

Biostatistics 615/815 Lecture 14: Statistical Computing with Matrix

Hyun Min Kang

October 25th, 2012

Linear Regression

Linear model

- $\mathbf{y} = X\beta + \epsilon$, where X is $n \times p$ matrix
- Under normality assumption, $y_i \sim N(X_i\beta, \sigma^2)$.

Key inferences under linear model

- Effect size : $\hat{\beta} = (X^T X)^{-1} X^T \mathbf{y}$
- Residual variance : $\hat{\sigma}^2 = (\mathbf{y} - X\hat{\beta})^T (\mathbf{y} - X\hat{\beta}) / (n - p)$
- Variance/SE of $\hat{\beta}$: $\hat{\text{Var}}(\hat{\beta}) = \hat{\sigma}^2 (X^T X)^{-1}$
- p-value for testing $H_0 : \beta_i = 0$ or $H_o : R\beta = 0$.

Using R to solve linear model

```
> y <- rnorm(100)
> x <- rnorm(100)
> summary(lm(y~x))
```

```
Call:
lm(formula = y ~ x)
```

```
Residuals:
    Min       1Q   Median       3Q      Max
-2.15759 -0.69613  0.08565  0.70014  2.62488
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.02722    0.10541   0.258   0.797
x            -0.18369    0.10559  -1.740   0.085 .
---
Signif. codes:  ...
```

```
Residual standard error: 1.05 on 98 degrees of freedom
Multiple R-squared: 0.02996, Adjusted R-squared: 0.02006
F-statistic: 3.027 on 1 and 98 DF, p-value: 0.08505
```

Dealing with large data with 1m

```
> y <- rnorm(5000000)
> x <- rnorm(5000000)
> system.time(print(summary(lm(y~x))))
```

```
Call:
lm(formula = y ~ x)
```

```
Residuals:
    Min       1Q   Median       3Q      Max
-5.1310 -0.6746  0.0004  0.6747  5.0860
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.0005130  0.0004473  -1.147   0.251
x            0.0002359  0.0004473   0.527   0.598
```

```
Residual standard error: 1 on 4999998 degrees of freedom
Multiple R-squared: 5.564e-08, Adjusted R-squared: -1.444e-07
F-statistic: 0.2782 on 1 and 4999998 DF, p-value: 0.5979
```

```
user system elapsed
57.434 14.229 100.607
```

A case for simple linear regression

A simpler model

- $\mathbf{y} = \beta_0 + \mathbf{x}\beta_1 + \epsilon$
- $X = [1 \ \mathbf{x}], \beta = [\beta_0 \ \beta_1]^T$.

Question of interest

Can we leverage this simplicity to make a faster inference?

A faster inference with simple linear model

Ingredients for simplification

- $\sigma_y^2 = (\mathbf{y} - \bar{y})^T(\mathbf{y} - \bar{y}) / (n - 1)$
- $\sigma_x^2 = (\mathbf{x} - \bar{x})^T(\mathbf{x} - \bar{x}) / (n - 1)$
- $\sigma_{xy} = (\mathbf{x} - \bar{x})^T(\mathbf{y} - \bar{y}) / (n - 1)$
- $\rho_{xy} = \sigma_{xy} / \sqrt{\sigma_x^2 \sigma_y^2}$.

Making faster inferences

- $\hat{\beta}_1 = \rho_{xy} \sqrt{\sigma_y^2 / \sigma_x^2}$
- $SE(\hat{\beta}_1) = \sqrt{\sigma_y^2 (1 - \rho_{xy}^2) / (n - 2) \sigma_x^2}$
- $t = \rho_{xy} \sqrt{(n - 2) / (1 - \rho_{xy}^2)}$ follows t-distribution with d.f. $n - 2$

A faster R implementation

```
# note that this is an R function, not C++
fastSimpleLinearRegression <- function(y, x) {
  y <- y - mean(y)
  x <- x - mean(x)
  n <- length(y)
  stopifnot(length(x) == n) # for error handling
  s2y <- sum( y * y ) / ( n - 1 ) # \sigma_y^2
  s2x <- sum( x * x ) / ( n - 1 ) # \sigma_x^2
  sxy <- sum( x * y ) / ( n - 1 ) # \sigma_xy
  rxy <- sxy / sqrt( s2y * s2x ) # \rho_xy
  b <- rxy * sqrt( s2y / s2x )
  se.b <- sqrt( s2y * ( 1 - rxy * rxy ) / (n-2) / s2x )
  tstat <- rxy * sqrt( ( n - 2 ) / ( 1 - rxy * rxy ) )
  p <- pt( abs(tstat) , n - 2 , lower.tail=FALSE ) * 2
  return(list( beta = b , se.beta = se.b , t.stat = tstat, p.value = p ))
}
```

Now it became must faster

```
> system.time(print(fastSimpleLinearRegression(y,x)))
$beta
[1] 0.0002358472

$se.beta
[1] 0.0004473

$t.stat
[1] 0.5274646

$p.value
[1] 0.597871

   user  system elapsed
0.382  1.849   3.042
```

Dealing with even larger data

Problem

- Supposed that we now have 5 billion input data points
- The issue is how to load the data
- Storing 10 billion double will require 80GB or larger memory

What we want

- As fast performance as before
- But do not store all the data into memory
- R cannot be the solution in such cases - use C++ instead

Streaming the inputs to extract sufficient statistics

Sufficient statistics for simple linear regression

- 1 n
- 2 $\sigma_x^2 = \hat{\text{Var}}(x) = (\mathbf{x} - \bar{x})^T(\mathbf{x} - \bar{x}) / (n - 1)$
- 3 $\sigma_y^2 = \hat{\text{Var}}(y) = (\mathbf{y} - \bar{y})^T(\mathbf{y} - \bar{y}) / (n - 1)$
- 4 $\sigma_{xy} = \hat{\text{Cov}}(x, y) = (\mathbf{x} - \bar{x})^T(\mathbf{y} - \bar{y}) / (n - 1)$

Extracting sufficient statistics from stream

- $\sum_{i=1}^n x = n\bar{x}$
- $\sum_{i=1}^n y = n\bar{y}$
- $\sum_{i=1}^n x^2 = \sigma_x^2(n - 1) + n\bar{x}^2$
- $\sum_{i=1}^n y^2 = \sigma_y^2(n - 1) + n\bar{y}^2$
- $\sum_{i=1}^n xy = \sigma_{xy}(n - 1) + n\bar{x}\bar{y}$

Implementation : Streamed simple linear regression

```
#include <iostream>
#include <fstream>
#include <boost/math/distributions/students_t.hpp>
using namespace boost::math; // for calculating p-values from t-statistic
int main(int argc, char** argv) {
    std::ifstream ifs(argv[1]); // read file from the file arguments
    double x, y; // temporay values to store the input
    double sumx = 0, sumsqx = 0, sumy = 0, sumsqy = 0, sumxy = 0;
    int n = 0;

    // extract a set of sufficient statistics
    while( ifs >> y >> x ) { // assuming each input line feeds y and x
        sumx += x;
        sumy += y;
        sumxy += (x*y);
        sumsqx += (x*x);
        sumsqy += (y*y);
        ++n;
    }
}
```

Streamed simple linear regression (cont'd)

```
// convert the set of sufficient statistics to
double s2y = (sumsqy - sumy*sumy/n)/(n-1); // s2y = \sigma_y^2
double s2x = (sumsqx - sumx*sumx/n)/(n-1); // s2x = \sigma_x^2
double sxy = (sumxy - sumx*sumy/n)/(n-1); // sxy = \sigma_{xy}
double rxy = sxy/(s2x*s2y); // rxy = cor(x,y)

// calculate beta, SE(beta), and p-values
double beta = rxy * s2y / s2x;
double seBeta = sqrt( s2y / s2x * ( 1 - rxy*rxy ) / (n-2) );
double t = rxy * sqrt( (n-2)/(1-rxy*rxy) ); // t-statistics

students_t dist(n-2); // use student's t-distribution to compute p-value
double pvalue = 2.0*cdf(complement(dist, t > 0 ? t : (0-t) ));
```

Streamed simple linear regression (cont'd)

```
std::cout << "Number of observations   = " << n << std::endl;
std::cout << "Effect size     - beta    = " << beta << std::endl;
std::cout << "Standard error - SE(beta) = " << seBeta << std::endl;
std::cout << "Student's-t statistic = " << t << std::endl;
std::cout << "Two-sided p-value     = " << pvalue << std::endl;
return 0;
}
```

Summary - Simple Linear Regression

- A linear regression with one predictor and intercept
- `lm()` function in R may be computationally slow for large input
- Faster inference is possible by computing a set of summary statistics in linear time
- Streaming via C++ programming further resolves the memory overhead
- The idea can be applied in more sophisticated, large-scale analyses.

Multiple regression - a general form of linear regression

Recap - Linear model

- $\mathbf{y} = X\beta + \epsilon$, where X is $n \times p$ matrix
- Under normality assumption, $y_i \sim N(X_i\beta, \sigma^2)$.

Key inferences under linear model

- Effect size : $\hat{\beta} = (X^T X)^{-1} X^T \mathbf{y}$
- Residual variance : $\hat{\sigma}^2 = (\mathbf{y} - X\hat{\beta})^T (\mathbf{y} - X\hat{\beta}) / (n - p)$
- Variance/SE of $\hat{\beta}$: $\hat{\text{Var}}(\hat{\beta}) = \hat{\sigma}^2 (X^T X)^{-1}$
- p-value for testing $H_0 : \beta_i = 0$ or $H_o : R\beta = 0$.

Using `lm()` function in R

```
> y <- rnorm(1000)
> X <- matrix(rnorm(5000),1000,5)
> summary(lm(y~X))
.....
```

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	0.010934	0.031597	0.346	0.729
X1	0.026340	0.031886	0.826	0.409
X2	-0.025339	0.031789	-0.797	0.426
X3	-0.036607	0.031739	-1.153	0.249
X4	-0.002549	0.031467	-0.081	0.935
X5	0.050064	0.031665	1.581	0.114

Residual standard error: 0.9952 on 994 degrees of freedom
 Multiple R-squared: 0.004966, Adjusted R-squared: -3.948e-05
 F-statistic: 0.9921 on 5 and 994 DF, p-value: 0.4213

Implementing in C++ : Using SVD for increasing reliability

$$\begin{aligned}
 X &= UDV^T \\
 \hat{\beta} &= (X^T X)^{-1} X^T \mathbf{y} \\
 &= (VDU^T UDV^T)^{-1} VDU^T \mathbf{y} \\
 &= (VD^2 V^T)^{-1} VDU^T \mathbf{y} \\
 &= VD^{-2} V^T VDU^T \mathbf{y} \\
 &= VD^{-1} U^T \mathbf{y} \\
 \widehat{\text{Cov}}(\hat{\beta}) &= \hat{\sigma}^2 (X^T X)^{-1} \\
 &= \hat{\sigma}^2 (VD^{-2} V^T) \\
 &= \frac{(\mathbf{y} - X\hat{\beta})^T (\mathbf{y} - X\hat{\beta})}{n - p} (VD^{-1} (VD^{-1})^T)
 \end{aligned}$$

Using Eigen library to implement multiple regression

```

#include "Matrix615.h" // The class is posted at the web page
                        // mainly for reading matrix from file
#include <iostream>
#include <Eigen/Core>
#include <Eigen/SVD>

using namespace Eigen;

int main(int argc, char** argv) {
    Matrix615<double> tmpy(argv[1]); // read n * 1 matrix y
    Matrix615<double> tmpX(argv[2]); // read n * p matrix X
    int n = tmpX.rowNums();
    int p = tmpX.colNums();

    MatrixXd y, X;
    tmpy.cloneToEigen(y); // copy the matrices into Eigen::Matrix objects
    tmpX.cloneToEigen(X); // copy the matrices into Eigen::Matrix objects
}

```

Implementing multiple regression (cont'd)

```

JacobiSVD<MatrixXd> svd(X, ComputeThinU | ComputeThinV); // compute SVD
MatrixXd betasSvd = svd.solve(y); // solve linear model for computing beta
// calculate VD^{-1}
MatrixXd ViD= svd.matrixV() * svd.singularValues().asDiagonal().inverse();
double sigmaSvd = (y - X * betasSvd).squaredNorm()/(n-p); // compute \sigma^2
MatrixXd varBetasSvd = sigmaSvd * ViD * ViD.transpose(); // Cov(\hat{beta})

// formatting the display of matrix.
IOFormat CleanFmt(8, 0, " ", " ", "\n", "[", "]");

// print \hat{beta}
std::cout << "----- beta -----\n" << betasSvd.format(CleanFmt) << std::endl;
// print SE(\hat{beta}) -- diagonals os Cov(\hat{beta})
std::cout << "----- SE(beta) -----\n"
    << varBetasSvd.diagonal().array().sqrt().format(CleanFmt) << std::endl;
return 0;
}

```

Working examples with $n = 1,000,000, p = 6$

Using R and lm() routines

```

> system.time(y <- read.table('y.txt'))
  user system elapsed
 4.249  0.079  4.345
> system.time(X <- read.table('X.txt'))
  user system elapsed
62.013  0.658 62.314
> system.time(summary(lm(y~X)))
  user system elapsed
 5.849  1.228  7.703

```

Using C++ implementations

```

Elapsed time for matrix reading is 23.802
Elapsed time for computation is 1.19252

```

Alternative implementations : speed-reliability tradeoffs

Decomposition	Method	Requirements on the matrix	Speed	Accuracy
PartialPivLU	partialPivLu()	Invertible	++	+
FullPivLU	fullPivLu()	None	-	+++
HouseholderQR	householderQr()	None	++	+
ColPivHouseholderQR	colPivHouseholderQr()	None	+	++
FullPivHouseholderQR	fullPivHouseholderQr()	None	-	+++
LLT	llt()	Positive definite	+++	+
LDLT	ldlt()	Positive or negative semidefinite	+++	++

Summary - Multiple regression

- Multiple predictor variables, and a single response variable.
- A reliable C++ implementation of linear model inference using SVD
- Eigen library provides a convenient and reasonably fast way to implement sophisticated matrix operations in C++
- C++ implementations may have advantages in both speed and memory in large-scale data analyses.

Bulk Linear Regression

- Given
 - X : $m \times n$ matrix
 - Y : $g \times n$ matrix
- Want
 - P : $g \times m$ matrix of p-values between every pair of rows in X and Y using simple linear regression

A Naive R Implementation

```
naiveBulkTest <- function(X, Y) {
  n <- ncol(X)
  m <- nrow(X)
  g <- nrow(Y)
  stopifnot(n == ncol(Y))

  P <- matrix(nrow=g, ncol=m)
  for(i in 1:g) {
    for(j in 1:m) {
      P[i,j] <- summary(lm(Y[i,]~X[j,]))$coefficients[2,4] ## obtain p-value
    }
  }
  return(P)
}
```

A Naive R Implementation : Results

```
> source('naiveBulkTest.R')
> Y <- matrix(rnorm(200*100),200,100)
> X <- matrix(rnorm(200*100),200,100)
> system.time(Ps <- naiveBulkTest(X,Y))
user system elapsed
124.947 0.573 131.781
```

- Takes 2 minutes for 40,000 tests
- For 1000×1000 test with $n = 100$, it will take 33 hours

A Faster R Implementation

```
standardize <- function(X) { ## standardize each row
  r <- nrow(X); c <- ncol(X);
  mu <- rowMeans(X)
  s <- apply(X,1,sd)
  X <- (X-matrix(mu,r,c,byrow=FALSE))/matrix(s,r,c,byrow=FALSE);
  return (X);
}

fastBulkTest <- function(X, Y) {
  n <- ncol(X); m <- nrow(X); g <- nrow(Y)
  stopifnot(n == ncol(Y))

  X <- standardize(X);
  Y <- standardize(Y);
  T <- tcrossprod(X,Y)/n ## X %*% t(Y)
  T <- T * sqrt((n-2)/(1-T^2))
  P <- 2*pt(abs(T),n-2,lower.tail=FALSE)
  return(P);
}
```

A Faster R Implementation : Results

```
> source('fastBulkTest.R')
> Y <- matrix(rnorm(200*100),200,100)
> X <- matrix(rnorm(200*100),200,100)
> system.time(Pf <- fastBulkTest(X,Y))
user system elapsed
0.061 0.004 0.064
> X <- matrix(rnorm(1000*100),1000,100)
> Y <- matrix(rnorm(1000*100),1000,100)
> system.time(Pf <- fastBulkTest(X,Y))
user system elapsed
0.843 0.058 0.869
```

- Much faster!
- Matrix input/output takes longer than computation if load from a file

Implementing Bulk Linear Test in C++

Steps

- 1 Read input data from files using Matrix615.h class
- 2 Copy Matrix615 object to MatrixXd objects
- 3 Standardize each row of X and Y (mean 0, variance 1)
- 4 Calculate pairwise correlation matrix $R = XY^T$
- 5 $T = R / \sqrt{(n-2)/(1-R^2)}$
- 6 Compute p-values for each element of T .
- 7 Print the output

Copying Matrix615 to MatrixXd objects

```
void Matrix615::cloneToEigen(Eigen::Matrix<T, Eigen::Dynamic, Eigen::Dynamic>& m)
{
    int nr = rowNums();
    int nc = colNums();
    m.resize(nr, nc);
    for(int i=0; i < nr; ++i) {
        for(int j=0; j < nc; ++j) {
            m(i, j) = data[i][j];
        }
    }
}
```

Standardizing each row (or column) of a matrix

```
class EigenHelper {
public:
    // ....
    static void standardize(MatrixXd& m, bool rowwise = true) {
        if ( rowwise ) { // standardize each row
            VectorXd cmean = m.rowwise().mean(); // rowwise mean
            VectorXd sqsum = m.rowwise().squaredNorm();
            VectorXd csd = (sqsum.array()/m.cols() - cmean.array().square())
                .sqrt().inverse().matrix(); // rowwise stdev
            m.colwise() -= cmean; // make each row has zero mean
            vectorWiseProd(m, csd, rowwise); // make each row has unit variance
        }
        else { // standardize each column
            VectorXd cmean = m.colwise().mean();
            VectorXd sqsum = m.colwise().squaredNorm();
            VectorXd csd = (sqsum.array()/m.rows() - cmean.array().square())
                .sqrt().inverse().matrix();
            m.rowwise() -= cmean;
            vectorWiseProd(m, csd, rowwise);
        }
    }
};
```

Multiplying each row (or column) of a matrix

```
class EigenHelper {
public:
    static void vectorWiseProd(MatrixXd& m, VectorXd& v, bool rowwise = true) {
        int nv = v.size();
        int nr = m.rows();
        int nc = m.cols();
        if ( rowwise == false ) { // m[i,] *= v (component-wise)
            for(int i=0; i < nr; ++i) {
                for(int j=0; j < nc; ++j) {
                    m(i, j) *= v(j); // multiply v for each row
                }
            }
        }
        else {
            for(int i=0; i < nr; ++i) {
                for(int j=0; j < nc; ++j) {
                    m(i, j) *= v(i); // multiply v for each column
                }
            }
        }
    }
};
// ...
```

Running Bulk Linear Tests

```
int main(int argc, char** argv) {
    if ( argc != 3 ) {
        std::cerr << "Usage : " << argv[0] << " [Y] [X]" << std::endl;
        return -1;
    }

    // read input files to Matrix615 objects
    Matrix615<double> fY(argv[1]); // read g * n matrix
    Matrix615<double> fX(argv[2]); // read m * n matrix
    if ( fY.colNums() != fX.colNums() ) {
        std::cerr << "ERROR: The number of columns are discordant between "
            << "the two matrices" << std::endl;
        return -1;
    }

    int g = fY.rowNums();
    int n = fY.colNums();
    int m = fX.rowNums();
}
```

Running Bulk Linear Tests

```
// copy Matrix615 objects to Eigen objects
MatrixXd mX, mY;
fX.cloneToEigen(mX);
fY.cloneToEigen(mY);

// normalize X and Y matrix
EigenHelper::standardize(mX,true);
EigenHelper::standardize(mY,true);

ArrayXXd aR = ((mX * mY.transpose()) / n).array();
aR = aR / ((1 - aR*aR) / (n-2)).sqrt(); // calculate t-statistics

students_t dist(n-1);
MatrixXd mP(m,g);
for(int i=0; i < m; ++i) {
    for(int j=0; j < g; ++j) {
        double t = aR(i,j);
        mP(i,j) = 2.0*cdf(dist, t > 0 ? 0-t : t);
    }
}
std::cout << mP << std::endl;
return 0;
}
```

Compiler option makes a big difference

- Code optimization significantly improves the performance of Eigen library

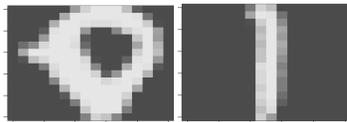
```
$ g++ -I ~/include -o bulkLinearTest bulkLinearTest.cpp
```

```
$ time ./bulkLinearTest Y.txt X.txt > P.txt
real 2m9.043s
```

```
$ g++ -O -I ~/include -o bulkLinearTest bulkLinearTest.cpp
```

```
$ time ./bulkLinearTest Y.txt X.txt > P.txt
real 0m35.037s
```

Image data : Handwritten digits



- Available at <http://www-stat.stanford.edu/tibs/ElemStatLearn/data.html>
- $16 \times 16 = 256$ pixels
- Each pixel value is scaled between -1 and 1
- Each image correspond to a single digit

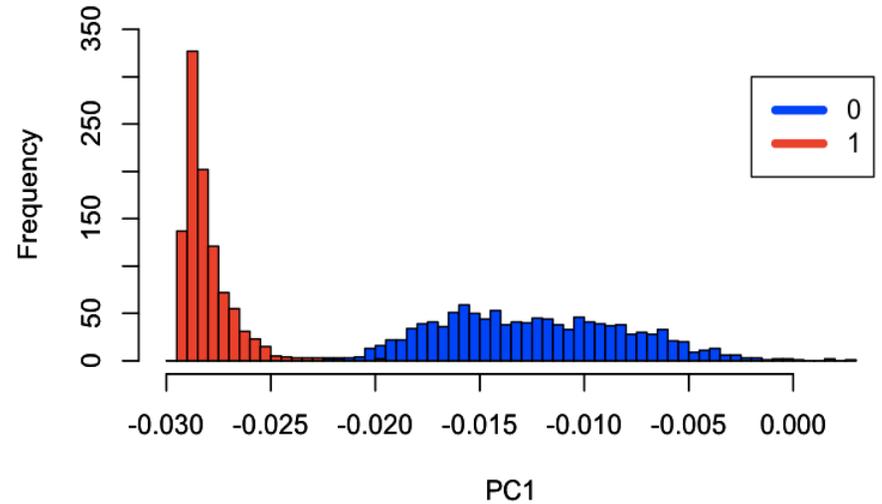
Clustering Handwritten Digits

- X : $n \times p$ matrix of p from n images
- Hidden variables - $Y \in \{0, 1\}^n$ - labels of zeros and ones
- Without known the actual label, the problem is to separate X into two different groups
- And compare whether the assignment was done correctly

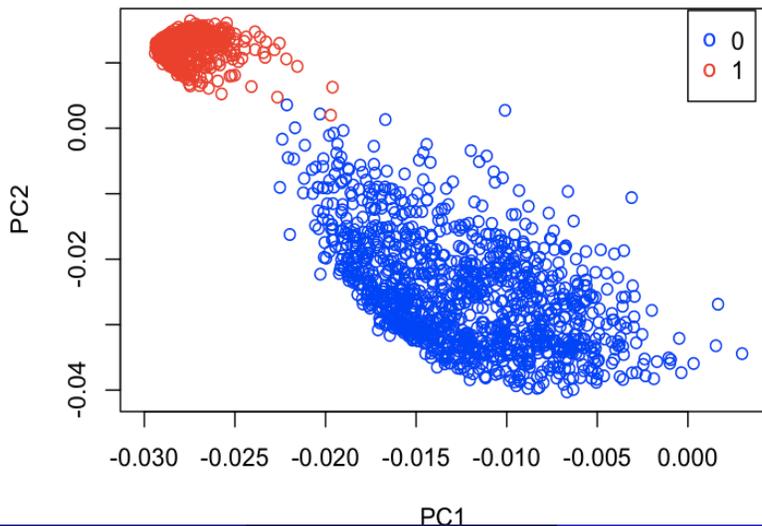
Clustering by Principal Components (or SVD)

- 1 Find SVD of $X : X = UDV^T$
- 2 Find top singular vector u_1 from U .
 - u_1 explains the most of variation between samples
 - u_1 is a linear combination of columns of X .
- 3 Predict the label using the value of u_1 using a threshold value

Top PC correlates well with the hidden labels



Top 2 PCs are highly informative for predicting class labels



How to compute top k PCs from a matrix

```
int main(int argc, char** argv) {
    Matrix615<double> fX(argv[1]); // read m * n matrix
    double k = atoi(argv[2]);
    int n = fX.rowNums();
    int m = fX.colNums();

    MatrixXd mX;
    fX.cloneToEigen(mX);
    JacobiSVD<MatrixXd> svd(mX, ComputeThinU);

    // print out first $k$ PCs
    std::cout << svd.matrixU().block(0,0,n,k) << std::endl;

    return 0;
};
```

Caveat: SVD in Eigen library is actually NOT optimal

- JacobiSVD routine provides a high reliability and accuracy, but slow
- LAPACK routine used in R is much more efficient
- No other SVD routine is currently implemented in Eigen library
- If SVD performance is critical, you may want to use alternative solution such as BLAS/LAPACK

Performing multiple regression with eigenvectors

If top k eigen vectors U_k is used for regression, $X = U_k D V^T$, and $D = V = I$.

$$\begin{aligned} \hat{\beta} &= VD^{-1}U^T\mathbf{y} \\ &= U^T\mathbf{y} \\ \widehat{Cov}(\hat{\beta}) &= \sigma^2(X^T X)^{-1} \\ &= \frac{(\mathbf{y} - X\hat{\beta})^T(\mathbf{y} - X\hat{\beta})}{n - p} (VD^{-1}(VD^{-1})^T) \\ &= \frac{(\mathbf{y} - X\hat{\beta})^T(\mathbf{y} - X\hat{\beta})}{n - p} \end{aligned}$$

Today

- Recap on multiple regression
- Bulk linear test
 - Leveraging correlation structure for rapid computation
 - type-specific specialization of lexical_cast
 - -o option improves performance significantly
- SVD computation itself may be slower than R