Imputing Missing Data via Penalized Matrix Decomposition

November 12, 2016

Jennifer Starling

Here we apply penalized matrix decomposition to impute missing data values and perform variable selection for continuous predictor variables in the logistic regression setting.

1. Introduction

Regression in the presence of missing data values is a well-studied but still relevant problem. Many simple proposed solutions exist, including exclusion of observations with one or more missing predictors from the data set, imputing missing values with each predictor's mean value, and others. More complex solutions include those proposed by Hastie et al. in their 1999 paper, "Imputing Missing Data for Gene Expression Arrays." Haste et al. propose imputation of missing data via three methods: Singular Value Decomposition, Nearest-Neighbor Imputation, and Imputation Using Regression, which is an Expectation-Maximization-based approach.

A relatively new solution uses properties of penalized matrix decomposition to impute missing values. This solution is proposed in the Witten, Hastie & Tibshirani (2009) paper, "A Penalized Matix Decomposition, With Applications to Sparse Principal Components and Canonical Correlation Analysis." This paper details an algorithm for recovering the rank-K penalized matrix decomposition, which has several benefits, to be discussed.

The setting for this project is prediction of whether an undergraduate student will pass a basic calculus course. Prediction of the binary response Pass/Fail could be handled in many ways (logistic regression, K-Means, random forest, etc.), which all perform optimally in the absence of missing values.

Data in this setting typically contains many missing values. Predictors are both categorical and quantitative; this project focuses on imputing missing values for continuous predictors. Continuous predictors include SAT scores, ACT scores, high school GPA data, extrapolated undergraduate GPA (as estimated by the University of Texas at Austin), and various standardized testing scores. Students commonly select either the SAT or ACT, and not all students sit for all standardized tests. The data set contains 10167 observations, 28 predictors, and has roughly half of the data missing (43,999 missing values out of 284,676 total values).

We will first summarize the methodology from the Witten, Hastie and Tibshirani paper. We will then present two toy examples to illustrate the methods. Third, we will impute the missing data values using the penalized matrix decomposition methodology. Last, we will briefly present using the methodology for variable selection, with a discussion of penalty parameter selection.

2. Method

Here we present a brief overview of the penalized matrix decomposition method.

Parameter Selections:

The method allows the user to specify a rank (from 1 to the number of predictors) for the decomposition. For purposes of data imputation, a full rank decomposition is used. For variable selection, a rank 1 decomposition allows the number of interesting selected variables to decrease towards one.

The method allows the user two specify two penalty terms, λ_U and λ_V. These penalty terms dicatate the sparsity of the U and V matrices in the penalized decomposition. In the data imputation case, large values of lambda prove useful, as we are not interested in introducing sparsity in this scenario. For variable selection, decreasing lambda progressively results in smaller subsets of variables identified as interesting. This is discussed in more detail later.

Rank 1 Penalized Matrix Decomposition:

We first implement the rank 1 sparse matrix factorization algorithm detailed on pages 519-520 of Witten, Hastie & Tibshirani, 2009. The R function implementation is *sp.matrix.decomp.rank1()*.

The optimization problem is:

$$\begin{aligned} \underset{u\in R^N, v\in R^p}{\text{argmin}} \quad &= ||\mathbf{X} - \mathbf{duv^T}||_\mathbf{F}^\mathbf{2} \text{ subject to} \\ &||u||_2^2 = 1, ||v||_2^2 = 1, \\ &||u||_1 \leq \lambda_u, ||v||_1 \leq \lambda_v, \end{aligned}$$

where F indicates the squared Frobenius norm of a matrix (sum of squared elements).
This problem is equivalent to:

maximize$_{u,v}\mathbf{u^T Xv}$ subject to the same constraints as above.

The equivalence proof is in the appendix of the Witten et al. paper.

The outline of the algorithm is as follows.

1. Initialize v to some random vector where the l2 norm equals 1.
2. Let $u = \frac{S_{\theta_u}(Xv)}{||S_{\theta_u}(Xv)||_2}$
3. Let $v = \frac{S_{\theta_v}(X^T u)}{||S_{\theta_v}(X^T u)||_2}$

Iterate these steps until convergence is reached.
Convergence criteria: $sum(abs(v.old - v.new)^2 < \epsilon$

Technical details:
• $S_{\theta_v}(a) = sign(a)(|a| - \theta_v)_+$ is the same soft-thresholding operator used in the standard LASSO solution.
• θ_u and θ_v are found using binary searches within each iteration.

Rank K Penalized Matrix Decomposition:

Witten et al. outline a recursive strategy to derive the rank K penalized matrix decomposition. This recursion relies on repeated use of the rank 1 algorithm. The R function implementation is *sparse.matrix.decomp.rankk()*.

The outline of the algorithm is as follows.

1. Initialize v to some random vector where the l2 norm equals 1.
2. Let $\mathbf{X^1} = \mathbf{X}$.
3. For $k \in 1, ..., K$:
    a. Find $\mathbf{u_k}, \mathbf{v_k}$ and d_k by using the rank 1 algorithm as described above, with input $\mathbf{X^K}$.
    b. $\mathbf{X^{K+1}} = \mathbf{X^K} - \mathbf{d_k u_k x_k^T}$

The authors note that this algorithm leads to the rank-K SVD of X when the lambda values are set to zero.
Missing Data Method:
Witten et al. note that the algorithm works even when there are missing observations; the missing elements of matrix X can be excluded from all computations. The algorithm is essentially identical to the rank K method above.

The algorithm is as follows.

maximize$_{u,v} \sum_{(i,j)\in C} X_{ij} u_i v_j$ subject to the same constraints as above. C indicates the set of indices where X has nonmissing values.

3. Examples

This section presents two examples. The first example illustrates using penalized matrix decomposition to impuate missing data. The first second example illustrates the intuition regarding the lambda penalty values and variable selection.

Example 1:

This example generates a small (5x4) X matrix of random values. Then a percentage of the X values (50% in this case) are randomly set to NA. The first matrix is the original, and the second matrix is imputed using the penalized matrix decomposition. Lambda penalties were set to ten in this case, as sparsity is not a desired feature.

```
> Xmiss

           [,1]      [,2]        [,3]        [,4]
[1,]  0.372412 -2.60326  0.3640716          NA
[2,]         NA        NA          NA 0.06710891
[3,]         NA   1.42029  1.2193184          NA
[4,]         NA        NA          NA 0.53054505
[5,] -0.828094        NA -1.2829601 0.02312756

> missing.test$X.rebuilt

            [,1]         [,2]        [,3]         [,4]
[1,]  0.3724120 -2.60325957  0.3640716  1.60432621
[2,] -1.1013169  0.01835828  0.5137207  0.06710891
[3,]  0.2876197  1.42029017  1.2193184 -1.47915011
[4,] -0.6904103 -2.35560414 -0.8330835  0.53054505
[5,] -0.8280940  0.15381885 -1.2829601  0.02312756
```

Example 2:

Decreasing the lambda penalties increases sparsity, effectively selecting a limited number of non-zero elements of U and V, leading to a reconstructed X matrix, where $X = U * D * t(V)$, with a limited number of columns containing non-zero elements.

This example uses another toy (5x4) X matrix, this time with no missing values. Lambda is varied. (Note that there are two lambda values, one for U and one for V. They may be set to the same value, or may be varied if different levels of sparisty are desired. Here, we use the same lambda for both parameters.)

This table shows the number of columns in X containing non-zero values. The number of non-zero columns decreases as lambda decreases, illustrating the variable selection effect of the lambda penalty.

```
     lambdas nonzero.x.cols
[1,]    2.00              4
[2,]    1.75              4
[3,]    1.50              4
[4,]    1.25              2
[5,]    1.00              1
```

The reconstructed $X = U * D * t(V)$ matrices are as follows, in decreasing lambda order.

```
            [,1]         [,2]        [,3]        [,4]
[1,]  0.13455597  0.07024089 -0.7914258  0.9565680
[2,] -0.04286561 -0.02237670  0.2521252 -0.3047346
[3,] -0.22841774 -0.11923859  1.3434981 -1.6238379
[4,]  0.08253913  0.04308706 -0.4854753  0.5867765
[5,] -0.13511929 -0.07053495  0.7947391 -0.9605726

            [,1]         [,2]        [,3]        [,4]
[1,]  0.09344614  0.05134998 -0.6222627  0.8668882
[2,]  0.00000000  0.00000000  0.0000000  0.0000000
[3,] -0.20536600 -0.11285155  1.3675430 -1.9051549
[4,]  0.04332265  0.02380641 -0.2884878  0.4018989
[5,] -0.08910263 -0.04896316  0.5933391 -0.8265940

            [,1]         [,2]        [,3]        [,4]
[1,] -0.01254320  0.07075051 -0.4572745  0.6198442
[2,]  0.00000000  0.00000000  0.0000000  0.0000000
[3,]  0.03956926 -0.22319237  1.4425363 -1.9553850
[4,]  0.00000000  0.00000000  0.0000000  0.0000000
[5,]  0.01324501 -0.07470915  0.4828600 -0.6545258

            [,1] [,2] [,3]        [,4]
[1,]  0.18618651    0    0  0.05732979
[2,]  0.30890519    0    0  0.09511682
[3,] -0.03356901    0    0 -0.01033643
[4,]  1.94765331    0    0  0.59971343
[5,]  0.00000000    0    0  0.00000000

          [,1] [,2] [,3] [,4]
[1,] 0.000000    0    0    0
[2,] 0.000000    0    0    0
[3,] 0.000000    0    0    0
[4,] 1.696754    0    0    0
[5,] 0.000000    0    0    0
```

4. Imputing Missing Data

Approach:

Before imputing the missing data, we address a concern that drives our chosen approach. The 1999 Hastie et al. paper notes that there is an issue with the SVD imputation method: "One concern with the SVD method is that it does a lot of borrowing strength from the bulk of the data." The penalized matrix decomposition method shares this issue. In short, if predictors vary in scale, imputed values for each predictor may not be in a reasonable range for that predictor. This is an issue in this data setting, where standardized testing and other scores have limited valid ranges. For example, SAT scores range in the hundreds, with a combined score generally between 1000 and 1400, while GPA-related metrics range from zero to four.

Hastie et al. suggest a nearest-neighbor approach as an antidote to the scaling problem, where missing values are imputed by combining nearest neighbor values based on a k-nearest-neighbors approach.

In the case of the calculus data, a clear solution is to break the predictors into four groups, and impute the data on each individual matrix before re-assembling into a full X matrix. There are four groups where predictors are clearly similar, if not identical, in scale, and the predictors in these groups are related, so it is intuitive to use their data together to impute missing values within each group.

Groups of categorical predictors were assembled as follows:

SAT: Contains verbal, quantitative, and total SAT score variables.
ACT: Contains english, math and composite ACT score variables.
GPA: Contains GPA-related variables, including highschool and estimated college values.
SCORE: Contains various standardized testing-related variables, all scored on a zero-to-100 scale.

Results:

After imputing the data, a few checks were performed to verify reasonableness.

First, histograms of the raw data for each continuous predictor, with the kernel density estimate of the predictor including imputed data overlaid in blue. It is not expected that these will align exactly; this is a check to ensure that imputing the data is not significantly changing the shape of the distribution of each predictor.

The histograms verify that the imputed data is not significantly altering the distribution of any of the predictors.

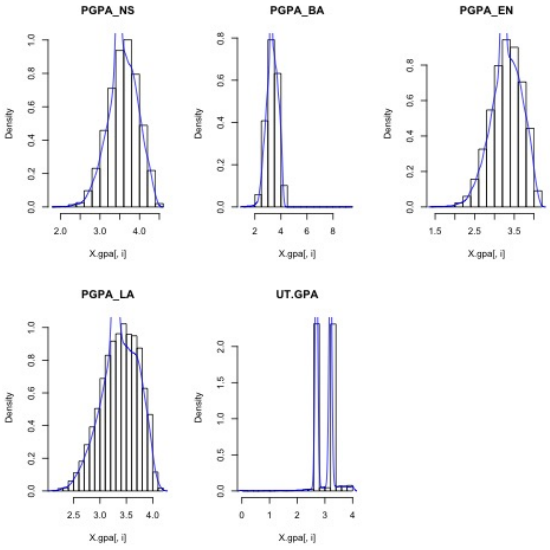

Figure 1: Histograms of imputed values for GPA-related predictors

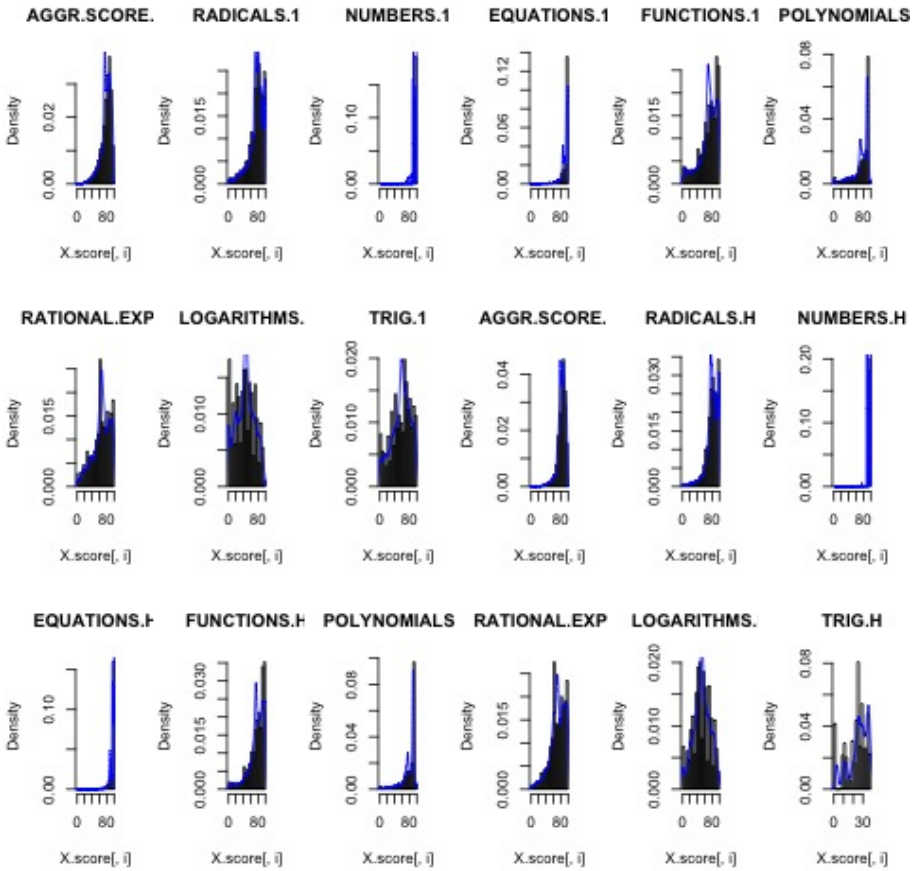

Figure 2: Histograms of imputed values for score-related predictors

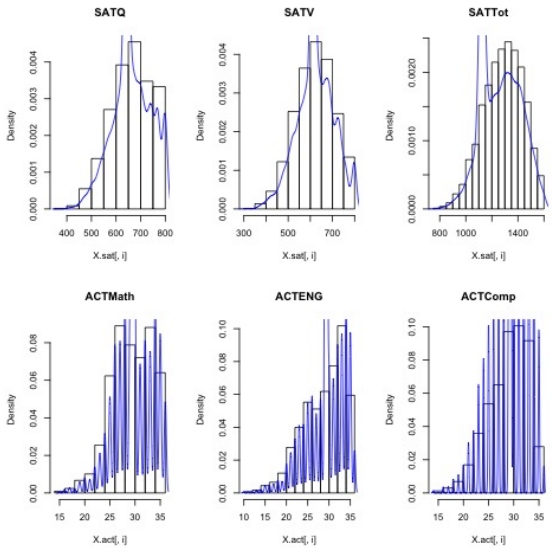

Figure 3: Histograms of imputed values for SAT and ACT-related predictors

Below is an example of checking the ranges, for the SAT and ACT predictors. These two groups illustrate the benefit in breaking up the predictors into several groups which share a scale.

```
Original SAT ranges:
     SATQ SATV SATTot
[1,]  370  310    770
[2,]  800  800   1600

Imputed SAT ranges:
          SATQ     SATV     SATTot
[1,] 369.8816 309.8385  769.7584
[2,] 799.7586 799.8298 1599.5605

Original ACT ranges:
     ACTMath ACTENG ACTComp
[1,]      15      10       14
[2,]      36      36       36

Imputed ACT ranges:
     ACTMath ACTENG ACTComp
[1,]      15      10       14
[2,]      36      36       36
```

Logistic Regression Model Improvements:

We split the data into a test (30%) and train set (70%). This split was performed for both the original data and the imputed data, yielding two training sets and two test sets.

A logistic regression was performed on the training data with missing values and fit to the test data with missing values. The test error was .78. Additionally, 2256 out of 3051 values in the test set were predicted as NA due to missing data. This is not a surprising result, given the high proportion (50%) of the data set values that are missing.

A second logistic regression was performed, this time using the training date with imputed values and fit to the test data with imputed values. The test error was .21, and of course no values were predicted as NA due to the presence of the imputed values.

Both of these models were fit with no other model selection techniques; both were fit to all continuous predictors. The imputed data results in a drastic improvement in test error, and the model is now ready to be improved in other ways such as variable selection.

5. Variable Selection

As illustrated in Example 2, the penalized matrix decomposition can also be used for variable selection. (Recall: Decreasing the lambda penalties increases sparsity, effectively selecting a limited number of non-zero elements of U and V, leading to a reconstructed X matrix, where $X = U * D * t(V)$, with a limited number of columns containing non-zero elements.)

The imputation of missing data and variable selection may be performed in one step, by imputing the missing data while using smaller lambda penalties than the ones selected in the previous section. The following table illustrates how the number of selected predictors decreases as the lambda penalty decreases.

	pred1	pred2	pred3	pred4
lambdas	"5"	"2.5"	"1.5"	"1"
	"PGPA_NS"	"PGPA_NS"	"AGGR.SCORE.H"	"ACTComp"
	"PGPA_BA"	"PGPA_EN"	"ACTENG"	NA
	"PGPA_EN"	"PGPA_LA"	"ACTComp"	NA
	"PGPA_LA"	"SATV"	NA	NA
	"AGGR.SCORE.1"	"SATTot"	NA	NA
	"AGGR.SCORE.H"	"ACTMath"	NA	NA
	"SATQ"	"ACTENG"	NA	NA
	"SATV"	"ACTComp"	NA	NA
	"SATTot"	NA	NA	NA
	"ACTMath"	NA	NA	NA
	"ACTENG"	NA	NA	NA
	"ACTComp"	NA	NA	NA

6. Appendix

Documentation Source

All project documentation and source code is available in the following github repository.

https://github.com/jstarling1/penalized-matrix-decomp

R Code: Main Analysis Code

```
#SDS 385 Final Project
#Jesse Miller & Jennifer Starling
#Fall 2016

rm(list=ls()) #Clean workspace.
library(stats)  #For KDE estimation in histograms.

setwd("/Users/jennstarling/UTAustin/2016_Fall_SDS 383C_Statistical Modeling 1/
    Final Project/penalized-matrix-decomp")

source(file="./R Code/Penalized_Matrix_Decomp_Functions.R") #Read in Penalized
    Matrix Decomp functions.

#-----------------------------------------------------------------
#DATA LOADING:

#Read in data.
data2014 = read.csv(file="./Data/2014FA_AllData.csv",header=T)
data2015 = read.csv(file="./Data/2015FA_AllData.csv",header=T)
data = rbind(data2014,data2015)

#Alternative: Load Rdata object directly.
load("./Data/data.Rdata")

#-----------------------------------------------------------------
#DATA PROCESSING:

#Create Y variable as Y=1 (pass), Y=0 (fail) based on Math Grade >= C.
passing.grades = c("A+","A","A-","B+","B","B-","C+","C","C-")
data$Y = ifelse(data$Math.Grade %in% passing.grades,1,ifelse(data$Math.Grade=="",
    NA,0))

#Restrict data set to only cases where Y != NA, ie where pass/fail known.
data = data[!is.na(data$Y),]

#Create a subset of just the calculus classes
data.calc = data[data$Math.Course %in% c("408C","408N","408K"),]

#Set up some initial dimensional information.
n = nrow(data)                          #Total number of obs for both years combined.
n2014 = sum(data$YEAR==2014)            #Total number of obs for 2014 only.
n2015 = nrow(data$YEAR==2015)           #Total number of obs for 2015 only.
p = ncol(data)-2                        #Number of predictors (excl ID and year cols).

#################################################################
###   TOY EXAMPLES OF PENALIZED MATRIX DECOMPOSITION:           ###
#################################################################
```

```
45
#Example 1: Illustrate missing data imputation.

#Set up X matrix.
X = matrix(rnorm(20),nrow=5,ncol=4)
50 n = nrow(X)
p = ncol(X)

#Randomly select values to set to NA.
n.elems = nrow(X) * ncol(X)
55 na.locs = sample(1:n.elems,size=n.elems*.5,replace=F)
Xmiss = X
Xmiss[na.locs] = NA

K=ncol(X)
60 lambdas = 10    #Want a large lambda, not trying to induce sparsity in features
    here.

missing.test = sparse.matrix.factorization.rankK(X,K,
                     lambdaU= lambdas,
                     lambdaV=lambdas,
65                   maxiter=20,tol=1E-6)

Xmiss
missing.test$X.rebuilt

70 #------------------------------------------------------------

#Example 2: A simulated matrix with no missing values.
#Illustrates how decreasing lambda penalty terms selects features.

75 X = matrix(rnorm(20),nrow=5,ncol=4)
n = nrow(X)
p = ncol(X)

#Paper notes that if you want u and v to be equally sparse, set a constant c,
80 #and let lambdaU = c*sqrt(n), and let lambdaV = c * sqrt(p)
c = 2
lambdaU = c*sqrt(n)
lambdaV = c*sqrt(p)

85 K = 1   #Set a K value for testing.  We'll use Rank 1 here.
#K=ncol(X)

lambdas = seq(2,1,by=-.25)  #Vector of lambdaU=lambdaV values.
tests = list()              #Empty vector for holding test cases.
90 nonzero.x.cols = rep(0,length(lambdas)) #Empty vector for holding sparsity info.

#Loop through test cases.
for (i in 1:length(lambdas)){
    tests[[i]] = sparse.matrix.factorization.rankK(X,K,
95                   lambdaU= lambdas[i],
                     lambdaV=lambdas[i],
                     maxiter=20,tol=1E-6)
    Xnew = tests[[i]]$X.rebuilt
    nonzero.x.cols[i] = nonzero.col.info(Xnew)$num.nonzero.cols
100 }

#Display results.
```

```
for (i in 1:length(tests)){
    print(tests[[i]]$X.rebuilt)
}

cbind(lambdas,nonzero.x.cols)



################################################################################
###    IMPUTING MISSING CONTINUOUS DATA USING PENALIZED MATRIX FACTORIZATION.  ###
################################################################################

#-------------------------------------------------------------------
#IMPUTING MISSING CONTINUOUS DATA USING PENALIZED MATRIX FACTORIZATION.

#-------------------------------------
#1. Identify continuous predictors to be included in the imputation.
head(data)

#cols.continuous.subset = c(13,14,15,16,20,30,39,40,41,42,43,44)

cols.continuous.all = c(13:16,20:28,30:38,39:44)
X = data.matrix(data[,cols.continuous.all])  #Subset of just the desired
    continuous data cols.

#Cols.continuous.all includes 21-28 and 31-38, which subset does not.
#These cols include the breakdowns of ALEKS-1 and ALEKS-H.

    #NOTE: This technique works well when predictors have same scale.
    #If predictors have vastly different scales, X must be scaled
    #first to ensure logical predictions.

    #Since we are working with a few different groups of predictors,
    #each which is related to a certain
    #type of test, it makes sense to work in groups.

#-------------------------------------
#2. Try imputing all missing values for continuous columns at once.

#Scale and center data.  (By hand, so can back-transform.)
col.means = colMeans(X,na.rm=T)
col.sdevs = sqrt(apply(X,2,function(a) var(a,na.rm=T)))
X.scaled = scale(X)

#Impute missing values for all continuous predictors at once.
pmd = sparse.matrix.factorization.rankK(X.scaled,K=ncol(X),lambdaU=1000,lambdaV
    =1000,maxiter=20)
X.filled.scaled = pmd$X.rebuilt

head(X.scaled)
head(X.filled.scaled)

#Reverse scaling.
X.filled = (X.filled.scaled * col.sdevs) + col.means
colnames(X.filled) = colnames(X)
head(X)
head(X.filled)
```

```
#Method fails, ended up with wrong scale on many variables.
#This is a common problem, as noted by Hastie et al (1999), as scale

#-------------------------------------
#3. Impute missing variables in four related groups.
#   These related groups are of similar scale.

#SAT VALUES:
X.sat = data.matrix(data[,39:41]) #Just the SAT variables.
pmd.sat = sparse.matrix.factorization.rankK(X.sat,K=ncol(X.sat),lambdaU=1000,
    lambdaV=1000,maxiter=20)
X.sat.filled = pmd.sat$X.rebuilt
colnames(X.sat.filled) = colnames(X.sat)

head(X.sat)
head(X.sat.filled)

#ACT VALUES:
X.act = data.matrix(data[,42:44]) #Just the SAT variables.
pmd.act = sparse.matrix.factorization.rankK(X.act,K=ncol(X.act),lambdaU=1000,
    lambdaV=1000,maxiter=20)
X.act.filled = pmd.act$X.rebuilt
colnames(X.act.filled) = colnames(X.act)

head(X.act)
head(X.act.filled)

#GPA VALUES:
X.gpa = data.matrix(data[,c(13:16,52)]) #Just the PGPA variables.
pmd.gpa = sparse.matrix.factorization.rankK(X.gpa,K=ncol(X.gpa),lambdaU=1000,
    lambdaV=1000,maxiter=20)
X.gpa.filled = pmd.gpa$X.rebuilt
colnames(X.gpa.filled) = colnames(X.gpa)

head(X.gpa)
head(X.gpa.filled)

#SCORE VALUES:
X.score = data.matrix(data[,c(20:28,30:38)]) #Just the PGPA variables.
pmd.score = sparse.matrix.factorization.rankK(X.score,K=ncol(X.score),lambdaU
    =1000,lambdaV=1000,maxiter=20)
X.score.filled = pmd.score$X.rebuilt
colnames(X.score.filled) = colnames(X.score)

head(X.score)
head(X.score.filled)

#-------------------------------------
#4. Reasonableness checks on imputed data:

#Histograms to compare distributions before and after imputing data.
#SATs & ACTs:
jpeg(file='/Users/jennstarling/UTAustin/2016_Fall_SDS 383C_Statistical Modeling 1/
    Final Project/LaTeX Files/SAT_ACT_hist.jpg')
par(mfrow=c(2,3))
for(i in 1:3){
    hist(X.sat[,i],freq=F,main=paste(colnames(X.sat)[i]))
    points(density(X.sat.filled[,i]),col='blue',type='l')
}
```

```
for(i in 1:3){
    hist(X.act[,i],freq=F,main=paste(colnames(X.act)[i]))
    points(density(X.act.filled[,i]),col='blue',type='l')
}
dev.off()

#GPA values:
jpeg(file='/Users/jennstarling/UTAustin/2016_Fall_SDS 383C_Statistical Modeling 1/
    Final Project/LaTeX Files/GPA_hist.jpg')
par(mfrow=c(2,3))
for(i in 1:5){
    hist(X.gpa[,i],freq=F,main=paste(colnames(X.gpa)[i]))
    points(density(X.gpa.filled[,i]),col='blue',type='l')
    main=paste(colnames(X.gpa[i]))
}
dev.off()

#SCORE values:
jpeg(file='/Users/jennstarling/UTAustin/2016_Fall_SDS 383C_Statistical Modeling 1/
    Final Project/LaTeX Files/SCORE_hist.jpg')
par(mfrow=c(3,6))
for(i in 1:18){
    hist(X.score[,i],freq=F,main=paste(colnames(X.score)[i]))
    points(density(X.score.filled[,i]),col='blue',type='l')
}
dev.off()

#-----------------------------------

#Sanity check ranges of the output for each group.
apply(X.sat, 2, function(x) range(x,na.rm=T))
apply(X.sat.filled, 2, function(x) range(x))

apply(X.act, 2, function(x) range(x,na.rm=T))
apply(X.act.filled, 2, function(x) range(x))

apply(X.gpa, 2, function(x) range(x,na.rm=T))
apply(X.gpa.filled, 2, function(x) range(x))

apply(X.score, 2, function(x) range(x,na.rm=T)) #Some neg vals here, ask Jesse.
apply(X.score.filled, 2, function(x) range(x))

#-----------------------------------
#5. Reconstruct data frame with imputed values.

# Construct an X version with just continuous vars,
# and construct a Data version with all vars.
#Note: Cols not in same order as in original data set.
X.cont.new = cbind(X.gpa.filled[,1:3], X.score.filled, X.sat.filled, X.act.filled,
     X.gpa.filled[,4])
colnames(X.cont.new) = colnames(X)
data.cont.new = cbind(X.cont.new,Y=data$Y)  #Save new imputed data.
data.cont.old = cbind(X,Y=data$Y)           #Save old data (just continuous
    predictors).

#Optional: Output the new imputed data as an R object.
#save(data.cont.new,file="./Data/data_continuous_imputed.Rdata")

#Preview reconstructed data.
```

```
head(data.cont.old)
head(round(data.cont.new,3))

#-----------------------------------

#Compare the results of a logistic regression before and after data imputation.
#This quick check involves holding out 30% of the data, and obtaining
#a 'test error' for the held out 30%.  This is performed for the continuous
    variables
#only, with and without imputed data.

train.rows = sample(nrow(X),nrow(X)*.7,replace=F)

train.missing = data.cont.old[train.rows,]
test.missing = data.cont.old[-train.rows,]

train.filled = data.cont.new[train.rows,]
test.filled = data.cont.new[-train.rows,]

#-----------------------------------

#Test error for data with missing values.
lm.with.missing = glm(Y ~ ., family=binomial,data=as.data.frame(train.missing))
pred.missing = predict.glm(lm.with.missing,newdata=as.data.frame(test.missing
    [,-29]),type='response')
yhat.missing = ifelse(pred.missing >= .5,1,0)

yhat.temp = yhat.missing    #To handle values that are predicted as NA due to
    missing data.
yhat.temp[is.na(yhat.missing)] = 999
test.err.missing = sum(test.missing[,29] != yhat.temp) / length(yhat.filled)
test.err.missing

paste(sum(is.na(yhat.missing)),'out of ',length(yhat.filled),' values predicted as
     NA due to missing data.')

#-----------------------------------
#Test error for data with imputed values.
lm.filled = glm(Y ~ ., family=binomial,data=as.data.frame(train.filled))
pred.filled = predict.glm(lm.filled,newdata=as.data.frame(test.filled[,-29]),type=
    'response')
yhat.filled = ifelse(pred.filled >= .5,1,0)

test.err.filled = sum(test.filled[,29] != yhat.filled) / length(yhat.filled)
test.err.filled

#Conclusion: Imputing the missing data using penalized matrix decomposition
    drastically
#decreased the logistic regression test error.

#A few things:
#No additional model fitting or analysis has been done.  Model could of course be
    improved in many ways.
#Could look at using this functionality for variable selection, as well.

#####################################################################
###   VARIABLE SELECTION USING PENALIZED MATRIX FACTORIZATION.  ###
#####################################################################

```

```
#The following is an example of how decreasing the lambda penalty can
#perform variable selection on the continuous variables.

#In this case, we are not worried about the scale of the values, so we
#will analyze all continuous predictors together.

cols.continuous.subset = c(13,14,15,16,20,30,39,40,41,42,43,44)
X = data.matrix(data[,cols.continuous.subset])  #Subset of just the desired
    continuous data cols.
X = scale(X)

#Impute missing values for all continuous predictors at once.  Vary the values of
    lambda.
#Will use a rank K factorization, so that we can get down to a single selected
    predictor.

test = list()
lambdas = c(5,2.5,1.5,1)
interesting.predictors = list() #Empty list for holding interesting predictors for
     each lambda.
num.nonzero.x.cols = rep(0,length(lambdas)) #Empty vector for holding number of
    interesting predictors.

#Loop through test cases.
for (i in 1:length(lambdas)){
    tests[[i]] = sparse.matrix.factorization.rankK(X,K=1,
                     lambdaU= lambdas[i],
                     lambdaV=lambdas[i],
                     maxiter=20,tol=1E-6)

    Xnew = tests[[i]]$X.rebuilt

    Xnew.col.info = nonzero.col.info(Xnew)

    nonzero.x.cols[i] = Xnew.col.info$num.nonzero.cols
    interesting.predictors[[i]] = colnames(X)[Xnew.col.info$nonzero.cols.idx]
}

#Display results.
pred1 = unlist(interesting.predictors[[1]])
pred2 = unlist(interesting.predictors[[2]])
pred3 = unlist(interesting.predictors[[3]])
pred4 = unlist(interesting.predictors[[4]])

#Fill in all unused values with NA, so can cbind results for easy viewing.
length(pred2) = length(pred3) = length(pred4) = length(pred1)

output = cbind(pred1,pred2,pred3,pred4)
output = rbind(lambdas=lambdas[1:4],output)
output #View results

#-----------------------------------------
#Try another logistic regression model with one of the variable selection results.

#The pred2 list looks promising; fit this to a training set, then predict to a
    test set,
#using the imputed data.
pred = pred2
selected.cols = c(pred[!is.na(pred)],"Y") #Must include y.
```

```
train.filled.2 = data.cont.new[train.rows, selected.cols]
test.filled.2 = data.cont.new[-train.rows, selected.cols]

#Test error for data with imputed values.
lm.filled.2 = glm(Y ~ ., family=binomial,data=as.data.frame(train.filled.2))
pred.filled.2 = predict.glm(lm.filled.2,newdata=as.data.frame(test.filled.2[,-29])
    ,type='response')
yhat.filled.2 = ifelse(pred.filled.2 >= .5,1,0)

test.err.filled.2 = sum(test.filled.2[,length(selected.cols)] != yhat.filled.2) /
    length(yhat.filled.2)
test.err.filled.2
```

R Code: Penalized Matrix Decomposition Functions

```
#Penalized Matrix Decomposition
#November 10 ,2016
#Jennifer Starling

#References paper:
# Witten, Tibshirani, Hastie.  2009. A penalized matrix decomposition...

#################################
###   REQUIRED FUNCTIONS:     ###
#################################

#l1 penalty; Soft thresholding operator.
soft <- function(x,theta){
  return(sign(x)*pmax(0, abs(x)-theta))
}

#l1 norm of a vector.
l1norm <- function(vec){
  a <- sum(abs(vec))
  return(a)
}

#l2 norm of a vector.
l2norm <- function(vec){
  a <- sqrt(sum(vec^2))
  return(a)
}

#Binary search function
#(Source: Witten, Hastie & Tibshirani R package PMA: https://cran.r-project.org/
    web/packages/PMA/)
#(For finding theta for soft-thresholding for each iteration)
BinarySearch <- function(argu,sumabs){

  #Define functions for the l2 norm and the soft thresholding operator.
  l2n = function(vec) {return(sqrt(sum(vec^2)))}
  soft = function(x,theta) { return(sign(x)*pmax(0, abs(x)-theta))}

  if(l2n(argu)==0 || sum(abs(argu/l2n(argu)))<=sumabs) return(0)
  lam1 <- 0
  lam2 <- max(abs(argu))-1e-5
```

```
  iter <- 1
  while(iter < 150){
    su <- soft(argu,(lam1+lam2)/2)
    if(sum(abs(su/l2n(su)))<sumabs){
      lam2 <- (lam1+lam2)/2
    } else {
      lam1 <- (lam1+lam2)/2
    }
    if((lam2-lam1)<1e-6) return((lam1+lam2)/2)
    iter <- iter+1
  }
  warning("Didn't quite converge")
  return((lam1+lam2)/2)
}

#########################################################
###   PENALIZED MATRIX DECOMPOSITION FUNCTIONS:     ###
#########################################################

#-----------------------------------------------------------------------------
#SPARSE MATRIX FACTORIZATION RANK 1 FUNCTION (Penalized Matrix Decomposition)
#For a single factor, ie K=1 (rank-1 approximation to original matrix)
#Inputs:
#   X = matrix to be factorized
#   v = initial v vector.
#   lambdaU = the u penalty (c1)
#   lambdaV = the v penalty (c2)
#      *If lambda1 = lambda2 = 0, function returns the non-sparse Rank 1 SVD of X.
#   maxiter = maximum number of iterations allowed
#   tol = tolerance level for convergence check
#Output: List object, including the following:
#   X.rebuilt = sparse matrix factorization of X; X = U * D * t(V)
#   U, D, V = the decomposed elements of X, where X = U * D * t(V)

sp.matrix.decomp.rank1 = function(X,v=NULL,lambdaU=1, lambdaV=1, maxiter=20, tol=1
    E-6){

    #Define functions for the l2 norm and the soft thresholding operator.
    l2norm = function(vec) {return(sqrt(sum(vec^2)))}
    soft = function(x,theta) { return(sign(x)*pmax(0, abs(x)-theta))}

    #1. Housekeeping parameters.
    i=1                 #Initialize iterator.
    converged <- 0      #Indicator for whether convergence met.
    p = ncol(X)         #Number of columns of X matrix.

    #2. Initializations
    v.old = rnorm(p)    #Initialize v.old to a random vector. (To get iterations
        started.)

    if(is.null(v)){
        v = rep(sqrt(1/p),p) #Initialize v to meet constraint l2norm(v) = 1.
    }

    #3. NA Handling:
    nas <- is.na(X)
    if(sum(nas)>0) X[nas] = mean(X[!nas])
```

```
    #4. Iterate until convergence.
    for (i in 1:maxiter){

        #1. Update u.

        #First, calculate theta for sign function.
        u.arg = X %*% v      #Argument to go into sign function: Xv
        u.theta = BinarySearch(u.arg,lambdaU)

        #Second, update u.
        u = matrix( soft(u.arg,u.theta) / l2norm(soft(u.arg,u.theta)), ncol=1)

        #---------------------------
        #2. Update v.

        #First, calculate theta for sign function.
        v.arg = t(X) %*% u
        v.theta = BinarySearch(v.arg,lambdaV)

        #Second, update v.
        v = matrix( soft(v.arg,v.theta) / l2norm(soft(v.arg,v.theta)), ncol=1)

        #---------------------------
        #3. Convergence check steps.

        #Exit loop if converged.
        if(sum(abs(v.old - v)) < tol){
            converged=1
            break
        }

        #If not converged, update v.old for next iteration.
        v.old = v
    }

    #Set d value.
    d = as.numeric(t(u) %*% (X %*% v))

    #Reconstruct sparse X matrix.
    Xsp = d * tcrossprod(u,v)

    #Return function results.
    return(list(Xsp=Xsp,u=u,d=d,v=v,lambdaU=lambdaU,lambdaV=lambdaV,converged=
        converged,iter=i))
}
#-------------------------------------------------------------------------

#SPARSE MATRIX FACTORIZATION RANK K FUNCTION (Penalized Matrix Decomposition)
#For a Rank K approximation of X.
#Inputs:
#   X = matrix to be factorized
#   K = rank of factorization.  Must be <= ncol(X)
#   lambdaU = the u penalty (c1)
#   lambdaV = the v penalty (c2)
#      *If lambda1 = lambda2 = 0, function returns the non-sparse Rank K SVD of X.
#   maxiter = maximum number of iterations allowed
#   tol = tolerance level for convergence check
#Output: List object, including the following:
#   Xsp = sparse matrix factorization of X.
```

```
#   U, D, V = the decomposed elements of X, where X = U * D * t(V)

sparse.matrix.factorization.rankK = function(X,K=2,lambdaU=1, lambdaV=1, maxiter
    =20, tol=1E-6){

    #1. Housekeeping parameters.
    i=1                 #Initialize iterator.
    converged <- 0      #Indicator for whether convergence met.

    #2. NA handling for imputing missing obs.
    nas = is.na(X)  #Identify locations of NA values in matrix X.
    X.use = X       #Initialize X values to whole matrix.

    #2. Initializations
    D = numeric(K)  #Initialize K-length vector to hold D values.
    U = matrix(0,nrow=nrow(X),ncol=K)   #Initialize matrix to hold U values.
    V = matrix(0,nrow=ncol(X),ncol=K)   #Initialize matrix to hold V values.

    #Initialize v to be a nxK matrix, with each column having an l2 norm = 1.
    p = ncol(X)
    v = rep(sqrt(1/p),p)
    v = matrix(rep(v,each=K), ncol=K, byrow=T)

    #3. Iterate through Rank 1 recursion to obtain Rank K factorization.
    for(k in 1:K){
        #Run Rank 1 factorization.
        output = sp.matrix.decomp.rank1(X.use,v=matrix(v[,k],ncol=1),lambdaU,
            lambdaV, maxiter=20, tol)

        #Assign values.
        U[,k] = output$u
        V[,k] = output$v
        D[k] = output$d

        Xnew = X.use - output$d * tcrossprod(output$u, output$v)
        X.use[!nas] = Xnew[!nas]
    }

    #Reconstruct the X matrix as U * D * t(V).
    if (K==1) { X.rebuilt = as.numeric(D) * U %*% t(V) }
    if (K > 1){ X.rebuilt = U %*% diag(D) %*% t(V) }

    #Return function outputs.
    return(list(U=U,V=V,D=D,X.rebuilt = X.rebuilt))
}

################################
###  OPTIONAL FUNCTIONS:     ###
################################

#NOTE: THis function is not used by any of the above functions.
#It is just a handy tool that makes doing analysis on feature selection easier.

#Function that checks how many columns have nonzero values, and returns list of
    these columns.
#Inputs: A matrix.
#Outputs:
#   nonzero.cols = Vector of 1/0 indicators for whether each column has nonzero
    values.
```

```
#   nonzero.cols.idx = column index numbers for cols with nonzero values.
#   num.nonzero.cols = Count of the nonzero columns in a matrix.
nonzero.col.info = function(mat){
    nonzero.cols = apply(mat,2,function(c) ifelse(sum(c!=0,na.rm=T)>0,1,0))
    nonzero.cols.idx = which(nonzero.cols==1)
    num.nonzero.cols = length(nonzero.cols.idx)

    return(list(nonzero.cols = nonzero.cols, nonzero.cols.idx=nonzero.cols.idx,
        num.nonzero.cols = num.nonzero.cols))
}
```