hw2

n.hwang

June 25, 2019

Introduction

One common practice to reduce the dimension and extract useful features is tensor decomposition. While there are many such decomposition algorithms, Tucker decomposition is popular given its elegance and interpretability of its components, despite the non-trivial nature of its implementation. Luckily for some, there is a R package, **rTensor**, that implements Tucker decomposition. In this project, I implement Tucker decomposition using first principles of the alternative least squares (ALS) and singular value decomposition (SVD), and compare results to those of the R package. Given a large array of order 3, I build a set of functions that perform a series of tensor mode projections and reconstructions to applly Tucker decomposition to get the core tensor and factorization matrices that comprise the original tensor array. I test my implementation versus the **rTensor** method using a fictitious movie ratings data stored in a 3-order tensor format.

Tucker Decomposition using the Alternative Least Squares

Hastie et al's *Elements of Statistical Learning* defines Tucker decomposition of a tensor $\mathcal{X}$ as follows:

$$\mathcal{X} \sim \mathcal{G} \times_1 A \times_2 B \times_3 C$$

where $\mathcal{G}$ is a core tensor that represents the interaction among various dimensions of the tensor and A, B, C are factorization matrices. As notated by the approximation symbol, the expressions are equivalent only when the ranks of the factorization matrices are the same as those of the tensor. Given that we want to find the minimum rank structure, the problem becomes one of an optimization, which happens to be NP-hard. \

However, one can use ALS to find an optimal solution modulo tolerance levels for approximation error as noted by the Frobenius norm. In particular, the goal becomes one of finding the factorization matrices $A^{(k)}$ such that we maximize the below quadratic norm:

$$\|A^{(k)T} X_{(k)} \big(A^{(n)} \otimes ... \otimes A^{(k+1)} \otimes A^{(k-1)} \otimes ... \otimes A^{(1)}\|^2$$

such that $A^{(k)} \in \mathbb{R}^{I_k \times R_k}$ are column-wise orthogonal.

The ALS-SVD Algorithm for Array Decomposition

I implement the following algorithm for decomposing multi-dimensional array using the first principles from ALS and SVD.\

Given a 3-order tensor:\

1) Initialize $A^{(1)}, A^{(2)}, A^{(3)} \in \mathbb{R}^{I_3 \times R_3}$ using SVD;\

2) While error > tolerance do:\

a) Y = $\mathcal{X} \times_1 \times_1 A^{(1)T} \times_2 A^{(2)T} \times_3 A^{(3)T}$\

b) $A^{(k)} \leftarrow R_k leadingleftsingularvaluesofY_{(k)}$\

c) $\mathcal{G} = \mathcal{X} \times_1 A^{(1)T} \times_2 A^{(2)T} \times_3 A^{(3)T} \backslash$

d) $M = \mathcal{G} \times_1 A^{(1)T} \times_2 A^{(2)T} \times_3 A^{(3)T} \backslash$

e) error $= \|\mathcal{X} - M\| / \|X\|$.

Implementation

Helper functions

Before I can implement the ALS decomposition, the following helper functions are needed. The first is **matricize**, which *matricizes* a multidimensional array into a 2-dimensional matrix so as to allow for n-mode tensor multiplications in the algorithm described above. \

```
matricize <- function(M,mode){
  mode_dim = dim(M)[mode]
  axes = c(1,2,3)
  axes = axes[!axes %in% mode]
  i = dim(M)[axes[1]]; j = dim(M)[axes[2]]
  dims = c(i,j)
  temp = NULL
  if (mode == 1){
    for (i in seq(1,dims[2],1)){
      for (j in seq(1,dims[1],1)){
        temp <- cbind(temp, M[,j,i])
      }
    }
  } else if (mode == 2){
    for (i in seq(1,dims[2],1)){
      for (j in seq(1,dims[1],1)){
        temp <- cbind(temp, M[j,,i])
      }
    }
  } else {
    for (i in seq(1,dims[2],1)){
      for (j in seq(1,dims[1],1)){
        temp <- cbind(temp, M[j,i,])
      }
    }
  }
  return(temp)
}
```

Next, once the n-mode tensor multiplications are done for a given pair of matrices, we need to reconstruct the result into the 3-order tensor to allow for the appropriate n-mode projection to continue on with the next n-mode tensor multiplication. This method follows below, called **reconstruct**.

```
reconstruct <- function(matricized, rank, mode){
  mode_dim = rank[mode]
  axes = c(1,2,3)
  axes = axes[!axes %in% mode]
  i = rank[axes[1]]; j = rank[axes[2]]
  dims = c(i,j)
  tensor = array(0, dim=rank)
```

```
  if (mode == 1){
    if (mode_dim == dim(matricized)[2]){
      matricized = t(matricized)
    }
    for (k in seq(0,dims[2]-1,1)){
      tensor[,,k+1] <- matricized[,(k*dims[1]+1):((k+1)*dims[1])]
    }
  } else if (mode == 2) {
    for (k in seq(0,dims[2]-1,1)){
      tensor[,,k+1] <- matricized[(k*dims[1]+1):((k+1)*dims[1]),]
    }
  } else {
    tensor = array(t(matricized), dim= rank)
  }
  return(tensor)
}
```

ALS-SVD Tucker Decomposition Implementation

Now, we are ready to implement the main algorithm for decomposing a multidimensional array. I call it **my_tucker_als** to distinguish it from the **tucker_als** method that comes standard with the **rTensor** package in R. The method takes four arguments, namely the original tensor to decompose, the desired rank of the core tensor, the tolerance for the Frobenius norm-based error metric, and finally the maximum number of loop iterations we want to tolerate should the error not fall below the threshold level.

```
my_tucker_als <- function(A, rank, tol, maxiter){
  r1 = rank[1]; r2 = rank[2]; r3 = rank[3]
  x = dim(A)[1]; y = dim(A)[2]; z = dim(A)[3]

  ## initialization of factor matrices
  A1 = svd(matricize(A,1),2)$u
  A2 = svd(matricize(A,2),2)$u
  A3 = svd(matricize(A,3),1)$u

  counter = 1
  while ( counter < maxiter ){
    ### Iteration for A1
    Y2 <- t(matricize(A,2)) %*% A2
    Y2 <- reconstruct(Y2, c(x,r2,z), 2)
    Y3 <- t(matricize(Y2,3)) %*% A3
    Y3 <- reconstruct(Y3, c(x,r2,r3), 3)
    s <- svd(matricize(Y3,1))
    A1 <- s$u[,1:r1]

    ### Iteration for A2
    Y1 <- t(matricize(A,1)) %*% A1
    Y1 <- reconstruct(Y1, c(r1,y,z), 1)
    Y3 <- t(matricize(Y1,3)) %*% A3
    Y3 <- reconstruct(Y3, c(r1,y,r3), 3)
    s <- svd(matricize(Y3,2))
    A2 <- s$u[,1:r2]

    ### Iteration for A3
```

```
    Y1 <- t(matricize(A,1)) %*% A1
    Y1 <- reconstruct(Y1, c(r1,y,z), 1)
    Y2 <- t(matricize(Y1,2)) %*% A2
    Y2 <- reconstruct(Y2, c(r1,r2,z), 2)
    s <- svd(matricize(Y2,3))
    A3 <- s$u[,1:r3]

    # Computation of G
    Y1 <- t(matricize(A,1)) %*% A1
    Y1 <- reconstruct(Y1, c(r1,y,z), 1)
    Y2 <- t(matricize(Y1,2)) %*% A2
    Y2 <- reconstruct(Y2, c(r1,r2,z), 2)
    G <- t(matricize(Y2,3)) %*% A3
    G <- reconstruct(G, c(r1,r2,r3), 3)

    # Computation of Error
    temp <- outer(t(matricize(G,1) %*% t(A1)) %*% t(A2), A3)
    error = fnorm(A-as.tensor(temp))
    rel_error = error / fnorm(as.tensor(A))
    if (error < tol){
      break
    }
    counter = counter + 1
  }
  return(list(G,A1,A2,A3,error,rel_error,counter))
}
```

Simulation

Below, we come up with a fictitious movie recommendation array of dimensions 5 x 4 x 3, where five movies are rated by 4 individuals at 3 different times. We call this array **M**.

```
M = array(0, dim=c(5,4,3))
for (i in seq(1,5,1)){
  for (j in seq(1,4,1)){
    for (k in seq(1,3,1)){
      M[i,j,k] <- max(0,rnorm(1)-.6)
    }
  }
}
```

Suppose that we want to come up with the core tensor of dimensions (2,2,1). Then, we simply feed this rank into the **my_tucker_als** method above. Below, one can see that the first output is the 2x2x1 core tensor, followed by 3 factorization matrices of dimensions 5x2, 4x2, and 3x1. We can confirm that the n-mode tensor multiplication of these matrices yields the correct dimension for reconstructing the original tensor of dimensions 5x4x3. The Frobenius error is 2.45, while the relative error is 0.68.

```
my_tucker_als(M,c(2,2,1),.1,15)
```

```
## [1] 2.469693
## [1] 2.450374
## [1] 2.450174
## [1] 2.450168
```

```
## [1] 2.450168
## [1] 2.450168
## [1] 2.450168
## [1] 2.450168
## [1] 2.450168
## [1] 2.450168
## [1] 2.450168
## [1] 2.450168
## [1] 2.450168
## [1] 2.450168
```

```
## [[1]]
## , , 1
##
##               [,1]          [,2]
## [1,]  2.036216e+00 -7.448014e-06
## [2,] -5.555585e-06  1.658425e+00
##
##
## [[2]]
##            [,1]         [,2]
## [1,] -0.9258573  0.285097807
## [2,] -0.1365716 -0.776698815
## [3,] -0.2134596 -0.068532478
## [4,] -0.1131077 -0.002513874
## [5,] -0.2564724 -0.557454187
##
## [[3]]
##            [,1]       [,2]
## [1,] -0.6926092  0.1568356
## [2,] -0.1992690 -0.9312853
## [3,] -0.6231994  0.2353531
## [4,] -0.3036559 -0.2296067
##
## [[4]]
## [1] 0.72372187 0.68918620 0.03534172
##
## [[5]]
## [1] 2.450168
##
## [[6]]
## [1] 0.6821865
##
## [[7]]
## [1] 15
```

Now, we compare the results above with the Tucker_ALS implementation in the **rTensor** package. One can verify that the results above closely match the ones returned by the method in the package.

```
rtucker <- tucker(as.tensor(M), c(2,2,1))
```

```
##
  |
  |                                                                      |   0%
  |
  |===                                                                   |   4%
```

```
  |
  |=====                                                                 |   8%
  |
  |========                                                              |  12%
  |
  |==========                                                            |  16%
  |
  |======================================================================| 100%
```

```
rtucker$Z
```

```
## Numeric Tensor of 3 Modes
## Modes:  2 2 1
## Data:
## [1]  2.0356059175 -0.0001301419 -0.0001825216  1.6591729681
```

```
rtucker$U
```

```
## [[1]]
##            [,1]         [,2]
## [1,] -0.9256341  0.286131277
## [2,] -0.1378806 -0.776951673
## [3,] -0.2130665 -0.067642164
## [4,] -0.1129034 -0.002214398
## [5,] -0.2569942 -0.556681798
##
## [[2]]
##            [,1]       [,2]
## [1,] -0.6923645  0.1577224
## [2,] -0.2006356 -0.9313282
## [3,] -0.6229810  0.2359339
## [4,] -0.3037621 -0.2282251
##
## [[3]]
##            [,1]
## [1,] 0.72417232
## [2,] 0.68871638
## [3,] 0.03527325
```

```
rtucker$fnorm_resid
```

```
## [1] 2.450168
```