    return err_xx, err_xy,time_xx,time_xy

def evaluate_multiply_with_different_sketch_size(fun_proj, fun_label):
    # evaluate the error of multiply with different sketch size
    err_multiply = [[] for _ in range(2)]
    time_multiply = [[] for _ in range(2)]
    sketch_size_list = [8,16,32,64,128,256,516,1024]
    for i in sketch_size_list:
        err_xx, err_xy,time_xx,time_xy = evaluate_multiply(i, fun_proj)
        err_multiply[0].append(err_xx)
        err_multiply[1].append(err_xy)
        time_multiply[0].append(time_xx)
        time_multiply[1].append(time_xy)
    plt.plot(sketch_size_list, np.log(err_multiply[0]), label = 'x*x.T')
    plt.plot(sketch_size_list, np.log(err_multiply[1]), label = 'x*y.T')
    plt.xlabel('sketch size of '+fun_label)
    plt.ylabel('log(error)')
    plt.legend()
    plt.show()
    return err_multiply, time_multiply

def evaluate_lsr(s_int, fun_proj):
    episodes_num = 2
    min_loss = np.Inf
    err_arr = []
    for i in range(episodes_num):
        [x_sketch, y_sketch] = fun_proj(s_int, a_mat_list=[x_mat, y_mat])
        weight_sketch = np.linalg.lstsq(x_sketch.T, y_sketch.T, rcond=None)[0]
        loss_sketch = np.linalg.norm(np.dot(x_mat.T, weight_sketch) - y_mat.T)
        if loss_sketch < min_loss:
            min_loss = loss_sketch
        err_arr.append(loss_sketch)
        print(s_int, i, loss_sketch, min_loss)
    return min_loss, np.mean(err_arr)

def evaluate_lsr_with_different_sketch_size(fun_proj, fun_label):
# evaluate the error of least squares regression with different sketch size
    print("evaluate the error of least square with ", fun_label)
    weight_lsr = np.linalg.lstsq(x_mat.T, y_mat.T, rcond=None)[0]
    loss_lsr = np.linalg.norm(np.dot(x_mat.T, weight_lsr) - y_mat.T)
    print("loss of origin problem is ", loss_lsr)
    sketch_size_list = [8,16,32,64,128,256,516,1024,2048]
    min_err_lsr = []
    avg_err_lsr = []
    time_lsr = []
    for i in sketch_size_list:
        time_start=time.time()
        results = evaluate_lsr(i, fun_proj)
        time_end=time.time()
        min_err_lsr.append(results[0])
        avg_err_lsr.append(results[1])
        time_lsr.append(time_end - time_start)
    print('The min errors are : ',min_err_lsr)
    print('The average errors are : ', avg_err_lsr)
    print('The computation times are : ', time_lsr)
```

```
plt.plot(sketch_size_list, err_lsr, label = fun_label)
plt.plot(sketch_size_list, [loss_lsr]*8 , label = 'Optimal')
plt.xlabel('sketch size')
plt.ylabel('Error of least square regression')
plt.legend()
plt.show()
```