Biostatistics 615/815 Lecture 22: Matrix with C++

Hyun Min Kang

December 1st, 2011

Recap - A case for simple linear regression

Ingredients for simplification

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

Making faster inferences

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

Recap - Streaming the inputs to extract sufficient statistics

Sufficient statistics for simple linear regression

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

Recap - Streaming the inputs to extract sufficient statistics

Sufficient statistics for simple linear regression

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

Extracting sufficient statistics from stream

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

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 $\hat{\beta}$: $\hat{\mathrm{Var}}(\hat{\beta}) = \hat{\sigma}^2(X^T X)^{-1}$
- 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

$$
\begin{aligned}
X &= UDV' \\
\hat{\beta} &= (X^T X)^{-1} X^T \mathbf{y} \\
&= (VDU^T UDV')^{-1} VDU^T \mathbf{y} \\
&= (VD^2 V^T)^{-1} VDU^T \mathbf{y} \\
&= VD^{-2} V^T VDU^T \mathbf{y} \\
&= VD^{-1} U^T \mathbf{y} \\
\hat{\mathrm{Cov}}(\hat{\beta}) &= \hat{\sigma}^2 (X'X)^{-1} \\
&= \hat{\sigma}^2 (VD^{-2} V^T) \\
&= \frac{(\mathbf{y} - X\hat{\beta})^T (\mathbf{y} - X\hat{\beta})}{n-p} (VD^{-1} (VD^{-1})^T)
\end{aligned}
$$

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
  // calcuate 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	+++	++

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

Reading large matrix from files

```
void Matrix615::readFromFile(const char* fileName) {
  // omitting the first few lines
  // ....
    while( std::getline(ifs, line) ) {
      if ( line[0] == '#' ) continue; // skip meta-lines starting with #
      wsTokenizer t(line,sep);
      data.resize(nr+1);

      for(wsTokenizer::iterator i=t.begin(); i != t.end(); ++i) {
        // the line below is >10 times slower than atof() function
data[nr].push_back(boost::lexical_cast<T>(*i));
  // ... skipping the rest part
}
```

Type-specific function overloading

```
// The following code will make lexical_cast from string to int or double
// much more efficient
namespace boost {
  template<>
    inline int lexical_cast(const std::string& arg)
    {
      char* stop;
      int res = strtol( arg.c_str(), &stop, 10 ); // string to integer
      if ( *stop != 0 ) throw_exception(bad_lexical_cast(typeid(int),
                                        typeid(std::string)));
      return res;
    }
  template<>
    inline double lexical_cast(const std::string& arg)
    {
      char* stop;
      double res = strtod( arg.c_str(), &stop ); // string to double
      if ( *stop != 0 ) throw_exception(bad_lexical_cast(typeid(double),
                                        typeid(std::string)));
      return res;
    }
}; //namespace boost
```

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

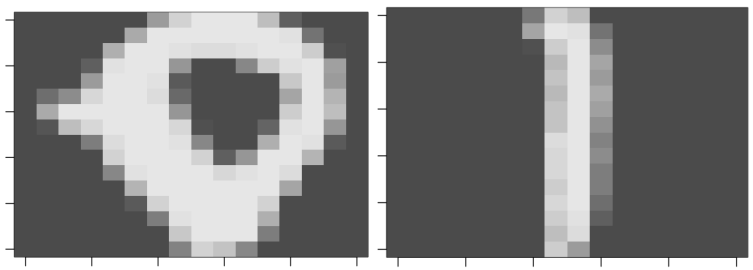
```
$ g++ -I ~/include -o bulkLinearTest bulkLinearTest.cpp

$ 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



- Avaialable 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

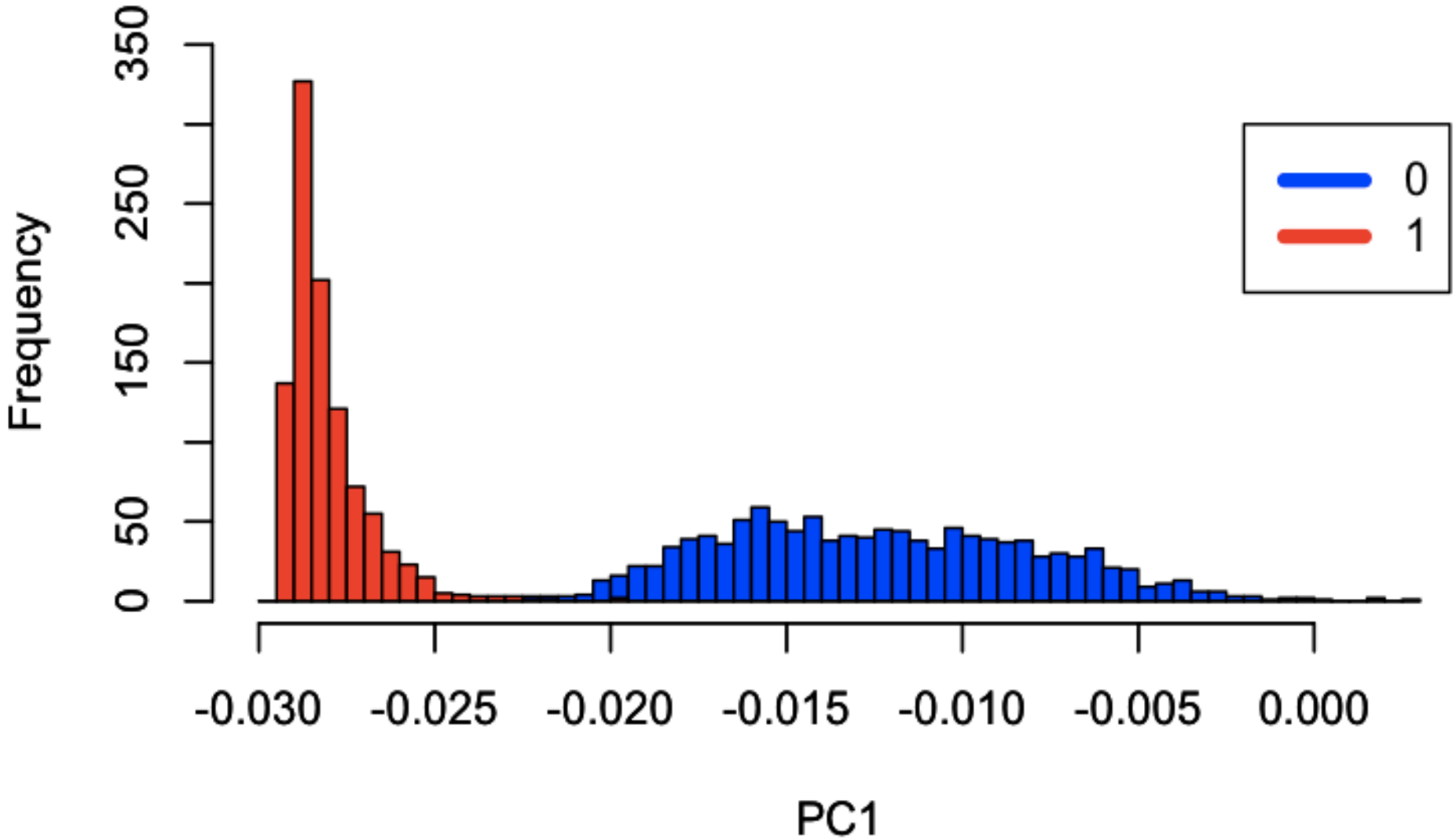
Clustering Handwritten Digits

- 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

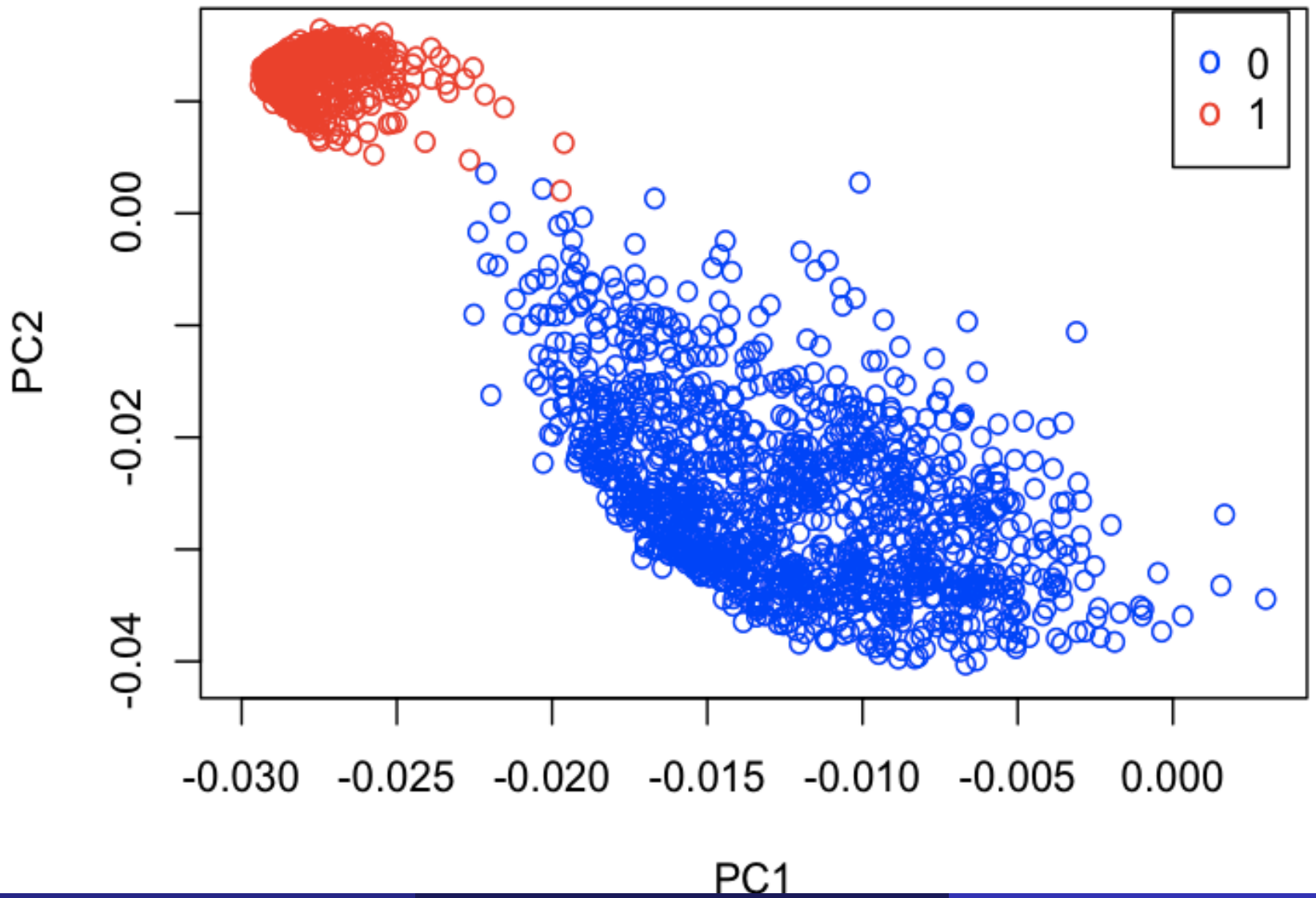
Clustering by Principal Components (or SVD)

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 columnes 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 crtical, 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, $X = U_k = U_k D V$, and $D = V = I$.

$$\begin{aligned}
\hat{\beta} &= VD^{-1}U^T\mathbf{y} \\
&= U^T\mathbf{y} \\
\hat{\mathrm{Cov}}(\hat{\beta}) &= \hat{\sigma}^2(X'X)^{-1} \\
&= \frac{(\mathbf{y}-X\hat{\beta})^T(\mathbf{y}-X\hat{\beta})}{n-p}(VD^{-1}(VD^{-1})^T) \\
&= \frac{(\mathbf{y}-X\hat{\beta})^T(\mathbf{y}-X\hat{\beta})}{n-p}
\end{aligned}$$

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