KSVD8054: An R Package for the Implementation of KSVD and LC-KSVD

Mitch Kinney

May 5, 2017

Abstract

KSVD8054 is a new R package that was designed to implement algorithms from papers "K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation" by Aharon et al. [1] and "Learning A Discriminative Dictionary for Sparse Coding via Label Consistent K-SVD" by Jiang et al. [2]. In the first paper a novel algorithm called KSVD is introduced to determine a sparse representation of signals using a linear combination of columns in a discriminative dictionary. It is a combination of the K-means algorithm and an SVD approximation of rank one matrices. In the second paper an extension of the KSVD algorithm is proposed which is called LC-KSVD. This extension incorporates labels into the KSVD algorithm to better handle the added information of classes within the signals. This package will be, to the best of my knowledge, the first implementation in R.

Introduction

The KSVD algorithm sets out to try to solve the problem of finding a vector $x \in \mathbb{R}^{\mathbb{K}}$ and a matrix $D \in \mathbb{R}^{n\,x\,K}$ such that for a response vector $y \in \mathbb{R}^n$ there exists the relationship $y = Dx$ or $||y - Dx||_2 \leq \epsilon$ for some small ϵ. A limitation of this representation is that x must be the sparsest representation as possible and still satisfy the relationship $||y - Dx||_2 \leq \epsilon$. In most cases there will be a matrix of responses $Y = \{y_i\}\ i = 1, .., N$ and a corresponding matrix $X = \{x_i\}\ i = 1, .., N$. Then the problem becomes finding the D, X for some tolerance T_0 that satisfies

$$\arg\min_{D,X} ||Y - DX||_F^2 \text{ subject to } \forall i\ ||x_i||_0 \leq T_0$$

Where $||\cdot||_0$ is the zero-norm or the number of non zero terms in a vector. The algorithm proposed to solve this problem in [1] is a two part algorithm that iteratively solves for the X matrix that minimizes the objective function given a fixed dictionary D and then solves for the dictionary D given a fixed X. In the first part, KSVD takes advantage of pursuit algorithms to solve for individual vectors x_i of the X matrix given the corresponding vector y_i and the dictionary D. There are many options for pursuit algorithms and in this package the user has the choice of Matching Pursuit, Orthogonal Matching Pursuit and FOCUSS. Orthagonal Matching Pursuit is the recommended algorithm by [1] based on the combination of its speed and accuracy. Then in the second part of KSVD the authors propose a way to solve for individual columns $d_{,k}$ in D by approximating error matrices by rank one matrices found using SVD. More details of both parts will come in later sections. An important note is that the columns of D must be L_2 normalized for purposes of uniqueness.

Another algorithm the package KSVD8054 implements is an extension of KSVD called LC-KSVD which stands for Label Consistent KSVD. This algorithm was proposed in [2] and uses the KSVD idea with the added feature of class labels. This allows datasets with signals from different classes to take advantage of this additional information in solving for the dictionary D and also in prediction of future data points. Additional terms involved in LC-KSVD are the number of classes m, the $H \in \mathbb{R}^{m\,x\,N}$ matrix which is the label matrix and the $Q \in \mathbb{R}^{K\,x\,N}$ matrix which is the input tracking matrix. For the H matrix the entry h_{ij} will be one if the jth column in Y belongs to the ith class and for the Q matrix the entry q_{ij} will be one if the ith column of D was initialized using a column of Y that is from the same group as the jth column of Y. Then the problem that LC-KSVD attempts to solve is finding D, X, W, A to satisfy

$$\arg\min_{D,X,W,A} ||Y - DX||_F^2 + \alpha||Q - AX||_F^2 + \beta||H - WX||_F^2 \text{ subject to } \forall i\ ||x_i||_0 \leq T_0$$

Where the matrices A, W are linear mapping matrices from X to Q and X to H respectively and the α, β scalars are penalty terms provided by the user. In plain English this function attempts to find a combination of dictionary D and sparse encoding matrix X to represent the response Y while making sure that columns of D that were initialized from classes in Y are being used to map those same classes and that columns from the same classes in Y are using similar columns of D. Then classification of a new point y_* can be done by finding the sparse representation vector x_* that corresponds to y_* given the trained dictionary D and using the trained W matrix to see what class used those same columns of D the most in training.

This paper will give details on the mathematical concepts being used, how they are coded in the package, and some reproducible experiments on simulated and real data. There will also be examples on how to use the functions in the package.

KSVD - Pursuit Algorithms

Pursuit algorithms solve the problem of given a dictionary D and response vector $y \in \mathbb{R}^n$ what is the best way to solve $\arg\min_x ||y - Dx||_2^2 \leq \epsilon$ for some ϵ small while adhering to the restriction that $||x||_0 \leq T_0$. Notice that this needs to be done for all columns $\{y_i\}$ i = 1,..,N but in this section I will only focus on how to solve it for a single vector. I will discuss the three pursuit algorithms that are implemented in the package: Matching Pursuit, Orthogonal Matching Pursuit and FOCUSS. Again this author agrees with the authors of [1] that Orthogonal Matching Pursuit is best to use.

Matching Pursuit

The first pursuit algorithm implemented is Matching Pursuit. The theory and implementation of the algorithm comes from [3]. The basic idea is to start with two values: an approximation $a_0 = 0$ and a residual $r_0 = y$. Then iteratively update both by first solving for the index λ_k which satisfies for a column in D

$$\lambda_k = \arg\max_{\omega} |\langle r_{k-1}, d_\omega \rangle|$$

Where $\langle \cdot, \cdot \rangle$ is the normal inner product. This is the index of the column in D which most closely matches the residual at time k. Then update the residual and approximation by doing

$$a_k = a_{k-1} + \langle r_{k-1}, d_{\lambda_k} \rangle d_{\lambda_k}$$
$$r_k = y - a_k$$

Do this until some convergence criteria is met (like $r_k < 10^{-6}$) and then save the coefficients $\langle r_{k-1}, d_{\lambda_k} \rangle$ from every iteration k along with the column index λ_k. This will give the column x associated with y. To satisfy the sparsity requirement I run the Matching Pursuit algorithm but only keep the first T_0 columns of D chosen and then iterate over those columns until a stopping rule is met. The code to run the Matching Pursuit is

```
while(<stopping rule not met>){

#update residual and approximation
r.now <- r.next
a.now <- a.next

#find.atom returns column number of D that maximizes the absolute innerproduct
# of r.now and the cols of D
lambda <- find.atom(r.now, D)

#if lambda is not in lambda set then add it
if(!(lambda %in% lambda.set)){
        lambda.set <- c(lambda.set,lambda)
}

#regular mapping pursuit based off of p.7 in Tropp
a.next <- a.now + sum(r.now*D[,lambda])*D[,lambda]

#update x[lambda] with how much the vector contributes
x[lambda] <- x[lambda] + sum(r.now*D[,lambda])

#get next residual term
r.next <- y.now - a.next
}
```

In practice I found this algorithm takes many iterations to converge to a reasonable stopping rule so it is not recommended for speed purposes. The function that will do this is called My.matching.pursuit(D,Y,Tol, orthogonal = F). Where D is the dictionary, Y is the response matrix, Tol is the T_0 and orthogonal is a boolean variable which is False. In the next section I will talk about Orthogonal Matching Pursuit which is what you get if orthogonal is set to True.

Orthogonal Matching Pursuit

The second pursuit algorithm available in the package is Orthogonal Matching Pursuit. This algorithm is similar to Matching Pursuit except it has a different update function for the approximation and at maximum will iterate T_0 number of times. The initialization is the same with the approximation $a_0 = 0$ and the residual $r_0 = y$ but instead the update will now be

$$\lambda_k = \arg\max_{\omega} |\langle r_{k-1}, d_\omega \rangle|$$
$$a_k = \arg\min_{z} ||y - \tilde{D}z||_2^2, \text{ where } \tilde{D} = [d_{\lambda_1}, d_{\lambda_2}, .., d_{\lambda_k}]$$
$$r_k = y - a_k$$

The $\tilde{D}$ is the columns of the dictionary D that have been selected in all previous steps as well as the current step. Solving for this is a familiar problem which uses the technique of least squares. This is reflected in the code to solve Orthogonal Matching Pursuit which is

```
while(<stopping rule not met>){

#update residual and appoximation
r.now <- r.next
a.now <- a.next

#find.atom returns column number of D that maximizes the absolute innerproduct
# of r.now and the cols of D
lambda <- find.atom(r.now, D)

#if lambda is not in lambda set then add it
if(!(lambda %in% lambda.set)){
        lambda.set <- c(lambda.set,lambda)
}

#get least squares of estimator for next approximation
beta <- qr.solve(crossprod(D[,sort(lambda.set)]),
        t(D[,sort(lambda.set)])%*%cbind(y.now))
a.next <- as.vector(D[,sort(lambda.set)]%*%beta)

#get next residual term
r.next <- y.now - a.next
```

This is what you get if you use the same function as before but instead set My.matching.pursuit(D,Y,Tol, orthogonal = T). The reason this method is preferred is because of it's ability to minimize the approximation vector in a closed form solution and for it's speed. Since there is a sparsity requirement and the way the index λ_k is chosen the columns of D selected will be best able to explain the residual at each step. Then you are guaranteed to have the T_0 most important columns of D chosen so the loop will only run T_0 times at most. To test I ran this function 500 times on a vector $y \in \mathbb{R}^{20}$ with a dictionary $D \in \mathbb{R}^{20\,x\,50}$ and $T_0 = 5$ I get the following timing:

```
system.time(replicate(500,My.matching.pursuit(D,y,Tol = 5,orthogonal = T)))
user  system elapsed
1.28    0.00    1.28
```

FOCUSS

The last pursuit algorithm is FOCUSS which is an iterative re-weighted least squares approach whose implementation comes from [4]. FOCUSS stands for FOcal Underdetermined System Solver. FOCUSS iterates over two parts until convergence: a solution vector and a weight matrix for pruning the solution space. So at each step for a given weight matrix the algorithm solves

$$\arg\min_x ||W^{-1}a||_2^2 \text{ subject to } Da = y$$

Initializations for the weight matrix is $W_0 = I$ and solution is $x_0 = 1$. Then the update for each variable is solved using Lagrangian methods to be

$$x_k = W_{k-1}^2 D^T (AW_{k-1}^2 D^T)^+ y$$
$$W_k = \text{diag}(|x_k^1|^{\frac{1}{2}}, |x_k^2|^{\frac{1}{2}}, ..., |x_k^K|^{\frac{1}{2}})$$

Where $(\cdot)^+$ represents the Moore-Penrose generalized inverse. In code this looks like

```
while(iterating){

#assign candidate x.vec according to algorithm, ginv is moore penrole gen inverse
x.vec <- (X.weight^2)%*%t(D)%*%ginv(D%*%(X.weight^2)%*%t(D))%*%cbind(y.now)
colnames(x.vec) <- NULL

#assign new weight matrix according to algorithm
X.weight <- diag(abs(as.vector(x.vec))^(1-.5))

#decide to exit, stopping rules were suggested
if(norm(x.vec - x.prev, "2") < error || k >= max.iters){
        iterating <- FALSE
}

#update comparison variable
x.prev <- x.vec

#update iterator
k <- k + 1

}
```

This algorithm can be called with the function My.FOCUSS(D, Y , Tol).

KSVD - SVD to update D

In this step every column in dictionary matrix D is updated according to the algorithm provided by [1]. This part of KSVD takes advantage of using SVD to find the closest rank one matrix in Frobenius norm to a higher ranking matrix. To see how this is used look at the original problem without the sparsity constraint to see for response matrix $Y \in \mathbb{R}^{n\,x\,N}$, dictionary matrix $D \in \mathbb{R}^{n\,x\,K}$ and sparse linear encoder matrix $X \in \mathbb{R}^{K\,x\,N}$

$$\begin{aligned} ||Y - DX||_F^2 &= ||Y - \sum_{j=1}^{K} d_{,j} x_{j,}^T||_F^2 \\ &= ||(Y - \sum_{j \neq k} d_{,j} x_{j,}^T) - d_{,k} x_{k,}^T||_F^2 \\ &= ||E_k - d_{,k} x_{k,}^T||_F^2 \end{aligned}$$

Where $d_{,k}$ is the kth column of D, $x_{k,}$ is the kth row of X and E_k is the matrix of dissimilarity when accounting for all column row combinations of D and X except for the kth column row combination. Then since we want to have $d_{,k} x_{k,}^T$ equal as closely to this E_k as possible the authors first remove the columns of E_k that are zeros in the row of $x_{k,}$ since that would imply the dictionary column $d_{,k}$ is not contributing to mapping those columns of Y so they can be discarded in the update. Call this new matrix E_k^{Ω}. Then the authors do an SVD decomposition of this new matrix to get $E_k^{\Omega} = U \Delta V^T$. Since the matrix Δ is a diagonal matrix of singular values where the number of singular values corresponds to the rank of the matrix, to get a rank one approximation the authors consider if only the first diagonal element was non-zero. Then this would be the matrix of $u_{,1} \Delta_{1,1} v_{,1}^T$ which we can use to update the column $d_{,k}$ and row $x_{k,}$. The actual update is

$$\begin{aligned} d_{,k}^{new} &= u_{,1} \\ x_{k,}^{new} &= \Delta_{1,1} v_{,1}^T \end{aligned}$$

The authors choose to separate the SVD rank one approximation this way to keep the columns of D L_2 normalized. Notice that both the pursuit algorithm and the SVD supply updates for the X matrix. The authors suggest choosing the update which reduces the Frobenius norm the most. How the SVD update looks for updating column i of D and row i of X in the code is

```
#compute error matrix, select only columns which are being used by
# ith col of D, and do SVD
E <- Y - (D%*%X - cbind(D[,i])%*%X[i,])
E.R <- E[,which(X[i,] != 0,arr.ind = T)]
E.svd <- svd(E.R)

#update column i of D
D[,i] <- E.svd$u[,1]

#update row i of X
x.new <- E.svd$d[1] * E.svd$v[,1]
for(j in 1:length(which(X[i,] != 0, arr.ind = T))){
        X[i,which(X[i,] != 0, arr.ind = T)[j]] <- x.new[j]
}
```

The function that implements KSVD in its entirety is My.Ksvd(Y, init.D, pursuit, Tol, max.iters). The function requires an initial dictionary since it is not implemented within the function to create one, the pursuit algorithm to use ("MP" for matching pursuit, "OMP" for orthogonal matching pursuit, and "F" for FOCUSS), and the maximum number of iterations to take if the user does not want to wait til the stopping rule is met (which is $||Y - DX||_F^2 \leq 10^{-6}$ or $|\,||Y - D_{k-1}X_{k-1}||_F^2 - ||Y - D_k X_k||_F^2| \leq 10^{-10}$). Later I will introduce a function to create an initial dictionary so the user is not left having to make one on their own.

LC-KSVD

The extension of KSVD proposed in [2] is LC-KSVD which stands for Label Consistent KSVD and utilizes the additional information that the response matrix Y might be made of signals from different classes. This proposed extension uses two main ideas: first that the initial dictionary will be made up of fuzzed signals in Y so that when building the x column for y in training, it is more likely to use columns of D that were initiated with the fuzzed columns of Y that have the same class as y. The second idea is that columns of Y with the same class will use similar columns of D in their representation. The two matrices of known values introduced to help do this are Q and H where $Q \in \mathbb{R}^{K\,x\,N}$ and element q_{ij} is one if dictionary column i was initialized using a fuzzed column of Y that has the same class as the jth column of Y and zero otherwise. And $H \in \mathbb{R}^{m\,x\,N}$ (m is the number of classes) where element h_{ij} is one if the jth column of Y belongs to class i and zero otherwise. The two new matrices to be solved for are $A \in \mathbb{R}^{K\,x\,K}$ and $W \in \mathbb{R}^{m\,x\,K}$ which are both linear maps from X to Q and X to H respectively. I'll show later how the W matrix is used for prediction. The overall problem that LC-KSVD solves is finding D, X, A, W to satisfy

$$\underset{D,X,A,W}{\arg\min} ||Y - DX||_F^2 + \alpha||Q - AX||_F^2 + \beta||H - WX||_F^2 \text{ subject to } \forall i \; ||x_i||_0 \leq T_0$$

Where α and β are scalar penalty terms that the user can adjust to control the contribution of each penalized norm. The algorithm implemented in my package takes advantage of a reparameterization that the authors note to transform this problem into something that KSVD can handle. Since all terms include the X matrix I can rewrite the objective function as

$$Y_{new} = \begin{bmatrix} Y \\ \sqrt{\alpha}Q \\ \sqrt{\beta}H \end{bmatrix}, \; D_{new} = \begin{bmatrix} D \\ \sqrt{\alpha}A \\ \sqrt{\beta}W \end{bmatrix}$$

And now solving the KSVD will look like

$$\underset{D_{new},X}{\arg\min} ||Y_{new} - D_{new}X||_F^2 \text{ subject to } \forall i \; ||x_i||_0 \leq T_0$$

Note that this is not what the authors of [2] did to solve the problem efficiently in their paper but this is how I chose to implement it in my package. The code for this is pretty straightforward as it is just cbinding() the inputted values and throwing them to My.KSVD(). The function to solve this problem is My.LCKsvd(Y, H, D.size, Tol, alpha, beta, pursuit, ksvd.iters, classification). The classification term is a boolean argument whether to return the W matrix to do classification since it needs to be de-L_2 normalized with respect to the larger combined matrix D_{new}. This function has the dictionary initialization built in so there is no need to manually initialize the dictionary.

KSVD and LC-KSVD Dictionary Initialization

In [2] the authors suggest a way to initialize the original dictionary D which I have implemented in my package. The function is LSKsvd.init(Y, H, D.size, Tol, lambda1, lambda2) where D.size is K and lambda1 and lambda2 are penalty terms in ridge regression to initialize A and W. The basic idea is sample an appropiate number of columns (relative to D.size) from Y and fuzz them for a single class. Then use the fuzzed columns as the initial dictionary in KSVD with the reduced response matrix being all the columns of Y in the class and iterate about 5 times. The dictionary that is outputted from these runs will be combined with all the other outputted dictionaries from the other classes to make the full initialized dictionary. I choose to build the columns in the dictionary from the classes in Y as equally as possible.The code to do this for a single class i, and desired number of dictionary columns dict.elements is

```
#randomly get columns of Y in class i to fuzz and use in dictionary
indices <- sample(which(H[i,] == 1, arr.ind = T), dict.elements)
D.part <- Y[,indices] + matrix(rnorm(dict.elements*nrow(Y)), nrow = nrow(Y))
D.part <- t(t(D.part)/apply(D.part,2,norm,type = "2"))
```

```
#need to keep track of which columns of Y are contributing to which columns of D
Q.start <- matrix(0,nrow = dict.elements, ncol = ncol(Y))
Q.start[,which(H[i,] == 1,arr.ind = T)] <- 1

#update D.init, first need to normalize the chosen columns of Y
input.Ys <- Y[,which(H[i,] == 1, arr.ind = T)]
input.Ys <- t(t(input.Ys)/apply(input.Ys,2,norm,type = "2"))

out.ksvd <- My.Ksvd(input.Ys,D.part, pursuit = "OMP",Tol, max.iters = 5)
D.init <- cbind(D.init, out.ksvd$D)
```

To initialize the A_0 and W_0 the authors in [2] suggest to use a ridge regression type approach which is what the lambda1 and lambda2 are for. In the equation for A_0, Q is used but this is automatically generated within LCKsvd.init() so it is not necessary for the user to input it themselves. The initialization equations are

$$W_0 = (XX^T + \lambda_1 I)^{-1} XH^T$$
$$A_0 = (XX^T + \lambda_2 I)^{-1} XQ^T$$

This function's time complexity will be proportional to the dictionary size and the number of classes. The time it takes to initialize a dictionary for $Y \in \mathbb{R}^{20\,x\,500}$ with dictionary size 50, number of classes 5, and tolerance 3 is

```
system.time(LCKsvd.init(Y, H, D.size, Tol, lambda1 = 1, lambda2 = 1))
user  system elapsed
4.67    0.00    4.69
```

Simulation Results for KSVD

To show how to use the LCKsvd.init() and My.KSVD() functions to do an analysis on a response matrix Y and proof that the function is working I will do a couple of simulation studies and report the results. I will have a baseline of the four parameters of $n = 25, N = 500, K = 50$ and $T_0 = 5$. Then I'll perturb each one in different directions to give a good idea of the effectiveness and speed of the algorithms. I will use a maximum iteration of 80 inside the KSVD algorithm as that seems standard in both papers I read rather than wait until convergence. To prove the effectiveness of the algorithms I will create a true dictionary D and sparse encoder matrix X and compute $Y = DX + \Sigma$ where $\Sigma_{i,j} \overset{ind}{\sim} N(0, \frac{1}{2})$. Then I will compare the Frobenius norm of my output to the Frobenius norm of the noise to show I am fitting Y well. The base code I will use is

```
#set parameters and seed
n <- 10; N <- 500; D.size <- 50; Tol <- 5
set.seed(8053)

#initialize the true dictionary by drawing from a random uniform(-1,1) and
#normalizing the columns
D.true <- matrix(runif(n*D.size,-1,1),nrow = n)
D.true <- t(t(D.true)/apply(D.true,2,norm,type = "2"))

#initialize the true X by randomly placing Tol number of ones in each column
X.true <- matrix(0,nrow = D.size,ncol = N)
for(i in 1:N){
indices <- sample(1:D.size,Tol,replace = F)
X.true[indices,i] <- 1
}
```

```
#create Y with small white noise (fuzz)
Y <- D.true%*%X.true + matrix(rnorm(n*N, sd = sqrt(.5)),ncol = N)

#time the entire process
init.time <- system.time(out.init <-
        LCKsvd.init(Y,H = matrix(1,nrow = 1,ncol = N),D.size,Tol,1,1))
D.init <- out.init$D.init
ksvd.time <- system.time(out.ksvd <- My.Ksvd(Y,D.init,"OMP",Tol,80))
init.time + ksvd.time

#assign output and compare effectiveness
D.ksvd <- out.ksvd$D
X.ksvd <- out.ksvd$X

norm(Y - D.true%*%X.true, "F")
norm(Y - D.ksvd%*%X.ksvd, "F")
```

Notice that to initialize the dictionary in KSVD I use a row vector for the matrix H of all ones to represent the matrix Y has only one class. The results from using this code and changing only the parameters from the baseline is

n	$\|\|Y - D_{true}X_{true}\|\|_F^2$	$\|\|Y - D_{ksvd}X_{ksvd}\|\|_F^2$	time (s)
10	50.04	4.95	120.8
25	78.69	35.99	151.9
50	111.55	75.41	145.9

N	$\|\|Y - D_{true}X_{true}\|\|_F^2$	$\|\|Y - D_{ksvd}X_{ksvd}\|\|_F^2$	time (s)
250	55.29	21.17	66.11
500	78.69	35.99	151.9
1000	111.37	55.91	267.7

K	$\|\|Y - D_{true}X_{true}\|\|_F^2$	$\|\|Y - D_{ksvd}X_{ksvd}\|\|_F^2$	time (s)
25	78.95	46.96	112.1
50	78.69	35.99	151.9
100	78.68	25.76	178.3

T_0	$\|\|Y - D_{true}X_{true}\|\|_F^2$	$\|\|Y - D_{ksvd}X_{ksvd}\|\|_F^2$	time (s)
3	79.13	47.41	85.23
5	78.69	35.99	151.9
10	78.82	17.08	271.1

From the tables I see a reduction in the Frobenius norm of the true difference which is the noise and the outputted dictionary, sparse encoder matrix combination for each example. This implies that the algorithm is working but unfortunately fitting some noise which is a sign of over fitting. The error decreases as both K and T_0 increases which is expected but so also does the time the algorithm takes.

In [1] the authors note that a decrease in the MSE of $||Y - DX||_F^2$ at each step is not guaranteed and is based on the how well the pursuit algorithm does. They also say that as long as n is much larger than T_0 the pursuit algorithms work well and KSVD will converge. To see an example of this I have included the norms at each step as output of My.KSVD() to make sure this relationship holds. For the first example I will simulate a response matrix $Y \in \mathbb{R}^{5\,x\,500}$ and use a dictionary $D \in \mathbb{R}^{5\,x\,50}$ with $T_0 = 3$. The norm versus iteration plot is in Figure 1. Clearly there is not a decrease in MSE at every step. If I increase n though and look at the case where $Y \in \mathbb{R}^{40\,x\,500}$ the plot looks much better. We see in Figure 2 it's not perfect but it's also not hard to imagine that as the margin between n and T_0 increases there will be the desired property that the Frobenius norm either decreases or stays the same at every step.

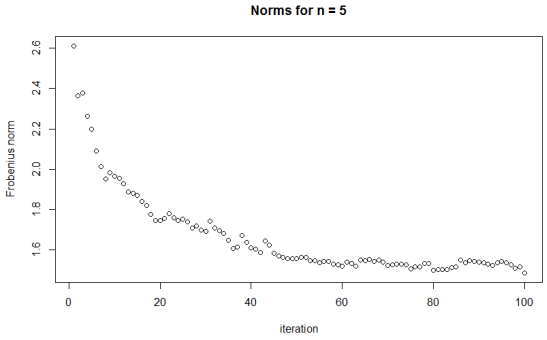

Figure 1: Tracking norms with n = 5

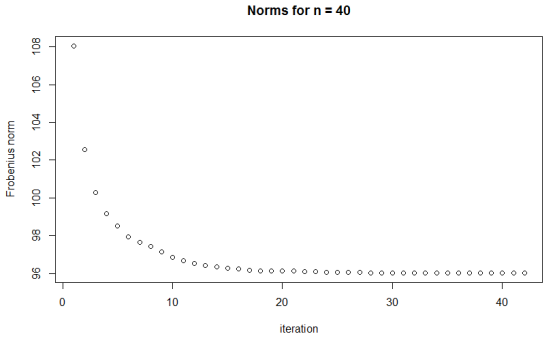

Figure 2: Tracking norms with n = 40

Overall I would say the algorithm is effective and hopefully the user is willing to wait a couple of minutes. There is no other implementation of KSVD in R that I know of so I am not sure how well my algorithm does in comparison to a professionally written one.

Real Data Results for LC-KSVD

To show an example of using My.LCKsvd() I will use the dataset located at `"http://www.tc.umn.edu/~elock/TCGA_Breast_Data.Rdata"` which is a dataset from genomics. In the dataset there are 348 samples of breast cancer tumors which are labeled Basal or Non-Basal and gene expression measurements taken on 645 genes for each sample. So the response matrix $Y \in \mathbb{R}^{645\,x\,348}$ is all gene expressions. The data also comes with a variable that has the class labels of each of the columns. The total number of Basal samples is 66 and Non Basal is 282. I suggest first loading in the data and then following the code to do this example on your own. The data is part of the package. The authors of [2] suggest to do LC-KSVD classification by using the trained dictionary D to get the best linear encoder matrix X through one run of a pursuit algorithm. Then for each column x_i of X use the trained W matrix to compute Wx_i and classify the column y_i as the row j which has the maximum value in the vector Wx_i. I will train the data on approximately 60% of the data and test on the remainder. The code to do this example after the data has been loaded in is

```
#assign the expression matrix and the label vector as booleans
Emat <- Expression.matrix
labs <- ifelse(Subtype == "Basal",T,F)
```

```
#create the H matrix
H <- rbind(ifelse(Subtype == "Basal",1,0),ifelse(Subtype == "Basal",0,1))

#get indices for training and testing
set.seed(8054)
training.indices <- sample(1:348,floor(.6*348))

#assign Y and H for testing and training
Y.train <- Emat[,training.indices]
Y.test <- Emat[,-training.indices]
H.train <- H[,training.indices]
H.test <- H[,-training.indices]

#dictionary size and tolerance
D.size <- 30; Tol <- 3

#alpha and beta chosen arbitrarily
alpha <- 1; beta <- 2

#getting the output and the X matrix for the testing dataset
out.real <- My.LCKsvd(Y.train,H.train,D.size,Tol,alpha,beta,"OMP",80,TRUE)
X.test <- My.matching.pursuit(out.real$D,Y.test,Tol,T)

#creating tables to compare the true table and the outputted table
table(apply(H.test,2,whichmax),apply(out.real$W%*%X.test,2,whichmax))
table(apply(H.test,2,whichmax),apply(H.test,2,whichmax))
```

The time the LC-KSVD took to run was 80 seconds the the table results are

True \Predicted	Basal	Non Basal
Basal	27	2
Non Basal	3	108

True \True	Basal	Non Basal
Basal	29	0
Non Basal	0	111

The algorithm was able to get 96% accuracy which is higher than simply randomly assigning samples based on the known distribution of $\frac{66}{348}$ samples being Basal. In this example the algorithm was effective and the time it took to run was not unreasonable. The dictionary size and tolerance were chosen based on experience with what the papers usually choose but both of these can be varied to possibly improve performance.

Discussion

In this paper I have outlined the functions in the package KSVD8054 and demonstrated their usage. I did not list all the outputs from the functions since those details will be in the help pages of the functions. In summary functions from this package can be used to solve for a sparse reprensentation of a response matrix of signals Y using a dictionary D and a sparse encoder matrix X using KSVD. If the response matrix Y is made up of different classes, the user can implement LC-KSVD to take advantage of this additional information. Thank you!

Bibliography

[1] Aharon et al. K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation. 2006

[2] Jiang et al. Learning a Discriminative Dictionary for Sparse Coding via Label Consistent K-SVD. 2011

[3] Tropp. Greed is good: Algorithmic results for sparse approximation. 2004

[4] Choi. The FOCUSS algorithm powerpoint (Iterative Re-weighted Least Squares).