SVD Computation

Samprit Chakraborty , Samahriti Mukherjee , Arunsoumya Basu , Aytijhya Saha
Indian Statistical Institute, Kolkata

Fall 2021

Contents

1 Introduction

We are interested to write a program to find SVD of a matrix from scratch.

2 Algorithm 1

We used QR Decomposition to find the SVD of a matrix.

Let A is a matrix and we want to find its eigenvalues. Let $A = A_0$, then let

$$A_0 = Q_0 R_0,$$

where $Q_0 R_0$ is the QR Decomposition of A_0.

Now we define $A_1 = R_0 Q_0$ and find the QR Decomposition of A_1, call it $Q_1 R_1$, then we define $A_2 = R_1 Q_1$. We continue like this and finally we will see that for large n, A_n converges to an upper triangular matrix. Now let $P_n = Q_1 Q_2 ... Q_n$ and $P'_n A P_n = Q'_n Q'_{n-1} ... Q'_1 \acute{A} Q_1 Q_2 ... Q_n$

$\implies$ P$'_n A P_n = Q'_n Q'_{n-1} ... Q'_2 \acute{A}_1 Q_2 Q_3 ... Q_n$

$\implies$ P$'_n A P_n = A_n$ Now as A is similar to A_n, so eigenvalues of A will be same as eigenvalues of A_n.

Now as A_n is upper triangular, so diagonal elements of A_n will be the eigenvalues of A. Columns of P_k gives the eigenvectors to the corresponding eigenvalues which are orthogonal to each other.

Proof of A_n converging to an upper triangular matrix:

We see $P_{k+1} = P_k Q_{k+1}$

$\implies$ Q$_{k+1} = P_k' P_{k+1}$

Now if $P_k \to P$,for large k, then Q_{k+1} $\to$I.

$\implies$ A$_{k+1} \to R_k$.

Now if we are given a matrix B for which we want the SVD Decomposition, we find the eigenvalues of BB^T and $B^T B$, which are the squares of the singular values of B and the corresponding orthogonal eigenvectors. The matrices formed by eigenvectors of BB^T and $B^T B$ are U and V, say. We take a matrix C which has singular values of B in diagonals and all other elements are 0. Then UCV^T will be the SVD Decomposition of B.

2.1 R code based on this algorithm:

```
QR=function(A)
{
  m <<- nrow(A)
  n <<- ncol(A)
  count <<- 1
  flag <<- 0
  Q <<- diag(1,n)
  unit = function(v) {v/sqrt(sum(v*v))}                          #
     Function to find the unit vector
  hmult = function(u, x) {x - 2* sum(u*x)*u}                     #
     Function to find the product of a Householder matrix with a vector
  shaver = function(x)                                           #
     Function to shave a vector
  {
    x[1] = x[1] - sqrt(sum(x*x))
    unit(x)
  }
  u_vec=numeric((m*(m+1)/2-(m-n)*(m-n+1)/2))                     #Vector
      to hold the values of the u_i s obtained at different steps
  for (i in 1:n)
  {
    if (i<m)
    {
```

```
      if (sum((A[(i+1):m,i])^2)>0.0000001)                                    #
          Checking if the entries of A are already shaved, in case A is not of
          full column rank
      {
        u_vec[count:(count+(m-i))]=shaver(A[i:m,i])                           #u_vec
            holds the values of the u_i s linearly
        for (j in 1:(n+1-i))
        {
          A[i:m,(i+j-1)]=hmult(u_vec[count:(count+(m-i))],A[i:m,(i+j-1)])   #
              Multiplication with Householder matrix
        }
      }
      else
      {
        u_vec[count:(count+(m-i))]=0
      }
    }
    newmat1=diag(1,m)
    temp=diag(1,(m+1-i))-2*matrix(u_vec[count:(count+(m-i))])%*%t(matrix(u_vec[
        count:(count+(m-i))]))
    newmat1[i:m,i:m]=temp
    Q <<- Q%*%newmat1
    count <<- count+m+1-i
  }
  R=round(A,digits=6)
  R <<- R
  return(list(Q=Q,R=R))
}
eig=function(M){
  X=M
  P=diag(1,dim(M)[1])
  for(i in 1:1000)
  {
    d=QR(X)
    Q=QR(X)$Q
    P=P%*%Q
    X=QR(X)$R%*%Q
  }
  return(list(diag(round(X,5)),round(P,5)))
}
SVDcomp=function(M){
  X=M
  D=t(M)%*%M
```

```
  E=M%*%t(M)
  V=eig(D)[2]
  U=eig(E)[2]
  a=unlist(eig(D)[1])
  f=sqrt(as.numeric(a))
  print(U)
  print(f)
  print(V)
}
SVDcomp(M)
```

2.2 Drawbacks of the method:

- We are blindly running the QR-flip loop (ie, computing QR and then finding RQ) a fixed number of times instead of running it until convergence.
- we are rounding off the result after each iteration.

3 Algorithm 2

First, we convert the given matrix $M_{m\times n}$ into a bidiagonal matrix by a series of multiplications by Householder matrices. Here, we assume that $m \geq n$, otherwise all the operations are performed on $\mathbf{M^T}$. This is done in the following manner: We know that we can shave all but the first coordinate of a vector by multiplying it with a Householder matrix.

So, given distinct vectors $\mathbf{u} \in \mathbb{R}^{\mathbf{n}}$ and $\mathbf{v} \in \mathbb{R}^{\mathbf{n}}$ (with $||\mathbf{u}|| = ||\mathbf{v}||$ and with the last $(n-1)$ coordinates of $\mathbf{v}$ 0) we can multiply $\mathbf{u}$ by a matrix $\mathbf{A}$ where $\mathbf{A} = \mathbf{I_n} - \mathbf{2tt'}$, with $\mathbf{t}$ being the unit vector in the direction of $\mathbf{u} - \mathbf{v}$.

This enables us to make the last $(m-1)$ entries of the first column of M 0 through a premultiplication with a Householder matrix $\mathbf{U_1}$. Next, we make the first $(n-2)$ entries of the first row of this new $\mathbf{M}$ zeroes, by postmultiplying the matrix with $\mathbf{V_1}$. Then the last $(m-2)$ entries of the second column of $\mathbf{M}$ are converted to 0 by another premultiplication with $\mathbf{U_2}$. After this, all but the first three entries of the second row of this new matrix are shaved by a multiplication with $\mathbf{V_2}$. This process of alternately shaving columns and rows continues till we have generated $\mathbf{U_1}, \mathbf{U_2}, \cdots, \mathbf{U_n}$ and $\mathbf{V_1}, \mathbf{V_2}, \cdots, \mathbf{V_{n-2}}$.

We note that at each step, the matrix $\mathbf{M}$ is overwritten after multiplication with the Householder matrix. Also, the Householder matrices are gradually decreasing in size, so they are packed into a $m \times m$ identity matrix (for $\mathbf{U}$) and into a $n \times n$ identity matrix (for $\mathbf{V}$) as a principal submatrix formed by the last few rows and columns. Thus, it makes sense to compute the product

$\mathbf{U^T} = \mathbf{U_n U_{n-1}} \cdots \mathbf{U_1}$ and $\mathbf{V} = \mathbf{V_1 V_2} \cdots \mathbf{V_{n-2}}$.

At the end of this process, **M** has been reduced to a bidiagonal matrix and we have found the orthogonal matrices **U** and **V** which transform **M** into a bidiagonal matrix.

Now we need to compute the Singular Value Decomposition of this bidiagonal **M**, which turns out to be computationally simpler than directly finding the Singular Value Decomposition of the original matrix.

This is done through a series of alternate **QR** and **LQ** decompositions, carried out until we get an approximately diagonal matrix of singular values.

This diagonal matrix is the singular matrix of the original matrix **M**, since orthogonal transformations do not alter the singular values. The **U** and the **V** matrices obtained from the Singular Value Decomposition of the bidiagonal matrix are then multiplied with the existing **U** and **V** matrices to get the final **U** and **V** marices.

3.1 R code based on this algorithm:

```
SVD=function(B)   #A function to compute the SVD of a matrix with nrow>=2, ncol>=2
{
  flag <<- 0

  if(nrow(B)<ncol(B))
  {
    A=matrix(0,(ncol(B)+1),(nrow(B)+1))
    A[1:ncol(B),1:nrow(B)]=t(B)
  }

  else
  {
    A=matrix(0,(nrow(B)+1),(ncol(B)+1))
    A[1:nrow(B),1:ncol(B)]=B
  }
  m <<- nrow(A)-1
  n <<- ncol(A)-1

    unit = function(v) {v/sqrt(sum(v*v))}          #Function to find the unit
        vector

    hmult = function(u, x) {x - 2* sum(u*x)*u}     #Function to find the product
        of a Householder matrix with a vector

    shaver = function(x)                           #Function to shave a vector
    {
```

```
    x[1] = x[1] - sqrt(sum(x*x))
    unit(x)
  }

  for (i in 1:n)
    {
      if (sum((A[(i+1):m,i])^2)>10^(-12))         #Checking if the entries of A
          are already shaved
      {
        if (i !=m)
        {
        temp=shaver(A[i:m,i])
        }
        else
        {
        temp=0
        }
        for (j in i:n)
        {
          A[i:m,j]=hmult(temp,A[i:m,j])           #Multiplication with
              Householder matrix
        }
        A[(i+1):(m+1),i]=temp
      }

      else
      {
        A[(i+1):(m+1),i]=0
      }


      if (i<=(n-2))
      {
        if (sum((A[i,(i+2):n])^2)>10^(-12))
        {
          temp=shaver(A[i,(i+1):n])
            for (k in i:m)
            {
              A[k,(i+1):n]=hmult(temp,A[k,(i+1):n])
            }
            A[i,(i+2):(n+1)]=temp
        }
        else
```

```
        {
          A[i,(i+2):(n+1)]=0
        }
      }
  }

  A <<- A

  #Calculating U_A and V_A, the orthogonal matrices used for bidiagonalization

  U_A <<- diag(1,m)

  for (i in 1:n)
  {
    newmat1=diag(1,m)
    temp=diag(1,(m+1-i))-2*matrix(A[(i+1):(m+1),i])%*%t(matrix(A[(i+1):(m+1),i])
       )
    newmat1[i:m,i:m]=temp
    U_A <<- U_A%*%newmat1
  }

  V_A <<- diag(1,n)

  if (n>=3)
  {
  for (i in 1:(n-2))
  {
    newmat1=diag(1,n)
    temp=diag(1,(n-i))-2*matrix(A[i,(i+2):(n+1)])%*%t(matrix(A[i,(i+2):(n+1)]))
    newmat1[(i+1):n,(i+1):n]=temp
    V_A <<- V_A%*%newmat1
  }
  }

  #print(U_A)
  #print(V_A)
  #print(round(t(U_A)%*%U_A,digits=6))
  #print(round(t(V_A)%*%V_A,digits=6))

  if (nrow(B)>=ncol(B))
  {
    bid <<- t(U_A)%*%B%*%V_A
  }
```

```
else
{
  bid <<- t(U_A)%*%t(B)%*%V_A
}

bid <<- round(bid,digits=6)    #The bidiagonal matrix

U3=diag(1,min(nrow(B),ncol(B)))
V3=diag(1,min(nrow(B),ncol(B)))

U_A <<- U_A[1:m,1:n]

epsilon=max(abs(bid[1:n,1:n]-diag(diag(bid))))

while(epsilon>0.0000001)    #Alternating between QR and LQ decompositions to
    iteratively arrive at the SVD of the bidiagonal matrix
{
  Q=qr.Q(qr(bid[1:n,1:n]))
  R=qr.R(qr(bid[1:n,1:n]))
  L=t(qr.R(qr(t(R))))
  P=qr.Q(qr(t(R)))
  U3=U3%*%Q
  V3=t(P)%*%V3
  bid[1:n,1:n]=L
  epsilon=max(abs(bid[1:n,1:n]-diag(diag(bid))))
}
V3=t(V3)

U <<- U_A%*%U3
V <<- V_A%*%V3
sig <<- round(bid[1:n,1:n],digits=6)

if (nrow(B)<ncol(B))
{
  mat1=t(U%*%sig%*%t(V))
  temp1 <<- U
  U <<- V
  V <<- temp1
}
else
{
  mat1=U%*%sig%*%t(V)
}
```

```
    for (i in 1:n)
    {
      if (sig[i,i]<0)
      {
        U[,i]=U[,i]*-1
        sig[i,i]=-sig[i,i]
      }
    }

    cat('\nU is:\n')
    print(round(U,digits=6))   #The final matrix U which may not be a square
       matrix; orthonormal basis extension may be required to do so
    cat('\nSigma is:\n')
    print(sig)                 #The diagonal matrix of singular values
    cat('\nV is:\n')
    print(round(V,digits=6))   #The final matrix V which can be a non-square
       matrix

    cat('\nThe product U%*%Sigma%*%t(V) is:\n')
    print(round(mat1,digits=5))
}

#Examples

mat=matrix(c(2,6,1,8,5,0,7,4,3),3,3)
SVD(mat)

mat=matrix(1:15,5,3)
SVD(mat)

mat=matrix(1:40,5,8)
SVD(mat)
```