Potential of the Randomized numerical methods

Junsoo Jung

Department of Applied Mathematics, San Francisco State University, San Francisco, CA 94132

Potential of the Randomized numerical methods

An Honors Thesis
Presented to the Department of Applied Mathematics
San Francisco State University
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by
Junsoo Jung
San Francisco, California
Sep 25, 2022

Contents

Acknowledgments

I would like to begin by thanking my thesis and major advisor, Professor Henry Boatang. He helped almost all of my plan and leads me where to head.
Also, he was very generous to understand every situation I faced.

My sister, who support me emotionally, and educationally.

Introduction

The accurate and efficient decomposition of large data matrices is one of the cornerstones of modern computational mathematics and data science. Decomposition of large data is an important step in computational science. In many cases, matrix decompositions are explicitly focused on extracting dominant low-rank structure in the matrix. These so-called randomized numerical methods have the potential to transform computational linear algebra, providing accurate matrix decompositions at a fraction of the cost of deterministic methods. Moreover, with increasingly vast measures (e.g., from 4K and 8K video, internet of things, etc.), it is often the case that the intrinsic rank of the data does not increase appreciable, even though the dimension of the ambient measurement space grows Configuration and recognizing those files takes getting slower and slower because of the size of the files. But by decomposition the large data file to smaller file, the time require to recognize the file will became faster.

In this expository project, I will compare randomized numerical method with normal process run-time to see how much time this decomposition could save and how accurate decomposition the data will be. The main insight is that randomness is an algorithmic resource creating efficient, unbiased approximations of nonrandom operations.

The textbook "Data Driven Science Engineering" by L. Brunton and Kutz provides a good introduction to the topic. I will be exploring a randomized approach to data decomposition.

CHAPTER 1

Randomized SVD

Low-rank matrix decompositions are fundamental tools and widely used for data analysis, dimension reduction, and data compression. Classically, highly accurate deterministic matrix algorithms are used for this task. However, the emergence of large-scale data has severely challenged our computational ability to analyze big data.

SVD is the most well-known method of decomposition. In many domains, complex systems will generate data that is naturally arranged in large matrices, or more generally in arrays. For example, a time-series of data from an experiment or a simulation may be arranged in a matrix with each column containing all of the measurements at a given time. If the data at each instant in time is multi-dimensional, as in a high-resolution simulation of the weather in three spatial dimensions, it is possible to reshape or flatten this data into a high-dimensional column vector, forming the columns of a large matrix. Similarly, the pixel values in a grayscale image may be stored in a matrix, or these images may be reshaped into large column vectors in a matrix to represent the frames of a movie. Remarkably, the data generated by these systems are typically low rank, meaning that there are a few dominant patterns that explain the high-dimensional data. The SVD is a numerically robust and efficient method of extracting these patterns from data.

By installing 'RSVD' package into the R studio, we can easily access to the randomized SVD example.

CHAPTER 2

Randomized SVD Using R

Thanks for the Rstudio, there is package already coded Randomized SVD method.

Start with the recalling the library.

```
library(rsvd)
library(microbenchmark)
```

Rsvd library is for use built in function randomized SVD, and Microbenchmark is for calculate the processing time.

Use built in function rsvd and reconstruct the matrix.

```
set.seed(0)
A= Matrix(40,20)
rank = rankMatrix(A)
s <- rsvd(A, k=150)
reconA = s$u %*% diag(s$d) %*% t(s$v)
```

Compare with SVD and RSVD processing time. This will measure the each calculation 10 times and will give us a mean value.

```
timing_svd <- microbenchmark(
  'SVD' = svd(A, nu=150, nv=150),
  'rSVD' = rsvd(reconA, k=150),
  times=10)
print(timing_svd, unit='s')
```

CHAPTER 3

Johnson-Lindenstrauss Lemma

Johnson-LindenStrauss is one of the decompositioning method.

LEMMA 3.1. *For Any $0 < \varepsilon < 1$ and any integer n let k be a positive interger such that*

$$k \geq \frac{24}{3\varepsilon^2 - 2\varepsilon^3} \log n$$

then for any set A of n points $\in R^d$ there exists a map $f : R^d \to R^k$ such that for all $x_i, x_j \in A$

$$(1-\varepsilon)||x_i - x_j||^2 \leq ||f(x_i) - f(x_j)||^2 \leq (1+\varepsilon)||x_i - x_j||^2$$

Further this map can be found in randomized ploynomial time. Repeating this projection $O(n)$ times can boost the success probability to any desired constant, giving us a randomized polynomial time algorithm.

DEFINITION. Let R be a random matrix of order $k \times d$ i.e $R_{ij} \overset{iid}{\sim} N(0,1)$ and u be any fixed vector $\in R^d$. Define $v = \frac{1}{\sqrt{k}} Ru$. Thus $v_i = \frac{1}{\sqrt{k}} \sum_j R_{ij} u_j$

LEMMA 3.2. $E[\|v\|_2^2] = \|u\|_2^2$

PROOF. Start with L.H.S.

$E[\|v\|_2^2] = E[\sum_{i=1}^k v_i^2]$
Factor out the summation out of expected value
$= \sum_{i=1}^k E[v_i^2]$
Because we set $v_i = \frac{1}{\sqrt{k}} \sum_j R_{ij} u_j$ in definition,

$= \sum_{i=1}^{k} \frac{1}{k} E[(\sum_j R_{ij} u_j)^2]$
Introduce a dummy variable k. In k x d matrix ($k \leq d$), $value of the matrix component R_{ij}$ is 1 when i = j, otherwise 0.
Let's say it is σ_{jk} It related j and k because,
$= \sum_{i=1}^{k} \frac{1}{k} \sum_{1 \leq j,k \leq d} u_j u_k \sigma_{jk}$

Because if $j \neq k$, σ_{jk} is 0. So, only terms that j = k are left in summation. So, $u_j u_k = u_j^2$
$= \sum_{i=1}^{k} \frac{1}{k} \sum_{1 \leq j \leq d} u_j^2$

Now, sum of k of same terms are just multiplying k
$= \sum_{1 \leq j \leq d} u_j^2$
$= \|u\|_2^2$

□

Definition. Let $X = \frac{\sqrt{k}}{\|u\|} v$ i.e $x_i = \frac{1}{\|u\|} R_i^T u$ for $i = 1(1)k$
$x = \sum_{i=1}^{k} x_i^2 = \|X\|_2^2 = \frac{k\|v\|_2^2}{\|u\|_2^2}$

Lemma 3.3. *The probability* $P[\|v\|_2^2 \geq (1+\varepsilon)\|u\|_2^2] > 1 - n^{-2}$

Proof. $P[\|v\|_2^2 \geq (1+\varepsilon)\|u\|_2^2]$

Because we set $x = \frac{k\|v\|_2^2}{\|u\|_2^2}$
$= P[\frac{x \cdot \|u\|_2^2}{k} \geq (1+\varepsilon)\|u\|_2^2]$
Cancel same component each side.
$= P[x \geq (1+\varepsilon) \cdot k]$
multiply lambda (lambda can be all positive value) and powered by exponential each side.
$= P[e^{\lambda x} \geq e^{\lambda(1+\varepsilon) \cdot k}]$

And Use Markov's inequality.

Definition. Markov's inequality
$E(X) = \int_{-\infty}^{\infty} x \cdot f(x) dx$
Split the range at some point a.

$= \int_{-\infty}^{a} x \cdot f(x)dx + \int_{a}^{\infty} x \cdot f(x)dx$
In probability, expected value is always positive.
So, by removing $\int_{-\infty}^{a} x \cdot f(x)dx$ in this equation, L.H.S. is always same or bigger than R.H.S.
$\geq \int_{a}^{\infty} x \cdot f(x)dx$
and also, because x is in range of a to infinity, $\int_{a}^{\infty} x \cdot f(x)dx$ is always same or bigger than $\int_{a}^{\infty} a \cdot f(x)dx$
$\geq \int_{a}^{\infty} a \cdot f(x)dx$
factor out a outside of the integral,and because $\int_{a}^{\infty} f(x)dx$ is eqaul to $P[x \geq a]$,
$\geq a \cdot P[x \geq a]$

Divide by a on both side of $E(X) \geq a \cdot P[x \geq a]$
Finally, we got $\frac{E(X)}{a} \geq P[x \geq a]$

Now, let's set $x = e^{\lambda x}, a = e^{\lambda(1+\varepsilon)\cdot k}$
Then, $P[e^{\lambda x} \geq e^{\lambda(1+\varepsilon)\cdot k}] \leq \frac{E[e^{\lambda x}]}{e^{\lambda(1+\varepsilon)\cdot k}}$

$\leq \prod_{i=1}^{\infty} \frac{E[e^{\lambda x_i}]}{e^{\lambda(1+\varepsilon)\cdot k}}$
x_i are i.i.d, So, product of x_i is x_i^k
$\leq (\frac{E[e^{\lambda x_i}]}{e^{\lambda(1+\varepsilon)\cdot k}})^k$

□

CHAPTER 4

Proving via R code(Johnson-Lindenstrauss)

In this chapter, this is R code that showing Johnson-Lindenstrauss Lemma by the graph.

1. Call Library

First, we need to call library what we will use.

```
library(Matrix)
library(microbenchmark)
```

Now generate a random matrix function that will fill matrix w/ random digit.

```
Matrix = function(m,n){
  A = matrix(sample(0, m*n, replace=TRUE), nrow=m, ncol=n)
  rank = rankMatrix(A)

  fillcolumn = function(){
    c = sample(1:4, 1, replace = T)
    if (c==1){
      a = sample(-1000:1, 1, replace = T)
      b = sample(1:1000, 1, replace = T)
      column = runif(m, min = a, max = b-a)
    }else if (c==2){
      a = -10+20*runif(1)
      b = 2*runif(1)
      column = rnorm(m, a, b)
    }else if (c==3){
      b = 2+2*runif(1)
      column = rgamma(m, shape = b, scale = b)
    }else if (c==4){
      a = 0.5+2*runif(1)
      column = rexp(m,a)
    }
```

```
    return(column)
  }
  for (i in 1:n){
    A[,i]=fillcolumn()
  }
  return(A)
}
```

This function will calculate the distance between columns.

```
#distance
distance = function(A){
  n = ncol(A)
  dist = list()
  for (i in 1:n){
    u = A[,i]
    for (j in i:n){
      if(i!=j){
        dist = append(dist, norm((u-A[,j]),type = "2")^2)
      }
    }
  }
  return(dist)
}
```

Make a block of Johnson-Lindenstrauss Lemma equation in chapter 1.

```
jllemma  = function(A, epsilon){
  m = nrow(A)
  n = ncol(A)
  k = ceiling((24*log(n)/(3*epsilon**2-2*epsilon**3)))
  z = matrix(rnorm(k*m),nrow = k,ncol = m)
  dist = list()
  for(i in 1:n){
    u = A[,i]
    for(j in i:n){
      if(i!=j){
        dist = append(dist, 1/k* norm(z%*%(u-A[,j]), type = "2")^2)
      }
    }
  }
```

```
  return(dist)
}
```

This is the calculation of the Johnson-Lindenstrauss equation. The return value will be graphed and show us the result.

```
test = function(A,epsilon,dist,nexpts){
  matdist = distance(A)
  JLsum = vector(mode="list",length = length(matdist))
  for (i in 1:length(matdist)){
    JLsum[i]=0
  }
  sum = Map("+", JLsum, jllemma(A,epsilon))
  upper = 1+epsilon
  lower = 1-epsilon
  dscale = Map("/", sum, (lapply(matdist,"*",nexpts)))

  return(c((sum(dscale <= upper)+sum(dscale >= lower)/length(dist)),d
}
```

Form A which is our target matrix.

```
#variables loading
set.seed(0)
A= Matrix(40,20)
rank = rankMatrix(A)
```

Run JL Lemma with matrix A.

```
epsilon= 0.2
upper = 1+epsilon
lower = 1-epsilon
result = test(A,0.2,distance(A),1)
val = result[1]
lens = length(scale)
plot(unlist(result[-1]),ylim = c(1-2*epsilon, 1+2*epsilon))
abline(h = upper, col ="blue")
abline(h = lower, col ="blue")
```

Set the code for the calculation time. This will measure the each calculation 10 times and will give us a mean value.

```
timing_jl <- microbenchmark(
  'JL Lemma' = test(A,0.2,distance(A),1),
  times=10)
```

```
print(timing_jl)
```

Bibliography

[1] Steven L. Brunton, J.Nathan Kutz, *Machine Learning, Dynamical Systems and Control*, Webpage Link

[2] *RSVD library - R*, CRAN-RSVD Link

[3] Stephen L. Cornford, Daniel F. Martin, Daniel T. Graves, Douglas F. Ranken, Anne M. Le Brocq, Rupert M. Gladstone, Antony J. Payne, Esmond G. Ng, William H. Lipscomb, *Adaptive mesh, finite volume modeling of marine ice sheets.*, PDF Link

[4] Jiequn Han, Arnulf Jentzen, Weinan E *Solving high-dimensional ,partial differential equations using deep learning*, August 6, 2018, Article Link