A Note on Statistical Computing

Ying Nian Wu, UCLA Statistics

For STATS 202A, Fall quarter, 2016

Contents

Preface

This note is mainly about the computational side of the commonly used modern statistical and machine learning methods. The main theme of the first three chapters is linear regression and its rich variations that span much of statistics and machine learning. The last chapter is on Monte Carlo methods.

Writing R and Python code to implement these methods enable us to gain first-hand experiences with these methods.

1 Linear regression: least squares, ridge, Lasso

1.1 Sweep operator

The sweep operator is a convenient tool for linear regression. For an $n \times n$ squared matrix $A = (a_{ij})$, let $\tilde{A} = (\tilde{a}_{ij}) = \text{SWP}[k]A$, then

$$\begin{aligned}
\tilde{a}_{kk} &= -\frac{1}{a_{kk}}, \\
\tilde{a}_{kj} &= \frac{a_{kj}}{a_{kk}},\ j \neq k, \\
\tilde{a}_{ik} &= \frac{a_{ik}}{a_{kk}},\ i \neq k, \\
\tilde{a}_{ij} &= a_{ij} - \frac{a_{ik}a_{kj}}{a_{kk}},\ i \neq k, j \neq k.
\end{aligned}$$

We can apply the sweep operator sequentially, e.g., $\text{SWP}[1:m]$ means we apply the sweep operator for $k = 1:m$.

There is also a matrix version of the sweep operator. Let

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix},$$

where A is $n \times n$, and A_{11} is $m \times m$, $m \leq n$. Let $\tilde{A} = \text{SWP}[A_{11}]A$, then

$$\tilde{A} = \begin{bmatrix} \tilde{A}_{11} & \tilde{A}_{12} \\ \tilde{A}_{21} & \tilde{A}_{22} \end{bmatrix} = \begin{bmatrix} -A_{11}^{-1} & A_{11}^{-1}A_{12} \\ A_{21}A_{11}^{-1} & A_{22} - A_{21}A_{11}^{-1}A_{12} \end{bmatrix}.$$

The most important property of the sweep operator is

$$\mathrm{SWP}[A_{11}] = \mathrm{SWP}[1:m],$$

and the order of operations in $\mathrm{SWP}[1:m]$ does not matter. The above property is obviously true if A_{11} is a scalar, i.e., 1×1. It appears mysterious if A_{11} is a matrix. We shall explain why it is true in a later section. Computationally, the above property enables us to implement the matrix sweep by a sequence of scalar sweeps. In particular, $\mathrm{SWP}[1:n]A = -A^{-1}$.

R code:

```
mySweep <- function(A, m)
{
n <- dim(A)[1]

for (k in 1:m)
{
  for (i in 1:n)
   for (j in 1:n)
      if (i!=k  & j!=k)
         A[i,j] <- A[i,j] - A[i,k]*A[k,j]/A[k,k]

  for (i in 1:n)
    if (i!=k)
      A[i,k] <- A[i,k]/A[k,k]

  for (j in 1:n)
    if (j!=k)
      A[k,j] <- A[k,j]/A[k,k]

  A[k,k] <- - 1/A[k,k]
}
return(A)
}

A = matrix(c(1,2,3,7,11,13,17,21,23), 3,3)
solve(A)
mySweep(A,3)
```

C++ code:

```
library(Rcpp)
```

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericMatrix mySweep_cpp(const NumericMatrix A_in, int m)
{
    NumericMatrix A = clone(A_in);
    int n = A.nrow();

    for (int k = 0; k < m; k++)
    {
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if ((i != k) & (j != k))
```

```
                    A(i,j) = A(i,j) - A(i,k)*A(k,j)/A(k,k);

        for (int i = 0; i < n; i++)
            if (i != k)
                A(i,k) = A(i,k)/A(k,k);

        for (int j = 0; j < n; j++)
            if (j != k)
                A(k,j) = A(k,j)/A(k,k);

        A(k,k) = - 1/A(k,k);
    }
    return A;
}
```

Python code:

```
import numpy as np

def mySweep(B, m):
    A = np.copy(B)
    n, c = A.shape
    for k in range(m):
        for i in range(n):
            for j in range(n):
                if i != k and j != k:
                    A[i,j] = A[i,j]- A[i,k]*A[k,j]/A[k,k]

        for i in range(n):
            if i != k:
                A[i,k] = A[i,k]/A[k,k]

        for j in range(n):
            if j != k:
                A[k,j] = A[k,j]/A[k,k]

        A[k,k] = -1/A[k,k]
    return A

A = np.array([[1,2,3],[7,11,13],[17,21,23]], dtype=float).T
print mySweep(A, 3)
```

1.2 Linear Regression

The dataset of linear regression consists of an $n \times p$ matrix $\mathbf{X} = (x_{ij})$, and a $n \times 1$ vector $\mathbf{Y} = (y_i)$. The model is of the following form:

$$y_i = \sum_{j=1}^{p} x_{ij}\beta_j + \epsilon_i,$$

for $i = 1, ..., n$, where $\epsilon_i \sim \mathrm{N}(0, \sigma^2)$ independently for $i = 1, ..., n$. Here we are deliberately ambiguous about intercept term. If $x_{i1} = 1$ for all i, then β_1 will be the the intercept term. $[\mathbf{X}, \mathbf{Y}]$ is called the training data. y_i is called response variable, outcome, dependent variable. x_{ij} is called predictor, regressor, covariate, independent variable, or simple variable. In the experimental design setting, $\mathbf{X}$ is called the design matrix.

The process of estimating β is called learning from the training data. The purpose is two-fold.

obs	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1	$x_{11}, x_{12}, ..., x_{1p}$	y_1
2	$x_{21}, x_{22}, ..., x_{2p}$	y_2
...		
n	$x_{n1}, x_{n2}, ..., x_{np}$	y_n

(1) Explanation: understanding the relationship between y_i and $(x_{ij}, j = 1, ..., p)$.

(2) Prediction: learn to predict y_i based on $(x_{ij}, j = 1, ..., p)$, so that in the testing stage, if we are given the predictor variables, we should be able to predict the outcome.

obs	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1	$X_1^\top$	y_1
2	$X_2^\top$	y_2
...		
n	$X_n^\top$	y_n

We can arrange the data in terms of $X_i^\top = (x_{ij}, j = 1, ..., p)$, where $X_i^\top$ is the i-th row of $\mathbf{X}$. Here X_i is not in bold font. We can write the model as $y_i = X_i^\top \beta + \epsilon_i$, where $\beta = (\beta_j, j = 1, ..., p)^\top$.

obs	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1 2 ... n	$\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_p$	$\mathbf{Y}$

We can also arrange the data in terms of $\mathbf{X}_j = (x_{ij}, i = 1, ..., n)$, where $\mathbf{X}_j$ is the j-th column of $\mathbf{X}$. Here $\mathbf{X}_j$ is in bold font. We can write the model as $\mathbf{Y} = \sum_{j=1}^p \mathbf{X}_j \beta_j + \epsilon$, where $\epsilon = (\epsilon_i, i = 1, ..., n)^\top \sim \mathrm{N}(0, \sigma^2 \mathbf{I}_n)$, where $\mathbf{I}_n$ is the n-dimensional identity matrix.

We can write the linear regression model as $\mathbf{Y} = \mathbf{X}^\top \beta + \epsilon$. The least squares estimate of β is

$$\hat{\beta} = \arg\min_\beta \|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2 = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y}.$$

We construct a matrix $\mathbf{Z} = [\mathbf{X}\mathbf{Y}]$, and let

$$A = \mathbf{Z}^\top \mathbf{Z} = \begin{bmatrix} \mathbf{X}^\top \mathbf{X} & \mathbf{X}^\top \mathbf{Y} \\ \mathbf{Y}^\top \mathbf{X} & \mathbf{Y}^\top \mathbf{Y} \end{bmatrix}$$

be the cross-product matrix. Then

$$\mathrm{SWP}[1:p]A = \begin{bmatrix} -(\mathbf{X}^\top \mathbf{X})^{-1} & (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y} \\ \mathbf{Y}^\top \mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} & \mathbf{Y}^\top \mathbf{Y} - \mathbf{Y}^\top \mathbf{X} (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{Y} \end{bmatrix} = \begin{bmatrix} -\frac{\mathrm{Var}(\hat{\beta})}{\sigma^2} & \hat{\beta} \\ \hat{\beta}^\top & \mathrm{RSS} \end{bmatrix},$$

where $\mathrm{RSS} = \|\mathbf{Y} - \mathbf{X}\hat{\beta}\|_{\ell_2}^2$ is the residual sum of squares.

We shall expand our treatment of linear regression in later sections. For now, it is enough to know that the sweep operator gives us all the key results we need for linear regression.

R code:

```
n = 100
p = 5
X = matrix(rnorm(n*p), nrow=n)
beta = matrix(1:p, nrow = p)
```

```
Y = X %*% beta + rnorm(n)
lm(Y~X)
Z = cbind(rep(1, n), X, Y)
A = t(Z) %*% Z
S = mySweep(A, p+1)
beta = S[1:(p+1), p+2]
```

We can compare our results with the built-in function in R:

R code:

```
n = 100
X1 = rnorm(n)
X2 = rnorm(n)
Y = X1 + 2*X2 + rnorm(n)
lm(Y~X1+X2)

A = data.frame(x1 = X1, x2 = X2, y = Y)
lm(y ~ x1 + x2, data = A)

data(trees)
lt = log(trees)
m <- lm(Volume~Height+Girth, data=lt)
summary(m)
```

1.3 Gauss-Jordan elimination

For a system of linear equations $Ax = b$, where $A = (a_{ij})$ is $n \times n$, $x = (x_i)$ is $n \times 1$, and $b = (b_i)$ is $n \times 1$, we can solve $x = A^{-1}b$ by Gauss-Jordan elimination.

Specifically, for any matrix A (here we assume A can be any $n \times N$ matrix, e.g., $N = n + 1$ or $N = 2n$), let $\tilde{A} = \mathrm{GJ}[k]A$, then

$$\begin{aligned} \tilde{A}_k &= A_k/a_{kk}, \\ \tilde{A}_i &= A_i - a_{ik}\tilde{A}_k,\ i \neq k, \end{aligned}$$

where A_k is the k-th row of A. The above operation makes $\tilde{a}_{kk} = 1$, and $\tilde{a}_{ik} = 0$ for $i \neq k$. We can apply Gauss-Jordan sequentially, e.g., $\mathrm{GJ}[1:m]$ means we apply Gauss-Jordan for $k = 1 : m$. Being a row operation, Gauss-Jordan is linear, i.e., $\tilde{A} = \mathrm{GJ}[k]A$ amounts to $\tilde{A} = G_k A$ for a matrix G_k. Thus

$$\mathrm{GJ}[1:n][A|b] = [I|A^{-1}b] = A^{-1}[A|b],$$

$$\mathrm{GJ}[1:n][A|I] = [I|A^{-1}] = A^{-1}[A|I].$$

That is, $\mathrm{GJ}[1:n] = A^{-1}$, and order does not matter. Moreover,

$$\mathrm{GJ}[1:m]\begin{bmatrix} A_{11} & A_{12} & | & I_1 & 0 \\ A_{21} & A_{22} & | & 0 & I_2 \end{bmatrix} = \begin{bmatrix} I_1 & A_{11}^{-1}A_{12} & | & A_{11}^{-1} & 0 \\ 0 & A_{22} - A_{21}A_{11}^{-1}A_{12} & | & -A_{21}A_{11}^{-1} & I_2 \end{bmatrix}.$$

We can write $\mathrm{GJ}[1:m] = \mathrm{GJ}[A_{11}]$, which is a matrix version of Gauss-Jordan. If we compare Gauss-Jordan $\mathrm{GJ}[1:m]$ with the sweep operator $\mathrm{SWP}[1:m]$, we can see that sweep operator is a space saving version of Gauss-Jordan, where we do not record the identity matrix in sweep. Because $\mathrm{GJ}[1:m] = \mathrm{GJ}[A_{11}]$, we have $\mathrm{SWP}[1:m] = \mathrm{SWP}[A_{11}]$. The above equation also leads to the identity

$$|A| = |A_{11}||A_{22} - A_{21}A_{11}^{-1}A_{12}|,$$

where $|A|$ denotes the determinant of A. Thus we can compute $|A|$ by the sweep operator, in addition to A^{-1}.

R code:

```
myGaussJordan <- function(A, m)
{
n <- dim(A)[1]
B <- cbind(A, diag(rep(1, n)))

for (k in 1:m)
{
  a <- B[k, k]
  for (j in 1:(n*2))
     B[k, j] <- B[k, j]/a
  for (i in 1:n)
     if (i != k)
     {
        a <- B[i, k]
        for (j in 1:(n*2))
           B[i, j] <- B[i, j] - B[k, j]*a;
    }
 }
return(B)
}

A = matrix(c(1,2,3,7,11,13,17,21,23), 3,3)
solve(A)
myGaussJordan(A,3)

myGaussJordanVec <- function(A, m)
{
n <- dim(A)[1]
B <- cbind(A, diag(rep(1, n)))

for (k in 1:m)
 {
  B[k, ] <- B[k, ]/B[k, k]
  for (i in 1:n)
     if (i != k)
        B[i, ] <- B[i, ] - B[k, ]*B[i, k];
 }
return(B)
}

A = matrix(c(1,2,3,7,11,13,17,21,23), 3,3)
solve(A)
myGaussJordanVec(A,3)
```

Python code:

```
def myGaussJordan(A, m):
    n = A.shape[0]
    B  = np.hstack((A, np.identity(n)))

    for k in range(m):
        a = B[k, k]
        for j in range(n*2):
            B[k, j] = B[k, j] / a
```

```
        for i in range(n):
            if i != k:
                a = B[i, k]
                for j in range(n*2):
                    B[i, j] = B[i, j] - B[k, j]*a;
    return B

def myGaussJordanVec(A, m):
    n = A.shape[0]
    B  = np.hstack((A, np.identity(n)))

    for k in range(m):
        B[k, :] = B[k, ] / B[k, k]
        for i in range(n):
            if i != k:
                B[i, ] = B[i, ] - B[k, ]*B[i, k];

    return B

A = np.array([[1,2,3],[7,11,13],[17,21,23]], dtype=float).T
print myGaussJordan(A, 3)
print myGaussJordanVec(A, 3)
```

1.4 Ridge regression

The ridge regression estimates β by

$$\hat{\beta}_\lambda = \arg\min_\beta \left[\|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2 + \lambda\|\beta\|_{\ell_2}^2\right] = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_p)^{-1}\mathbf{X}^\top\mathbf{Y},$$

for a tuning parameter $\lambda > 0$. The $\lambda\|\beta\|_{\ell_2}^2$ term is a penalty or regularization term. The resulting estimator is called the shrinkage estimator.

The computation can be accomplished by the sweep operator.

R code:

```
myRidge <- function(X, Y, lambda)
{
  n = dim(X)[1]
  p = dim(X)[2]
  Z = cbind(rep(1, n), X, Y)
  A = t(Z) %*% Z
  D = diag(rep(lambda, p+2))
  D[1, 1] = 0
  D[p+2, p+2] = 0
  A = A + D
  S = mySweep(A, p+1)
  beta = S[1:(p+1), p+2]
  return(beta)
}
```

1.5 Spline regression

As an example, consider the piecewise linear spline model, where the training examples are (x_i, y_i), for $i = 1, ..., n$, and x_i is one-dimensional. The model assumes

$$y_i = \beta_0 + \sum_{j=1}^{p} \beta_j \max(0, x_i - k_j) + \epsilon_i,$$

where k_j is the j-th knot of the spline, and β_j is the change of slope at knot k_j. We can estimate β by minimizing

$$\sum_{i=1}^{n} \left[y_i - \left(\beta_0 + \sum_{j=1}^{p} \beta_j \max(0, x_i - k_j) \right) \right]^2 + \lambda \sum_{j=1}^{p} \beta_j^2,$$

where the regularization term prefers smooth spline function.

R code:

```
n = 20
p = 500
sigma = .1
lambda = 1.

x = runif(n)
x = sort(x)
Y = x^2 + rnorm(n)*sigma
X = matrix(x, nrow=n)
for (k in (1:(p-1))/p)
  X = cbind(X, (x>k)*(x-k))

beta = myRidge(X, Y, lambda)
Yhat = cbind(rep(1, n), X)%*%beta

plot(x, Y, ylim = c(-.2, 1.2), col = "red")
par(new = TRUE)
plot(x, Yhat, ylim = c(-.2, 1.2), type = 'l', col = "green")
```

The above spline model can be extended in the following two directions:

(1) Cubic spline. We replace the linear piece to cubic piece to make it continuously differentiable at the knots.

(2) Rectified neural network. The piecewise linear form of the linear spline also underlies modern multi-layer neural networks, where $\max(0, r)$ is called rectified linear unit (ReLU).

1.6 Coordinate descent and matching pursuit

To minimize $R(\beta) = \|\mathbf{Y} - \sum_{j=1}^{p} \mathbf{X}_j \beta_j\|_{\ell_2}^2$, we can use coordinate descent to update one β_j at a time. We can use the following two equivalent schemes:

(1) Let $\mathbf{R}_j = \mathbf{Y} - \sum_{k \neq j} \mathbf{X}_k \beta_k$. Update $\beta_j \leftarrow \langle \mathbf{R}_j, \mathbf{X}_j \rangle / \|\mathbf{X}_j\|^2$.

(2) Let $\mathbf{R} = \mathbf{Y} - \sum_{j=1}^{p} \mathbf{X}_j \beta_j$, compute $\Delta\beta_j \rightarrow \langle \mathbf{R}, \mathbf{X}_j \rangle / \|\mathbf{X}_j\|^2$. Then update $\beta_j \rightarrow \beta_j + \Delta\beta_j$, and $\mathbf{R} \rightarrow \mathbf{R} - \mathbf{X}_j \Delta\beta_j$.

R code:

```
n = 1000
p = 10
s = 10
T = 10000
X = matrix(rnorm(n*p), nrow=n)
beta_true = matrix(rep(0, p), nrow = p)
beta_true[1:s] = 1:s
Y = X %*% beta_true + rnorm(n)

beta = matrix(rep(0, p), nrow = p)
R = Y
ss = rep(0, p)
for (j in 1:p)
    ss[j] = sum(X[, j]^2)

for (t in 1:T)
{
    for (j in 1:p)
    {
        db = sum(R*X[, j])/ss[j]
        beta[j] = beta[j]+db
        R = R - X[, j]*db
    }
}
print(beta)
```

The matching pursuit or forward selection method is to select the best j to update.

R code:

```
beta = matrix(rep(0, p), nrow = p)
R = Y
db = rep(0, p)
for (t in 1:T)
{
    for (j in 1:p)
        db[j] = sum(R*X[, j])
    j = which.max(abs(db))
    db = db/ss[j]
    beta[j] = beta[j]+db[j]
    R = R - X[, j]*db[j]
}
print(beta)
```

1.7 Lasso regression

The Lasso regression estimate β by

$$\hat{\beta}_\lambda = \arg\min_\beta \left[\frac{1}{2} \|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2 + \lambda \|\beta\|_{\ell_1} \right],$$

where $\|\beta\|_{\ell_1} = \sum_{j=1}^p |\beta_j|$. Lasso stands for "least absolute shrinkage and selection operator." There is no closed form solution for general p.

We do have closed form solution for $p = 1$, where $\mathbf{X}$ is an $n \times 1$ vector,

$$\hat{\beta}_\lambda = \begin{cases} (\langle \mathbf{Y}, \mathbf{X} \rangle - \lambda)/\|\mathbf{X}\|_{\ell_2}^2, & \text{if } \langle \mathbf{Y}, \mathbf{X} \rangle > \lambda; \\ (\langle \mathbf{Y}, \mathbf{X} \rangle + \lambda)/\|\mathbf{X}\|_{\ell_2}^2, & \text{if } \langle \mathbf{Y}, \mathbf{X} \rangle < -\lambda; \\ 0 & \text{if } |\langle \mathbf{Y}, \mathbf{X} \rangle| < \lambda. \end{cases}$$

We can write it as

$$\hat{\beta}_\lambda = \text{sign}(\hat{\beta}) \max(0, |\hat{\beta}| - \lambda/\|\mathbf{X}\|_{\ell_2}^2),$$

where $\hat{\beta} = \langle \mathbf{Y}, \mathbf{X} \rangle / \|\mathbf{X}\|_{\ell_2}^2$ is the least squares estimator. The above transformation from $\hat{\beta}$ to $\hat{\beta}_\lambda$ is called soft thresholding.

Compare Lasso with ridge regression in one-dimensional situation, the latter being

$$\hat{\beta}_\lambda = \langle \mathbf{Y}, \mathbf{X} \rangle / (\|\mathbf{X}\|_{\ell_2}^2 + \lambda),$$

the behavior of Lasso is richer, including both shrinkage (by subtracting λ) and selection (via thresholding at λ).

The reason for the fact that $\hat{\beta}$ can be zero is that the left and right derivatives of $|\beta|$ at 0 are not the same, so that the function $\|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2/2 + \lambda\|\beta\|_{\ell_1}$ may have a negative left derivative and a positive right derivative at 0, so that 0 can be the minimum. For $|\beta|^{1+\delta}$ with $\delta > 0$, its derivative at 0 is 0, so that $\hat{\beta}$ cannot be zero in general. For $|\beta|^{1-\delta}$, there is a sharp turn at 0, but it is not convex anymore. $|\beta|$ or piecewise linear function in general is the only choice that has a sharp turn at 0 but is still barely convex.

Thus the Lasso regression prefers sparse β, i.e., only a small number of components of β are non-zero.

1.8 Primal form of Lasso

The primal form of Lasso is $\min \|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2/2$ subject to $\|\beta\|_{\ell_1} \le t$. The dual form of Lasso is $\min \|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2/2 + \lambda\|\beta\|_{\ell_1}$. The two forms are equivalent with a one-to-one correspondence between t and λ. If $\hat{\beta}_\lambda$ is the solution to the dual form, then it must be the solution to the primal form with $t = \|\hat{\beta}_\lambda\|_{\ell_1}$. The reason is that if a different $\hat{\beta}$ is the solution to the primal form, then $\hat{\beta}$ is a better solution to the dual form than $\hat{\beta}_\lambda$, which results in contradiction.

The primal form also reveals the sparsity inducing property of ℓ_1 regularization in that the ℓ_1 ball has low-dimensional corners, edges, and faces, but is still barely convex.

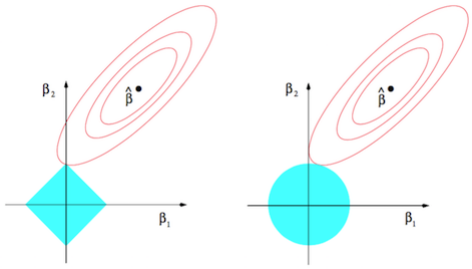

Figure 1: Lasso. Source: web.

The above is the well known figure of Lasso taken from the web. Take the left plot for example. The blue region is $\|\beta\|_{\ell_1} \le t$. The red curves is the contour plot, where each red elliptical circle consists of those β that have the same value of $\|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2$. The circle on the outside has bigger$\|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2$ than the circle inside. The solution to the problem of $\min \|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2$ subject to $\|\beta\|_{\ell_1} \le t$ is where the red circle touches the blue region. Any other points in the blue region will be outside the outer red circle and thus have bigger

values of $\|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2$. The reason that the ℓ_1 regularization induces sparsity is that it is likely for the red circle to touch the blue region at a corner, which is a sparse solution. If we use ℓ_2 regularization, as is the case with the plot on the right, then the solution is not sparse in general.

1.9 Coordinate descent for Lasso solution path

For multi-dimensional $\mathbf{X} = (\mathbf{X}_j, j = 1, ..., p)$, we can use the coordinate descent algorithm to compute $\hat{\beta}_\lambda$. The algorithm updates one component at a time, i.e., given the current values of $\beta = (\beta_j, j = 1, ..., p)$, let $\mathbf{R}_j = \mathbf{Y} - \sum_{k \neq j} \mathbf{X}_k \beta_k$, we can update $\beta_j = \text{sign}(\hat{\beta}_j) \max(0, |\hat{\beta}_j| - \lambda / \|\mathbf{X}\|_{\ell_2}^2)$, where $\hat{\beta}_j = \langle \mathbf{R}_j, \mathbf{X}_j \rangle / \|\mathbf{X}_j\|_{\ell_2}^2$.

We can find the solution path of Lasso by starting from a big λ so that all of the estimated β_j are zeros. Then we gradually reduce λ. For each λ, we cycle through $j = 1, ..., p$ for coordinate descent until convergence, and then we lower λ. This gives us $\hat{\beta}(\lambda)$ for the whole range of λ. The whole process is a forward selection process, which sequentially selects new variables and occasionally removes selected variables.

R code:

```
n = 50
p = 200
s = 10
T = 10
lambda_all = (100:1)*10
L = length(lambda_all)

X = matrix(rnorm(n*p), nrow=n)
beta_true = matrix(rep(0, p), nrow = p)
beta_true[1:s] = 1:s
Y = X %*% beta_true + rnorm(n)

beta = matrix(rep(0, p), nrow = p)
beta_all = matrix(rep(0, p*L), nrow = p)

R = Y
ss = rep(0, p)
for (j in 1:p)
    ss[j] = sum(X[, j]^2)

err = rep(0, L)
for (l in 1:L)
{
    lambda = lambda_all[l]
    for (t in 1:T)
    {
      for (j in 1:p)
      {
        db = sum(R*X[, j])/ss[j]
        b = beta[j]+db
        b = sign(b)*max(0, abs(b)-lambda/ss[j])
        db = b - beta[j]
        R = R - X[, j]*db
        beta[j] = b
      }
    }
    beta_all[, l] = beta
    err[l] = sum((beta-beta_true)^2)
}
par(mfrow=c(1,2))
matplot(t(matrix(rep(1, p), nrow = 1)%*%abs(beta_all)), t(beta_all), type = 'l')
```

```
plot(lambda_all, err, type = 'l')
```

1.10 Least angle regression

In the above algorithm, at any given λ, let $\mathbf{R} = \mathbf{Y} - \sum_{j=1}^{p} \mathbf{X}_j \beta_j$, then $\hat{\beta}_j = \beta_j + \langle \mathbf{R}, \mathbf{X}_j \rangle / \|\mathbf{X}_j\|_{\ell_2}^2$. If β is the Lasso solution, then

$$\langle \mathbf{R}, \mathbf{X}_j \rangle = \begin{cases} \lambda, & \text{if } \beta_j > 0, \\ -\lambda, & \text{if } \beta_j < 0, \\ s\lambda & \text{if } \beta_j = 0. \end{cases}$$

where $|s| < 1$. Thus in the above process, for all of those selected $\mathbf{X}_j$, the algorithm maintains that $\langle \mathbf{R}, \mathbf{X}_j \rangle$ to be λ or $-\lambda$, for all selected $\mathbf{X}_j$. If we interpret $|\langle \mathbf{R}, \mathbf{X}_j \rangle|$ in terms of the angle between $\mathbf{R}$ and $\mathbf{X}_j$, then we may call the above process the equal angle regression or the least angle regression (LARS). In fact, the solution path is piecewise linear, and the LARS computes the linear pieces analytically instead of gradually reducing λ as in coordinate descent.

1.11 Stagewise regression or epsilon-boosting

The stagewise regression iterates the following steps. Given the current $\mathbf{R} = \mathbf{Y} - \sum_{j=1}^{p} \mathbf{X}_j \beta_j$, find j with the maximal $|\langle \mathbf{R}, \mathbf{X}_j \rangle|$. Then update $\beta_j \leftarrow \beta_j + \epsilon \langle \mathbf{R}, \mathbf{X}_j \rangle$ for a small ϵ. This is similar to the matching pursuit but is much less greedy. Such an update will change $\mathbf{R}$ and reduce $|\langle \mathbf{R}, \mathbf{X}_j \rangle|$, until another $\mathbf{X}_j$ catches up. So overall, the algorithm ensures that all of the selected $\mathbf{X}_j$ to have the same $|\langle \mathbf{R}, \mathbf{X}_j \rangle|$, which is the case with the algorithm in the above two sections. The stagewise regression is also called ϵ-boosting.

R code:

```
T = 3000
epsilon = .0001
beta = matrix(rep(0, p), nrow = p)
db = matrix(rep(0, p), nrow = p)
beta_all = matrix(rep(0, p*T), nrow = p)

R = Y
for (t in 1:T)
    {
      for (j in 1:p)
        db[j] = sum(R*X[, j])
      j = which.max(abs(db))
      beta[j] = beta[j]+db[j]*epsilon
      R = R - X[, j]*db[j]*epsilon
      beta_all[, t] = beta
}
matplot(t(matrix(rep(1, p), nrow = 1)%*%abs(beta_all)), t(beta_all), type = 'l')
```

We can also view the stagewise regression from the perspective of the primal form of the Lasso problem: minimize $\|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2$ subject to $\|\beta\|_{\ell_1} \le t$. If we relax the constraint by increasing t to $t + \Delta t$, then we want to update β_j with the maximal $|\langle \mathbf{R}, \mathbf{X}_j \rangle|$ in order to maximally reducing $\|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2$.

1.12 Bayesian regression

There is also a Bayesian interpretation of regularization.

For example, the ridge regression has a Bayesian interpretation. Let $\beta \sim \mathrm{N}(0, \tau^2 \mathbf{I}_p)$ be the prior distribution of β. The log probability density of β and $\mathbf{Y}$ is

$$-\frac{1}{2\sigma^2}\|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2 - \frac{1}{2\tau^2}\|\beta\|_{\ell_2}^2,$$

up to an additive constant. The above function is quadratic in β. By setting the first derivative to 0, we get the mode of β,

$$\hat{\beta} = (\mathbf{X}^\top \mathbf{X}/\sigma^2 + \mathbf{I}_p/\tau^2)^{-1}\mathbf{X}^\top \mathbf{Y}/\sigma^2.$$

which corresponds to the ridge regression with $\lambda = \sigma^2/\tau^2$. The second derivative or the Hessian matrix is $H = \mathbf{X}^\top \mathbf{X}/\sigma^2 + \mathbf{I}_p/\tau^2$. The inverse is the variance-covariance matrix $V = H^{-1}$. So the posterior distribution of β given $\mathbf{X}$ and $\mathbf{Y}$ is

$$[\beta|\mathbf{X}, \mathbf{Y}] \sim \mathrm{N}\left(\hat{\beta}, V\right).$$

Both $\hat{\beta}$ and V can be obtained by the sweep operator, very much like the original linear regression.

1.13 Calculation details

To flesh out the details of least squares estimation, let

$$R(\beta) = \sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2,$$

we have

$$\frac{R(\beta)}{\partial \beta_j} = -2\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p} x_{ij}\beta_j\right)x_{ij}.$$

We can pack the above results in terms of $\mathbf{X}_j$ and X_i. Recall

obs	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1	$X_1^\top$	y_1
2	$X_2^\top$	y_2
...		
n	$X_n^\top$	y_n

obs	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1 2 ... n	$\mathbf{X}_1, \mathbf{X}_2, ..., \mathbf{X}_p$	$\mathbf{Y}$

$$\frac{\partial R(\beta)}{\partial \beta_j} = -2\langle \mathbf{Y} - \sum_{j=1}^{p}\mathbf{X}_j\beta_j, \mathbf{X}_j\rangle = -2\mathbf{X}_j^\top(\mathbf{Y} - \sum_{j=1}^{p}\mathbf{X}_j\beta_j),$$

$$R'(\beta) = \begin{bmatrix}\partial R/\partial\beta_1 \\ \partial R/\partial\beta_2 \\ ... \\ \partial R/\partial\beta_n\end{bmatrix} = -2\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p}x_{ij}\beta_j\right)\begin{bmatrix}x_{i1}\\ x_{i2}\\ ...\\ x_{ip}\end{bmatrix} = -2\sum_{i=1}^{n}(y_i - X_i^\top\beta)X_i.$$

We can further pack the above two equations as

$$\frac{\partial R(\beta)}{\partial \beta_j} = -2\mathbf{X}^\top(\mathbf{Y} - \mathbf{X}\beta).$$

For the second derivative,

$$\frac{\partial^2 R(\beta)}{\partial \beta_j \beta_k} = 2\sum_{i=1}^{n} x_{ij}x_{ik} = 2\langle \mathbf{X}_j, \mathbf{X}_k \rangle.$$

Define

$$R''(\beta) = \left(\frac{\partial^2 R(\beta)}{\partial \beta_j \beta_k}\right)$$

be the $p \times p$ matrix, we can write

$$R''(\beta) = 2\sum_{i=1}^{n} X_i X_i^\top = 2\mathbf{X}^\top \mathbf{X},$$

which is positive definite.

In order to solve the least squares problem, we only need to solve $R'(\beta) = 0$. Geometrically, it means $\mathbf{X}_j \perp \mathbf{R}$, where $\mathbf{R} = \mathbf{Y} - \hat{\mathbf{Y}}$, and $\hat{\mathbf{Y}} = \sum_{j=1}^{p} \mathbf{X}_j \hat{\beta}_j$ is the projection of $\mathbf{Y}$ onto the subspace spanned by $(\mathbf{X}_j, j = 1, ..., p)$.

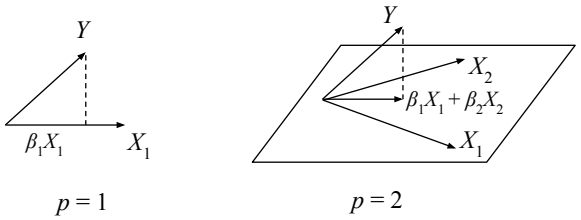

Figure 2: Least squares projection

The calculation for ridge regression is similar. Let $f(\beta) = \|\mathbf{Y} - \mathbf{X}\beta\|_{\ell_2}^2 + \lambda\|\beta\|_{\ell_2}^2$, then

$$f'(\beta) = -2\mathbf{X}^\top(\mathbf{Y} - \mathbf{X}\beta) + 2\lambda\beta,$$

$$f''(\beta) = 2(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_p).$$

1.14 Matrix calculus

Suppose $Y = (y_i)_{m\times 1}$, and $X = (x_j)_{n\times 1}$. Suppose $Y = h(X)$. We can define

$$\frac{\partial Y}{\partial X^\top} = \left(\frac{\partial y_i}{\partial x_j}\right)_{m\times n}.$$

If Y is a scaler, then the gradient $h'(X) = \partial Y/\partial X$ is a $n \times 1$ column vector, and $\partial Y/\partial X^\top$ is a $1 \times n$ row vector. For scaler Y, we can define the Hessian or second derivative

$$h''(X) = \frac{\partial^2 Y}{\partial X \partial X^\top} = \left(\frac{\partial^2 Y}{\partial x_i \partial x_j}\right)_{n\times n}.$$

If $Y = AX$, then $y_i = \sum_k a_{ik}x_k$. Thus $\partial y_i/\partial x_j = a_{ij}$. So $\partial Y/\partial X^\top = A$.

Chain rule. If $Y = h(X)$ and $X = g(Z)$, then $\partial y_i/\partial z_j = \sum_k (\partial y_i/\partial x_k)(\partial x_k/\partial z_j)$. Thus

$$\frac{\partial Y}{\partial Z^\top} = \frac{\partial Y}{\partial x^\top}\frac{\partial X}{\partial Z^\top}.$$

Product rule. If $Y = \langle h(X), g(X)\rangle = \sum_i h_i(X)g_i(X)$, then $\partial Y/\partial x_j = \sum_i [\partial h_i/\partial x_j g_i + h_i \partial g_i/\partial x_j]$. So

$$\frac{\partial Y}{\partial X^\top} = h(X)^\top \frac{\partial g(X)}{\partial X^\top} + g(X)^\top \frac{\partial h(X)}{\partial X^\top}.$$

Taylor expansion

$$f(X) = f(X_0) + \langle f'(X_0), X - X_0\rangle + \frac{1}{2}(X - X_0)^\top f''(X_0)(X - X_0) + o(|X - X_0|^2).$$

Jacobian. Let $Y = h(X)$ where both X and Y are $n \times 1$. Assume that h is one-to-one differentiable mapping. Let D_X be a local region around X in the domain of X. Suppose h maps D_X to a region D_Y in the domain of Y. Then as the size of D_X goes to 0, $|D_Y|/|D_X| \to |h'(X)|$, where $|h'(X)|$ is the determinant of $h'(X) = \partial Y/\partial X^\top$.

For $R(\beta) = \|\mathbf{Y} - \mathbf{X}\beta\|^2$, $R'(\beta) = -2\mathbf{X}^\top(\mathbf{Y} - \mathbf{X}\beta)$ and $R''(\beta) = 2\mathbf{X}^\top\mathbf{X}$. We can derive these by the chain rule. Let $e = \mathbf{Y} - \mathbf{X}\beta$. Then

$$\frac{\partial R}{\partial \beta^\top} = \frac{\partial R}{\partial e^\top}\frac{\partial e}{\partial \beta^\top} = -2e^\top\mathbf{X}.$$

$R'(\beta) = \partial R/\partial\beta$, which is obtained by transposing $-2e^\top\mathbf{X}$.

$$R''(\beta) = \frac{\partial^2 R}{\partial\beta\partial\beta^\top} = \partial(-2\mathbf{X}^\top e)/\partial\beta^\top = -2\mathbf{X}^\top\mathbf{X}.$$

1.15 Learning issues

For simplicity, assume $\mathbf{X}$ is orthonormal, i.e.. $\|\mathbf{X}_j\|_{\ell_2}^2 = 1$, and $\mathbf{X}_j \perp \mathbf{X}_k$ for $j \neq k$. Then the least squares estimate $\hat\beta_j = \langle \mathbf{Y}, \mathbf{X}_j\rangle$ and $\hat\beta = \mathbf{X}^\top\mathbf{Y}$.

Sampling distribution. Suppose $\mathbf{Y} = \mathbf{X}\beta_{\text{true}} + \epsilon$, where $\epsilon \sim \mathrm{N}(0, \sigma^2\mathbf{I}_n)$. Then $\hat\beta_j = \beta_{j,\text{true}} + \delta_j$, where $\delta_j = \langle \epsilon, \mathbf{X}_j\rangle \sim \mathrm{N}(0, \sigma^2)$ independently. Thus the sampling distribution of $\hat\beta_j \sim \mathrm{N}(\beta_{\text{true},j}, \sigma^2)$. $\hat\beta = \beta_{\text{true}} + \delta$, where $\delta = \mathbf{X}^\top\epsilon$ is the noise absorbed by $\mathbf{X}$.

Hypothesis testing and regularization. If we want to test $H_0 : \beta_{\text{true},j} = 0$ versus $H_1 : \beta_{\text{true},j} \neq 0$. We can use $\hat\beta_j$ as our test statistic. Under H_0, $\hat\beta_j = \delta_j \sim \mathrm{N}(0, \sigma^2)$. Our decision rule can be: accept H_0 if $|\hat\beta_j| \leq \lambda = 2\sigma$, and reject H_0 otherwise. This corresponds to a hard thresholding:

$$\hat\beta_{j,\text{Hypothesis},\lambda} = 1(|\hat\beta_j| > \lambda)\hat\beta_j.$$

The Lasso estimator is soft thresholding,

$$\hat\beta_{j,\text{Lasso},\lambda} = \text{sign}(\hat\beta_j)\max(0, |\hat\beta_j| - \lambda).$$

The ridge estimator is shrinkage,

$$\hat\beta_{j,\text{Ridge},\lambda} = \hat\beta_j/(1 + \lambda).$$

The point is that hypothesis testing and regularization are not that different. The common goal is to avoid overfitting or absorbing the noise, or interpreting noise as signal, or interpreting coincidence as pattern, which is bad for both explanation and prediction.

Training and testing errors. For least squares $\hat\beta = \mathbf{X}^\top\mathbf{Y}$, the training error is

$$\|\mathbf{Y} - \hat{\mathbf{Y}}\|^2 = \|\mathbf{X}\beta_{\text{true}} + \epsilon - \mathbf{X}\hat\beta\|^2 = \|\epsilon - \mathbf{X}^\top\delta\|^2,$$

whose expectation is $(n - p)\sigma^2$, assuming $p < n$. The reason is as follows. Let $\mathbf{X}_{p+1}, ..., \mathbf{X}_n$ be the complementary orthonormal vectors that are orthogonal to $\mathbf{X}_1, ..., \mathbf{X}_p$, then we can write $\epsilon = \sum_{j=1}^n \delta_j\mathbf{X}_j$, and $\epsilon - \mathbf{X}^\top\delta = \sum_{j=p+1}^n \delta_j\mathbf{X}_j$, so that the expected training error is $\sum_{j=p+1}^n \mathrm{E}(\delta_j^2) = (n - p)\sigma^2$.

The testing error is

$$\|\tilde{\mathbf{Y}} - \hat{\mathbf{Y}}\|^2 = \|\mathbf{X}\beta_{\text{true}} + \tilde{\epsilon} - \mathbf{X}\hat{\beta}\|^2 = \|\tilde{\epsilon} - \mathbf{X}^\top \delta\|^2,$$

whose expectation is $(n+p)\sigma^2$, because $\tilde{\epsilon}$ and δ are uncorrelated, because $\delta = \mathbf{X}^T\epsilon$, and $\tilde{\epsilon}$ is independent of ϵ. The difference between testing and training errors, which measures the amount of overfitting or over-optimism, is $2p\sigma^2$, where p is the model complexity or degrees of freedom. This motivates the Mallow's Cp, which minimizes

$$\frac{1}{2}\|\mathbf{Y} - \mathbf{X}\beta\|^2 + \|\beta\|_{\ell_0}\sigma^2$$

for variable selection, where we define the pseudo-norm $\|\beta\|_{\ell_0}$ to be the number of non-zero components in β. Lasso is to relax $\|\beta\|_{\ell_0}$ to $\|\beta\|_{\ell_1}$, which still maintains the selection behavior via thresholding. Ridge regression is to regularize by $\|\beta\|_{\ell_2}^2$.

For a general estimator $\hat{\beta}_\lambda$, the difference between testing error and training error is

$$\|\mathbf{X}\beta_{\text{true}} + \tilde{\epsilon} - \mathbf{X}\hat{\beta}_\lambda\|^2 - \|\mathbf{X}\beta_{\text{true}} + \epsilon - \mathbf{X}\hat{\beta}_\lambda\|^2,$$

whose expectation is $2\text{E}[\langle \epsilon, \mathbf{X}\hat{\beta}_\lambda\rangle]$. So we may define the effective model complexity or effective degrees of freedom

$$p_{\text{effective}} = \text{E}[\langle \epsilon, \mathbf{X}\hat{\beta}_\lambda\rangle]/\sigma^2.$$

The difference between testing and training errors can be small if $\hat{\beta}_\lambda$ does not absorb too much of ϵ due to shrinkage and selection, in other words, regularization reduces the effective model complexity or capacity.

Bias and variance. For orthonormal $\mathbf{X}$, the testing error

$$\text{E}\|\mathbf{X}\beta_{\text{true}} + \tilde{\epsilon} - \mathbf{X}\hat{\beta}_\lambda\|^2 = \text{E}\|\hat{\beta}_\lambda - \beta_{\text{true}}\|^2 + n\sigma^2,$$

and the first term is the mean squared error or estimation error. We can decompose the mean squared error

$$\text{E}[\|\hat{\beta}_\lambda - \beta_{\text{true}}\|^2] = \|\text{E}(\hat{\beta}_\lambda) - \beta_{\text{true}}\|^2 + \text{Var}(\hat{\beta}_\lambda) = \text{bias}^2 + \text{variance}.$$

By introducing some bias into $\hat{\beta}_\lambda$ via shrinkage and selection, we may reduce the variance, thus reducing the mean squared error and testing error. In fact, for the ridge or shrinkage estimator, the Stein's estimator

$$\hat{\beta}_{\text{Stein}} = \hat{\beta}\left(1 - \frac{(p-2)\sigma^2}{\|\hat{\beta}\|^2}\right)$$

has smaller mean squared error than the least squares estimator $\hat{\beta}$ for any β_{true} as long as $p \geq 3$.

Take-home message. Stronger regularization (big λ) means:

(1) smaller effective model complexity or model capacity

(2) bigger bias, smaller variance

(3) bigger training error, smaller overfitting = testing error - training error.

We want

(1) small testing error = training error + overfitting.

(2) small mean square error = bias^2 + variance.

We can tune λ by cross-validation. An over-regularized model may become too dumb, but an under-regularized model may grow superstitious.

2 Latent factor models based on linear regression

2.1 Supervised and unsupervised learning

Regression is a typical example of supervised learning, where for each training example $X_i = (x_{ij}, j = 1, ..., p)^\top$, we observe y_i. We want to learn to predict y_i based on X_i.

obs	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1	$X_1^\top$	y_1
2	$X_2^\top$	y_2
...		
n	$X_n^\top$	y_n

obs	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1	$X_1^\top$	?
2	$X_2^\top$	?
...		
n	$X_n^\top$	?

In unsupervised learning, we only observed $\{X_i, i = 1, ..., n\}$ as independent training examples, without observing the corresponding $\{y_i, i = 1, ..., n\}$, i.e., there is no supervision in terms of $\{y_i\}$. We want to figure out the hidden structure and pattern in $\{X_i\}$ without supervision.

2.2 Factor analysis

In factor analysis, we assume the following data frame:

obs	$\mathbf{Z}_{n\times d}$	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1	$Z_1^\top$	$X_1^\top$	?
2	$Z_2^\top$	$X_2^\top$	?
...			
n	$Z_n^\top$	$X_n^\top$	?

Since $\mathbf{Y}$ is unobserved, instead of regressing $\mathbf{Y}$ on $\mathbf{X}$, we regress $\mathbf{X}$ on $\mathbf{Z}$, where $\mathbf{Z}$ is unobserved and to be inferred from $\mathbf{X}$.

The factor analysis model assumes that

$$X_i = WZ_i + \epsilon_i,\ Z_i \sim \mathrm{N}(0, \mathbf{I}_d),\ \epsilon_i \sim \mathrm{N}(0, \sigma^2\mathbf{I}_p),$$

where $X_i = (x_{ij}, j = 1, ..., p)^\top$ is p-dimensional vector for $i = 1, ..., n$, $Z_i = (z_{ik}, k = 1, ..., d)$ is d-dimensional vector of latent factors or hidden sources, $d \ll p$. Each z_{ik} is called a factor or a source.

W is $p \times d$. We can interpret W in three different ways:

(1) Loading matrix. Let $w_j^\top$ be the j-th row of W. Then

$$x_{ij} \approx \langle w_j, Z_i\rangle = \sum_{k=1}^{d} w_{jk}Z_{ik},$$

that is, x_{ij} is a loading of the factors in Z_i, and w_{jk} tells us how much of z_{ik} is to be loaded into x_{ij}

(2) Basis vectors. Let W_k be the k-th column of W. Then

$$X_i \approx \sum_{k=1}^{k} W_k z_{ik},$$

that is, X_i is the linear superposition of the basis vectors W_k, with z_{ik} being the coefficient or coordinate. This representation can be mapped to the linear regression $Y \approx \sum_{j=1}^{p} \mathbf{X}_j\beta_j$.

(3) Matrix factorization. Let $\mathbf{X}$ be the $n \times p$ matrix whose i-th row is $X_i^\top$. Let $\mathbf{Z}$ be the $n \times d$ matrix whose i-th row is $Z_i^\top$. Then

$$\mathbf{X} \approx \mathbf{Z}W^\top.$$

The model can be viewed in terms of two regressions:

(1) Each observation i contains a regression (W, X_i, Z_i), which can be mapped to $(\mathbf{X}, \mathbf{Y}, \beta)$.

(2) All the observations for a regression $(\mathbf{Z}, \mathbf{X}, W)$, which can be mapped to $(\mathbf{X}, \mathbf{Y}, \beta)$, except that the response $\mathbf{X}$ in $(\mathbf{Z}, \mathbf{X}, W)$ is p-dimensional, and the coefficient W is a matrix of $p \times d$.

2.3 Alternating least squares

The joint log probability density is

$$-\sum_{i=1}^{n} \left[\frac{1}{2\sigma^2} \|X_i - WZ\|^2 + \frac{1}{2} \|Z_i\|^2 \right],$$

up to an additive constant. We may maximize it over W and $\{Z_i\}$ by iterating the following two steps:

(1) Given W, infer Z_i for each i, which is a Bayesian version of ridge regression

$$\hat{Z}_i = (W^\top W + \sigma^2 \mathbf{I}_d)^{-1} W^\top X_i.$$

Specifically, let

$$A = \begin{bmatrix} W^\top W + \sigma^2 \mathbf{I}_d & W^\top X_i \\ X_i^\top W & X_i^\top X_i \end{bmatrix},$$

we can sweep A to obtain the quantities in $[Z_i | X_i, W] \sim \mathrm{N}(\hat{Z}_i, V_i)$, where $V_i = (W^\top W/\sigma^2 + \mathbf{I}_d)^{-1}$.

If we map this regression to the notation of the previous chapter, it should be $(W, X_i, Z_i) \to (\mathbf{X}, \mathbf{Y}, \beta)$ for each i.

(2) Given $\{Z_i\}$, learn W by multivariate regression of $\{X_i\}$ on $\{Z_i\}$, specifically, let

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n Z_i Z_i^\top & \sum_{i=1}^n Z_i X_i^\top \\ \sum_{i=1}^n X_i Z_i^\top & \sum_{i=1}^n X_i^\top X_i \end{bmatrix} = \begin{bmatrix} \mathbf{Z}^\top \mathbf{Z} & \mathbf{Z}^\top \mathbf{X} \\ \mathbf{X}^\top \mathbf{Z} & \mathbf{X}^\top \mathbf{X} \end{bmatrix},$$

we can estimate $W = B_{11}^{-1} B_{12}$. If σ^2 is unknown, we can estimate it as the average of the diagonal elements of $\tilde{B}_{22}$, where $\tilde{B}$ is the matrix after sweeping B. For the Z_i in Step (2), we can plug in $\hat{Z}_i$ obtained in Step (1).

If we map this regression to the notation of the previous chapter, it should be $(\mathbf{Z}, \mathbf{X}, W) \to (\mathbf{X}, \mathbf{Y}, \beta)$. The difference is the $\mathbf{Y}$ has one column, but $X^\top$ has p columns. As a result, β is a vector, but W is a matrix.

Step 1 minimizes the objective function $\|X_i - WZ\|^2/2\sigma^2 + \|Z_i\|^2/2$ over Z_i given W. Step 2 minimizes the objective function over W given $\{Z_i\}$.

2.4 EM algorithm as multiple imputations

The EM algorithm amounts to a simple modification of the above algorithm, where we calculate

$$\mathrm{E}[Z_i Z_i^\top | Y_i, W] = \hat{Z}_i \hat{Z}_i^\top + V,$$

because $[Z_i | Y_i, W] \sim \mathrm{N}(\hat{Z}_i, V)$. This amounts to impute each Z_i by multiple guesses from the posterior distribution $\mathrm{N}(\hat{Z}_i, V)$, instead of a single best guess $\hat{Z}_i$. Such a multiple imputation accounts for the uncertainty in inferring Z_i. More specifically, we let $Z_i = \hat{Z}_i + \epsilon_i$, where $\epsilon_i \sim \mathrm{N}(0, V)$ accounts for uncertainties in different imputations. Then $\mathrm{E}[Z_i Z_i^\top] = \hat{Z}_i \hat{Z}_i^\top + \mathrm{E}[\epsilon_i \epsilon_i^\top] = \hat{Z}_i \hat{Z}_i^\top + V$.

R code:

```
n = 1000
p = 5
d = 2
sigma = 1.
IT = 1000
W_true = matrix(rnorm(d*p), nrow=p)
Z_true = matrix(rnorm(n*d), nrow=d)
epsilon = matrix(rnorm(p*n)*sigma, nrow=p)
X = W_true%*%Z_true + epsilon

sq = 1.;
XX = X%*%t(X)
W = matrix(rnorm(p*d)*.1, nrow=p)
for (it in 1:IT)
{
    A = rbind(cbind(t(W)%*%W/sq+diag(d), t(W)/sq), cbind(W/sq, diag(p)))
    AS = mySweep(A, d)
    alpha = AS[1:d, (d+1):(d+p)]
    D = -AS[1:d, 1:d]
    Zh = alpha %*% X
    ZZ = Zh %*% t(Zh) + D*n
    B = rbind(cbind(ZZ, Zh%*%t(X)), cbind(X%*%t(Zh), XX))
    BS = mySweep(B, d)
    W = t(BS[1:d, (d+1):(d+p)])
    sq = mean(diag(BS[(d+1):(d+p), (d+1):(d+p)]))/n;
    sq1 = mean((X-W%*%Zh)^2)
    print(cbind(sq, sq1))
}
print(W)
```

Another view of the EM algorithm for factor analysis is as follows. Consider the B matrix

$$\begin{aligned} B &= \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} \frac{1}{n}\sum_{i=1}^n Z_i Z_i^\top & \frac{1}{n}\sum_{i=1}^n Z_i X_i^\top \\ \frac{1}{n}\sum_{i=1}^n X_i Z_i^\top & \frac{1}{n}\sum_{i=1}^n X_i^\top X_i \end{bmatrix} \\ &\to \begin{bmatrix} \mathrm{E}(Z_i Z_i^\top) & \mathrm{E}(Z_i X_i^\top) \\ \mathrm{E}(X_i Z_i^\top) & \mathrm{E}(X_i X_i^\top) \end{bmatrix} = \begin{bmatrix} \mathbf{I}_d & W^\top \\ W & WW^\top + \sigma^2 \mathbf{I}_d \end{bmatrix}, \end{aligned}$$

where we divide the sums by $1/n$, and the resulting averages converge to their expectations. We can use the sweep operator on the limiting or theoretical B to get the posterior distribution of $[Z_i|X_i, W] \sim \mathrm{N}(\alpha X_i, V)$.

R code:

```
n = 1000
p = 5
d = 2
sigma = 1.
IT = 100
beta_true = matrix(rnorm(d*p), nrow=p)
Z = matrix(rnorm(n*d), nrow=n)
epsilon = matrix(rnorm(n*p)*sigma, nrow=n)
X = Z%*%t(beta_true) + epsilon

XX = t(X)%*%X/n
beta = matrix(rnorm(d*p)*.1, nrow=p)
Sig = matrix(rnorm(p)*.1, nrow=p)
for (it in 1:IT)
{
    B = rbind(cbind(beta%*%t(beta)+diag(Sig), beta), cbind(t(beta), diag(rep(1, d))))
```

```
    BS = mySweep(B, p)
    alpha = BS[1:p, (p+1):(p+d)]
    D = BS[(p+1):(p+d), (p+1):(p+d)]
    XZ = XX %*% alpha
    ZZ = t(alpha)%*%XX%*%alpha + D
    A = rbind(cbind(ZZ, t(XZ)), cbind(XZ, XX))
    AS = mySweep(A, d)
    beta = t(AS[1:d, (d+1):(d+p)])
    Sig = diag(AS[(d+1):(d+p), (d+1):(d+p)])
}
print(beta)
print(Sig)
print(beta%*%t(beta)+diag(Sig))
print(XX)
```

2.5 Independent component analysis

The independent component analysis (ICA) is similar to the factor analysis, where $z_{ik} \sim p(z)$ for a non-Gaussian distribution $p(z)$, and we usually assume $\sigma^2 = 0$. We first linearly transform the data so that $\mathrm{E}(X_i) = 0$ and $\mathrm{Var}(X_i) = \mathbf{I}_p$. Then we apply ICA to discover the non-Gaussian structure. A popular algorithm is fast ICA.

2.6 Sparse coding

Let $\{X_i, i = 1, ..., n\}$ be a set of $p \times 1$ observed vectors, such as image patches. We want to encode $\{X_i\}$ by a dictionary of basis vectors $W = (W_k, k = 1, ..., d)$. The dictionary is over-complete or redundant in the sense that $d > p$. But the coding is supposed to be sparse, in the sense that for each X_i, we only need to select a small number of basis vectors from the dictionary to code it by sparse linear regression. But for different X_i, we may select different subsets of basis vectors, i.e.,

$$X_i = \sum_{k=1}^{d} W_k z_{ik} + \epsilon_i = W Z_i + \epsilon_i,$$

where $Z_i = (z_{ik}, k = 1, ..., d)^\top$ is the sparse coefficient vector for encoding X_i, and ϵ_i is the error vector.

We can find W by minimizing

$$\sum_{i=1}^{n} \left[\|X_i - W Z_i\|_{\ell_2}^2 / 2 + \lambda \|Z_i\|_{\ell_1} \right].$$

The computation can be accomplished by alternating gradient descent, which iterates the following two steps: (1) Given W, gradient descent on Z_i for each X_i. (2) Given $\{Z_i\}$, gradient descent on W.

2.7 Matrix factorization and completion

Consider a recommender system, where x_{ij} is the rating of user i on item j, where $i = 1, ..., n$, and $j = 1, ..., p$. Let z_{ik}, $k = 1, ..., d$ be the desire (or zeal for "z") of user i in the k-th aspect, and w_{jk}, $k = 1, ..., d$ be the desirability (or worth for "w") of item j in the k-th aspect. We can model

$$x_{ij} \approx \sum_{k=1}^{d} z_{ik} w_{jk} = \langle Z_i, W_j \rangle,$$

where $Z_i = (z_{ik}, k = 1, ..., d)^\top$ and $W_j = (w_{jk}, k = 1, ..., d)^\top$.

We can estimate Z_i and W_j by minimizing

$$\sum_{i,j}(x_{ij} - \langle Z_i, W_j\rangle)^2 + \lambda_1 \sum_i \|Z_i\|^2 + \lambda_2 \sum_j \|W_j\|^2,$$

where the sum is over (i, j) that x_{ij} is observed. The computation can be accomplished by alternating least squares or more precisely alternating ridge regressions.

Let $\mathbf{X} = (x_{ij})$ be the matrix of ratings. Let $\mathbf{Z} = (Z_i, i = 1, ..., n)^\top$. Let $W = (W_j, j = 1, ..., p)$. Then $\mathbf{X} \approx \mathbf{Z}W^\top$. Here the treatment of Z_i and W_j are more symmetric than factor analysis and sparse coding.

R code:

```
n = 200
p = 100
d = 3
sigma = .1
prob = .2
IT = 100
lambda = .1

W_true = matrix(rnorm(p*d), nrow = p)
Z_true = matrix(rnorm(n*d), nrow = d)
epsilon = matrix(rnorm(p*n)*sigma, nrow=p)
X = W_true%*%Z_true + epsilon

R = matrix(runif(p*n)<prob, nrow = p)
W = matrix(rnorm(p*d)*.1, nrow = p)
Z = matrix(rnorm(n*d)*.1, nrow = d)

for (it in 1:IT)
{
  for (i in 1:n)
  {
    WW = t(W)%*%diag(R[,i])%*%W+lambda*diag(d)
    WX = t(W)%*%diag(R[,i])%*%X[,i]
    A = rbind(cbind(WW, WX), cbind(t(WX), 0))
    AS = mySweep(A, d)
    Z[,i] = AS[1:d, d+1]
  }
  for (j in 1:p)
  {
    ZZ = Z%*%diag(R[j, ])%*%t(Z)+lambda*diag(d)
    ZX = Z%*%diag(R[j,])%*%X[j,]
    B = rbind(cbind(ZZ, ZX), cbind(t(ZX), 0))
    BS = mySweep(B, d)
    W[j,] = BS[1:d, d+1]
  }
  sd1 = sqrt(sum(R*(X-W%*%Z)^2)/sum(R))
  sd0 = sqrt(sum((1.-R)*(X-W%*%Z)^2)/sum(1.-R))
  print(cbind(sd1, sd0))
}
```

2.8 Non-negative matrix factorization

In non-negative matrix factorization or positive factor analysis, $\mathbf{X} \approx \mathbf{Z}W^\top$, but we assume that $z_{ik} \geq 0$ for all i and k. It can be learned by iterated constrained least squares.

2.9 QR decomposition

We have been using the sweep operator to power the least squares computation. We can also use QR decomposition, which does not require the computation of the cross-product matrix such as $\mathbf{X}^\top\mathbf{X}$.

QR decomposition is to decompose a matrix $\mathbf{X}$ into a product $\mathbf{X} = QR$ where Q is an orthogonal matrix and R is an upper triangular matrix.

2.10 Orthogonal matrix

Let $Q = (q_1, q_2, \ldots, q_n)$ be an orthogonal matrix, $Q^\top Q = QQ^\top = I$, then Q forms an orthogonal basis:

(1) For each vector q_i, $\|q_i\| = 1$.

(2) For any two different vectors q_i and q_j, $\langle q_i, q_j\rangle = 0$, i.e., $q_i \perp q_j$.

For any vector v, we have

(1) Analysis: $u_i = \langle v, q_i\rangle = q_i^\top v$ is the coordinate of v on the axis q_i, for $i = 1, ..., n$, i.e., $u = Q^\top v$.

(2) Synthesis: $v = \sum_{i=1}^n q_i u_i = Qu$.

From (1) and (2), Q and $Q^\top$ are inverse of each other.

2.11 Householder reflections

To obtain a QR decomposition, we can apply the Householder reflections repeatedly. Given an $n \times p$ matrix $\mathbf{X}$, as the first step, we want to find an orthogonal transformation H_1 such that only the first element in the first column is non-zero after the transformation:

$$\begin{bmatrix} x_{11} & x_{12} & \ldots & y_1 \\ x_{21} & x_{22} & \ldots & y_2 \\ \ldots & \ldots & \ldots & \ldots \\ x_{n1} & x_{n2} & \ldots & y_n \end{bmatrix} \xrightarrow{H_1} \begin{bmatrix} x_{11}^* & x_{12}^* & \ldots & y_1^* \\ 0 & x_{22}^* & \ldots & y_2^* \\ \ldots & \ldots & \ldots & \ldots \\ 0 & x_{n2}^* & \ldots & y_n^* \end{bmatrix}$$

Since the orthogonal transformation preserves the length of vectors, we know

$$|\mathbf{X}_1^*| = |\mathbf{X}_1| = \sqrt{x_{11}^2 + x_{12}^2 + \ldots + x_{1n}^2},$$

which means the value of x_{11^*} is determined by

$$x_{11}^* = \pm |\mathbf{X}_1| .$$

The sign of x_{11}^* is chosen as the opposite of x_{11} for numerical stability.

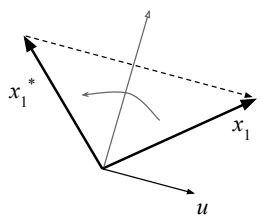

Figure 3: Household Reflection

To find a transformation H which can rotate the vector $\mathbf{X}_1$ to $\mathbf{X}_1^*$, one simple way is to construct an isosceles triangle where $\mathbf{X}_1^*$ is a reflection of $\mathbf{X}_1$:

$$\mathbf{X}_1^* = \mathbf{X}_1 - 2\langle \mathbf{X}_1, u\rangle u = \mathbf{X}_1 - 2uu^\top \mathbf{X}_1 = H_1 \mathbf{X}_1,$$

where

$$u = \frac{\mathbf{X}_1 - \mathbf{X}_1^*}{|\mathbf{X}_1 - \mathbf{X}_1^*|}, \quad H_1 = I - 2uu^\top.$$

We apply the Householder reflection on all the column vectors of $\mathbf{X}$, i.e., $\mathbf{X} \leftarrow H_1\mathbf{X}$. In implementing $H_1\mathbf{X}$, we let $\mathbf{X}_j \leftarrow \mathbf{X}_j - 2\langle \mathbf{X}_j, u\rangle u$, for $j = 1, ..., p$. The notation $H_1\mathbf{X}$ is only for theoretical understanding. In real computation, we do not form the matrix H_1 and compute $H_1\mathbf{X}$ by matrix computation. We also apply H_1 to $\mathbf{Y}$.

Let $\mathbf{X}^{(1)}$ be the sub-matrix of the resulting $\mathbf{X}$ with the first row and the first column removed from $\mathbf{X}$. We continue to apply the Householder reflection on the sub-matrix $\mathbf{X}^{(1)}$, while maintaining the first row and the first column of $\mathbf{X}$. This amounts to left multiplying $\mathbf{X}$ by an orthogonal matrix H_2.

We can keep going until we change $\mathbf{X}$ into an upper triangular matrix. Let $H = H_p \dots H_2 H_1$, we have $H\mathbf{X} = R$. Let $Q = H^\top$, we obtain the QR decomposition $\mathbf{X} = QR$. In order to obtain H, we can apply the transformations $H_p \dots H_2 H_1$ to the identity matrix in the above process.

Python code:

```
import numpy as np
from scipy import linalg

def qr(A):
    n, m = A.shape
    R = A.copy()
    Q = np.eye(n)

    for k in range(m-1):
        x = np.zeros((n, 1))
        x[k:, 0] = R[k:, k]
        v = x
        v[k] = x[k] + np.sign(x[k,0]) * np.linalg.norm(x)

        s = np.linalg.norm(v)
        if s != 0:
            u = v / s
            R -= 2 * np.dot(u, np.dot(u.T, R))
            Q -= 2 * np.dot(u, np.dot(u.T, Q))
    Q = Q.T
    return Q, R
```

2.12 Linear regression by QR

We rotate the matrix $(\mathbf{XY})$ by QR decomposition, by applying the Householder reflections for $j = 1, ..., p$,

$$\begin{bmatrix}\mathbf{X} & \mathbf{Y}\end{bmatrix} \xrightarrow{Q^\top} \begin{bmatrix}R & \mathbf{Y}^*\end{bmatrix} = \begin{bmatrix}R_1 & \mathbf{Y}_1^* \\ 0 & \mathbf{Y}_2^*\end{bmatrix},$$

where R_1 is a upper triangular squared matrix.

To solve the least squares problem,

$$\min_\beta \|\mathbf{Y}^* - R\beta\|^2 = \min_\beta \left(\|Y_1^* - R_1\beta\|^2 + \|\mathbf{Y}_2^*\|^2\right),$$

the solution $\hat{\beta} = R_1^{-1}\mathbf{Y}_1^*$ and RSS $= \|\mathbf{Y}_2^*\|^2$.

Since R_1 is an upper triangular matrix, we can solve the elements of $\hat{\beta}$ in reverse order $\hat{\beta}_p, \hat{\beta}_{p-1}, \ldots, \hat{\beta}_1$. It is numerically stable and efficient.

Python code:

```
n = 100
p = 5
X = np.random.random_sample((n, p))
beta = np.array(range(1, p+1))
Y = np.dot(X, beta) + np.random.standard_normal(n)

Z = np.hstack((np.ones(n).reshape((n, 1)), X, Y.reshape((n, 1))))
_, R = qr(Z)
R1 = R[:p+1, :p+1]
Y1 = R[:p+1, p+1]
beta = np.linalg.solve(R1, Y1)
print beta
```

2.13 Eigen decomposition and diagonalization

A $p \times p$ symmetric matrix Σ can be diagonalized by $\Sigma = Q\Lambda Q^\top$, where Q is an orthogonal matrix, and Λ is a diagonal matrix, $\Lambda = \text{diag}(\lambda_1, \lambda_2, ..., \lambda_p)$, where we order λ_j from largest to smallest in magnitude for $j = 1, ..., p$. $\Sigma Q = Q\Lambda$, so $\Sigma q_j = \lambda_j q_j$. The column vectors in Q and the diagonal elements in Λ are eigenvectors and eigenvalues of Σ.

2.14 Power method

For a vector v, let u be its coordinates in system Q, i.e. $v = Qu$, then

$$\Sigma v = Q\Lambda Q^\top Qu = Q \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_p \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_p \end{bmatrix} = Q \begin{bmatrix} \lambda_1 u_1 \\ \lambda_2 u_2 \\ \vdots \\ \lambda_p u_p \end{bmatrix},$$

which means the vector Σv becomes $(\lambda_1 u_1, \lambda_2 u_2, \cdots, \lambda_p u_p)^\top$ in basis Q, i.e., Σ is Λ in Q.

If we repeat this process n times, then Σ^n is Λ^n in Q,

$$v \xrightarrow{\Sigma^n} (\lambda_1^n u_1, \lambda_2^n u_2, \cdots, \lambda_p^n u_p)^\top.$$

We can keep normalizing $v \leftarrow v/|v|$ to make v a unit vector in this process.

Suppose λ_1 has the greatest magnitude, this procedure will converge to $u = (1, 0, \cdots, 0)$ in the space of Q, and the corresponding $v = q_1$.

The power method iterates the following two steps:

- Compute normalized vector $\tilde{v} = \dfrac{v}{|v|}$.
- Update $v = \Sigma\tilde{v}$.

To get q_2 using this method, we initialize the above procedure with a vector $v \perp q_1$ that is perpendicular to q_1. In Q, the first component of u will always be 0, then the procedure will converge to $u = (0, 1, 0, \cdots, 0)$ in the space of Q, and the corresponding $v = q_2$.

To get q_3, we initialize the above procedure with v perpendicular to both q_1 and q_2, i.e. $v \perp q_1$ and $v \perp q_2$.

Continue the above procedure, we eventually get all the vectors in Q.

2.15 Matrix form of power method

We can parallelize the above sequential method, by starting from p vectors $V = (V_1, ..., V_p)$ and maintain their orthogonality after each multiplication by Σ, by iterating the following two step

- Compute $\tilde{V}$, the orthogonalized V.
- Update $V = \Sigma\tilde{V}$.

2.16 QR for Gram-Schimdt

We use QR decomposition to perform orthogonalization, i.e., let $V = QR$, and let $\tilde{V} = Q$. The QR method is to perform Gram-Schimdt orthogonalization. Specifically, $V_1 = r_{11}q_1$, i.e., q_1 is the normalized version of V_1. $V_2 = r_{12}q_1 + r_{22}q_2$, i.e., q_2 is obtained by projecting V_1 on q_1, and get the reminder $r_{22}q_2$, and then normalize this remainder vector, and so on.

Python code:

```
def eigen_qr(A):
    T = 1000
    A_copy = A.copy()
    r, c = A_copy.shape

    V = np.random.random_sample((r, r))

    for i in range(T):
        Q, _ = qr(V)
        V = np.dot(A_copy, Q)

    Q, R = qr(V)

    return R.diagonal(), Q
```

2.17 Principal component analysis

Consider the $n \times p$ data matrix $\mathbf{X}$. Let us assume that all the columns of $\mathbf{X}$ are centralized, i.e., $\sum_{i=1}^{n} x_{ij}/n = 0$. Otherwise, we can subtract the column average from each element of the column. In other words, let $\mathbf{1}$ be the $n \times 1$ column vector of 1's. Then $\langle \mathbf{X}_j, \mathbf{1} \rangle = 0$, for $j = 1, ..., p$, i.e., $\mathbf{1}^\top \mathbf{X} = 0$.

For each row of $\mathbf{X} = (X_1^\top, ..., X_n^\top)^\top$, we want to represent observation X_i in a new basis system Q, so that $X_i = QZ_i$. Let $\mathbf{Z} = (Z_1^\top, ..., Z_n^\top)^\top$ be the data matrix in Q. We want the columns of $\mathbf{Z} = (\mathbf{Z}_1, ..., \mathbf{Z}_p)$ to be orthogonal to each other, so that they are uncorrelated, i.e., if you regress any column of $\mathbf{Z}$ on another column of $\mathbf{Z}$, the regression coefficient is 0. $\mathbf{X} = \mathbf{Z}Q^\top$. Let $\lambda_j = \|\mathbf{Z}_j\|^2/n = \sum_{i=1}^{n} z_{ij}^2/n$, then λ_j is the variance of $\{z_{ij}, i = 1, ..., n\}$, and $\mathbf{Z}^\top\mathbf{Z} = \Lambda = \text{diag}(\lambda_1, ..., \lambda_p)$. Then

$$\mathbf{X}^\top\mathbf{X} = Q\mathbf{Z}^\top\mathbf{Z}Q^\top = Q\Lambda Q^\top.$$

We can use the power method to compute Q and Λ.

We can choose $d < p$, and represent $X_i \approx \sum_{k=1}^{d} z_{ik} q_k$. This is principal component analysis for dimension reduction.

Python code:

```
n = 100
p = 5
X = np.random.random_sample((n, p))
A = np.dot(X.T, X)

D, V = eigen_qr(A)
print D.round(6)
print V.round(6)

# Compare the result with the numpy calculation
eigen_value_gt, eigen_vector_gt = np.linalg.eig(A)
print eigen_value_gt.round(6)
print eigen_vector_gt.round(6)
```

2.18 Singular value decomposition

Let $\tilde{\mathbf{Z}}_j = \mathbf{Z}_j/\lambda_j$, then $\mathbf{X} = \tilde{\mathbf{Z}}\Lambda Q$, where $\tilde{\mathbf{Z}}^\top \tilde{\mathbf{Z}} = \mathbf{I}_p$. This is singular value decomposition.

2.19 PCA and factor analysis

PCA is similar to factor analysis, with $(q_1, ..., q_d)$ correspond to $(W_1, ..., W_d)$. But in factor analysis, W is not assumed to be orthogonal.

3 Classification based on generalized linear regression

3.1 Logistic regression and 0/1 outcome

Consider a dataset with n training examples, where $X_i^\top = (x_{i1}, \cdots, x_{ip})$ consists of p predictors or features, $y_i \in \{0, 1\}$ is the outcome or class label.

obs	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1	$X_1^\top$	y_1
2	$X_2^\top$	y_2
...		
n	$X_n^\top$	y_n

We assume $y_i \sim \text{Bernoulli}(p_i)$, i.e., $\Pr(y_i = 1) = p_i$, and we assume

$$\text{logit}(p_i) = \log \frac{p_i}{1 - p_i} = X_i^\top \beta.$$

Let $\eta_i = X_i^\top \beta$ be the score, then

$$p_i = \sigma(\eta_i) = \frac{1}{1 + e^{-\eta_i}} = \frac{1}{1 + e^{-X_i^\top \beta}} = \frac{e^{X_i^\top \beta}}{1 + e^{X_i^\top \beta}},$$

where the function $\sigma()$ is the sigmoid function, which is the inverse of the logit function.

3.2 Maximum likelihood

The likelihood function is

$$L(\beta) = \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-y_i} = \prod_{i=1}^{n} \frac{e^{y_i X_i^\top \beta}}{1+e^{X_i^\top \beta}}.$$

The log-likelihood is

$$l(\beta) = \log L(\beta) = \sum_{i=1}^{n} \left[y_i X_i^\top \beta - \log(1+\exp X_i^\top \beta) \right].$$

The maximum likelihood is to find the most plausible explanation to the observed data.

3.3 Gradient ascent

To find the maximum of $l(\beta)$, we first calculate the gradient

$$l'(\beta) = \sum_{i=1}^{n} \left[y_i X_i - \frac{e^{X_i^\top \beta}}{1+e^{X_i^\top \beta}} X_i \right] = \sum_{i=1}^{n} (y_i - p_i) X_i.$$

We use gradient ascent to iteratively update β,

$$\beta^{(t+1)} = \beta^{(t)} + \gamma_t \sum_{i=1}^{n} (y_i - p_i) X_i,$$

where γ is the learning rate. This is a hill climbing algorithm, where each step we take the steepest direction uphill. If we minimize a loss function such as $-l(\beta)$, then we use the gradient descent algorithm, which means each step we take the steep direction downhill.

R code:

```
myLogistic <- function(X_train, Y_train, X_test, Y_test,
num_iterations = 500, learning_rate = 1e-1)
{
  n     <- dim(X_train)[1]
  p     <- dim(X_train)[2]+1
  ntest <- dim(X_test)[1]

  X_train1 <- cbind(rep(1, n), X_train)
  X_test1  <- cbind(rep(1, ntest), X_test)

  sigma <- .1
  beta  <- matrix(rnorm(p)*sigma, nrow=p)

  acc_train <- rep(0, num_iterations)
  acc_test  <- rep(0, num_iterations)
  for(it in 1:num_iterations)
  {
    pr    <- 1/(1 + exp(-X_train1 %*% beta))
    dbeta <- matrix(rep(1, n), nrow = 1) %*%((matrix(Y_train - pr, n, p)*X_train1))/n
    beta  <- beta + learning_rate * t(dbeta)

    prtest <- 1/(1 + exp(-X_test1 %*% beta))
    acc_train[it] <- accuracy(pr, Y_train)
    acc_test[it]  <- accuracy(prtest, Y_test)
```

```
    print(c(it, acc_train[it], acc_test[it]))
 }
  output <- list(beta = beta, acc_train = acc_train, acc_test = acc_test)
  output
}
```

}

3.4 Learning from mistakes

The algorithm learns from mistakes by trial and error. If a mistake is made such that p_i is very different from y_i, then β accumulates X_i if y_i is positive, and $-X_i$ is y_i is negative.

3.5 Classification, $\pm$ outcome, and logistic loss

We want to learn β either for the purpose of explanation or understanding, or for the purpose of classification or prediction. In the context of classification, we usually let $y_i \in \{+1, -1\}$ instead of $y_i \in \{1, 0\}$. Those X_i with $y_i = +$ are called positive examples, and those $X_i = -$ are called negative examples.

We may call β a classifier. We can think of $\langle X_i, \beta \rangle$ as projection of X_i on the vector β, so the vector β is the direction that reveals the difference between positive X_i and negative X_i, thus β should be aligned with positive X_i and negatively aligned with negative X_i.

$\Pr(y_i = +1) = 1/(1 + \exp(-X_i^\top \beta))$, and $\Pr(y_i = -1) = 1/(1 + \exp(X_i^\top \beta))$. Thus $p(y_i) = 1/(1 + \exp(-y_i X_i \beta)$. Let the loss function $loss(\beta)$ be the negative log-likelihood, then

$$loss(\beta) = \sum_{i=1}^{n} \log[1 + \exp(-y_i X_i^\top \beta)].$$

The gradient

$$loss'(\beta) = -\sum_{i=1}^{n} \sigma(-y_i X_i^\top \beta) y_i X_i.$$

The gradient descent algorithm is

$$\beta^{(t+1)} = \beta^{(t)} + \gamma \sum_{i=1}^{n} \sigma(-y_i X_i^\top \beta) y_i X_i.$$

3.6 Score and margin

For (X_i, y_i), $X_i^\top \beta$ is the score. If the score is positive, we classify y_i to be positive. If the score is negative, we classify y_i to be negative.

The term $y_i X_i^\top \beta$ is the margin for this observation. If the margin is large, it means the classification is confident. If the margin is small, it means the classification is uncertain. If the margin is negative, it means β makes a mistake. The more negative it is, the bigger the mistake is.

3.7 Perceptron

A deterministic version of the logistic regression is the perceptron model $y_i = \text{sign}(X_i^\top \beta)$, where $y_i \in \{+1, -1\}$ (instead of $\{0, 1\}$), and $\text{sign}(\eta) = +1$ if $\eta \geq 0$, and $\text{sign}(\eta) = -1$ if $\eta < 0$. The gradient learning algorithm can be modified into the perceptron algorithm. Starting from $\beta_0 = 0$,

$$\beta^{(t+1)} = \beta^{(t)} + \sum_{i=1}^{n} \delta_i y_i X_i,$$

where $\delta_i = 1(y_i \neq \text{sign}(X_i^\top \beta^{(t)}))$ to determine whether $\beta^{(t)}$ makes a mistake in classifying y_i. We can compare δ_i with $\sigma(-y_i X_i^\top \beta)$ in logistic regression.

The algorithm can be considered the gradient descent algorithm for the loss function

$$loss(\beta) = \sum_{i=1}^{n} \max(0, -y_i X_i^\top \beta) = \sum_{i=1}^{n} \max(0, -margin_i),$$

where $\max(0, -margin_i) = 0$ if $margin_i \geq 0$, i.e., no mistake is made, and $\max(0, -margin_i) = -margin_i$ if $margin_i < 0$. Again the algorithm learns from the mistakes.

3.8 Newton-Raphson for solving equation

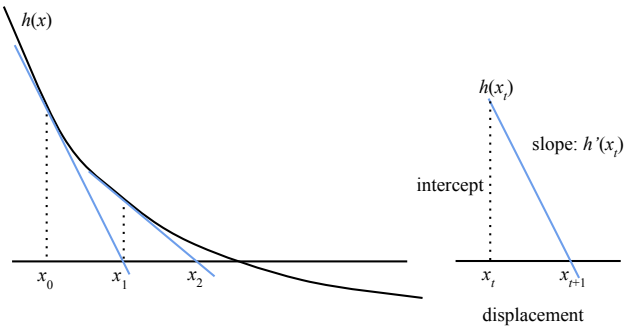

Figure 4: Newton-Raphson

A more efficient ways is to update β using Newton-Raphson method. Suppose we want to solve $h(x) = 0$. At x_t, we take the first order Taylor expansion

$$h(x) \doteq h(x_t) + h'(x_t)(x - x_t).$$

Each iteration, we find the root of the linear surrogate function, which is the above first order Taylor expansion,

$$x_{t+1} = x_t - \frac{h(x_t)}{h'(x_t)}.$$

3.9 Newton-Raphson for optimization

Suppose we want to find the mode of $f(x)$, we can solve $f'(x) = 0$. Using Newton-Raphson, we have

$$x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}.$$

Each iteration maximizes a quadratic approximation to the original function at x_t,

$$f(x) \doteq f(x_t) + f'(x_t)(x - x_t) + \frac{1}{2} f''(x_t)(x - x_t)^2.$$

$f''(x_t)$ is the curvature of f at x_t. If the curvature is big, the step size should be small. If the curvature is small, the step size can be made larger.

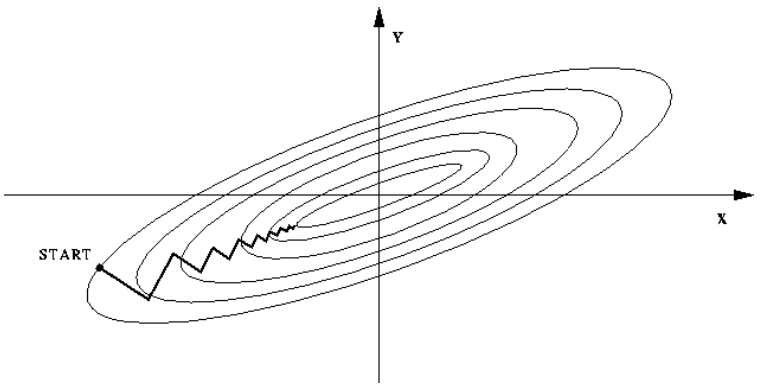

Figure 5: Gradient descent, source web.

3.10 Multivariate Newton-Raphson

If the variable is a vector $x = (x_1, x_2, \cdots, x_n)^\top$, let

$$f'(x) = \left(\frac{\partial f}{\partial x_i}\right)_{n\times 1} \qquad f''(x) = \left(\frac{\partial f}{\partial x_i \partial x_j}\right)_{n\times n},$$

$f''(x_t)$ is called the Hessian matrix, we have

$$x_{t+1} = x_t - f''(x_t)^{-1} f'(x_t).$$

$f''(x_t)$ tells us the local shape of f around x_t. $f''(x_t)^{-1} f'(x_t)$ gives us better direction than $f'(x_t)$ as shown in the above figure. The Newton-Raphson is a second order algorithm.

3.11 Iterated reweighed least squares

For maximum likelihood estimate of β in logistic regression, the second derivative of the log likelihood function

$$l''(\beta) = -\sum_{i=1}^{n} p_i(1-p_i) X_i X_i^\top.$$

We can update β by

$$\beta^{(t+1)} = \beta^{(t)} + l''(\beta^{(t)})^{-1} l'(\beta^{(t)}).$$

Let $w_i = p_i(1-p_i)$, we can rewrite the update equation as

$$\begin{aligned}
\beta^{(t+1)} &= \beta^{(t)} + \left[\sum_{i=1}^{n} p_i(1-p_i) X_i X_i^\top\right]^{-1} (y_i - p_i) X_i \\
&= \left(\sum_{i=1}^{n} w_i X_i X_i^\top\right)^{-1} \left[\sum_{i=1}^{n} w_i X_i X_i^\top \beta^{(t)} + (y_i - p_i) X_i\right] \\
&= \left(\sum_{i=1}^{n} w_i X_i X_i^\top\right)^{-1} \left[\sum_{i=1}^{n} w_i X_i \left(X_i^\top \beta^{(t)} + \frac{y_i - p_i}{w_i}\right)\right].
\end{aligned}$$

Let

$$\hat{y}_i = X_i^\top \beta^{(t)} + \frac{y_i - p_i}{w_i},$$

let $\tilde{X}_i = X_i\sqrt{w_i}, \tilde{y}_i = \hat{y}_i\sqrt{w_i}$, we can rewrite the equation above as follows:

$$\begin{aligned}\beta^{(t+1)} &= \left(\sum_{i=1}^{n} w_i X_i X_i^\top\right)^{-1}\left(\sum_{i=1}^{n} w_i X_i \hat{y}_i\right) \\ &= \left(\sum_{i=1}^{n} \tilde{X}_i \tilde{X}_i{}^\top\right)^{-1}\left(\sum_{i=1}^{n} \tilde{X}_i \tilde{y}_i\right).\end{aligned}$$

The flow is

$$\beta^{(t)} \to \eta_i = X_i^\top \beta^{(t)} \to p_i = \sigma(\eta_i) \to w_i = p_i(1-p_i) \to \hat{y}_i = \eta_i + \frac{y_i - p_i}{w_i} \to \tilde{X}_i = X_i\sqrt{w_i}, \tilde{y}_i = \hat{y}_i\sqrt{w_i} \to \beta^{(t+1)}.$$

This procedure is referred to as iterated re-weighed least squares (IRLS).

Python code:

```
import numpy as np
from scipy import linalg

def mylogistic(_x, _y):
    x = _x.copy()
    y = _y.copy()
    r, c = x.shape

    beta = np.zeros((c, 1))
    epsilon = 1e-6

    while True:
        eta = np.dot(x, beta)
        pr = exp_it(eta)
        w = pr * (1 - pr)
        z = eta + (y - pr) / w
        sw = np.sqrt(w)
        mw = np.repeat(sw, c, axis=1)

        x_work = mw * x
        y_work = sw * z

        beta_new, _, _, _ = np.linalg.lstsq(x_work, y_work)
        err = np.sum(np.abs(beta_new - beta))
        beta = beta_new
        if err < epsilon:
            break

    return beta

def exp_it(_x):
    x = _x.copy()
    y = 1 / (1 + np.exp(-x))
    return y

if __name__ == '__main__':
    n = 1000
    p = 5

    X = np.random.normal(0, 1, (n, p))
    #beta = np.arange(p) + 1
    beta = np.ones((p, 1))
    print beta
```

```
Y = np.random.uniform(0, 1, (n, 1)) < exp_it(np.dot(X, beta)).reshape((n, 1))

logistic_beta = mylogistic(X, Y)
print logistic_beta
```

3.12 Neural network: multi-layer perceptrons

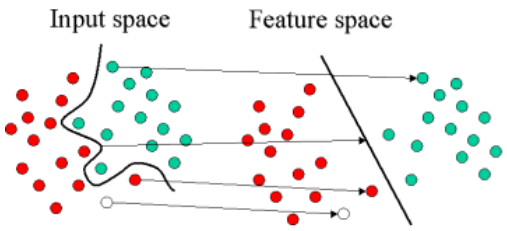

Figure 6: non-linear boundary, source web.

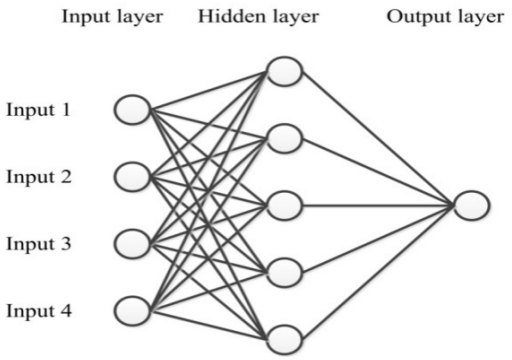

Figure 7: A two layer feedforward neural network, source web.

A perceptron seeks to separate the positive examples and negative examples by projecting them onto β, or in other words, separating them using a hyperplane. If the data are not linearly separable, a perceptron cannot work. We may need to transform the original variables into some features so that they can be linearly separated.

One way to solve this problem is to generalize the perceptron into multi-layer perceptrons. This structure is also called feedforward neural network.

3.13 Logistic regression on top of logistic regressions

obs	$\mathbf{Z}_{n\times d}$	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1	$Z_1^\top$	$X_1^\top$	y_1
2	$Z_2^\top$	$X_2^\top$	y_2
...			
n	$Z_n^\top$	$X_n^\top$	y_n

The neural network is logistic regression on top of logistic regressions. y_i follows a logistic regression on $Z_i = (z_{ik}, k =$

$1, ..., d)^\top$, and each z_{ik} follows a logistic regression on $X_i = (x_{ij}, j = 1, ..., p)^\top$,

$$
\begin{aligned}
y_i &\sim \text{Bernoulli}(p_i), \\
p_i &= \sigma(Z_i^\top \beta) = \sum_{k=1}^{d} \beta_k z_{ik}, \\
z_{ik} &= \sigma(X_i^\top \alpha_k) = \sum_{j=1}^{p} \alpha_{kj} x_{ij}.
\end{aligned}
$$

3.14 Back-propagation chain rule calculation

The log-likelihood is

$$
l(\beta, \alpha) = \sum_{i=1}^{n} \left[y_i \sum_{k=1}^{d} \beta_k z_{ik} - \log[1 + \exp(\sum_{k=1}^{d} \beta_k z_{ik})] \right].
$$

The gradient is

$$
\begin{aligned}
\frac{\partial l}{\partial \beta} &= \sum_{i=1}^{n} (y_i - p_i) Z_i, \\
\frac{\partial l}{\partial \alpha_k} &= \frac{\partial l}{\partial z_k} \frac{\partial z_k}{\partial \alpha_k} = \sum_{i=1}^{n} (y_i - p_i) \beta_k z_{ik} (1 - z_{ik}) X_i
\end{aligned}
$$

$\partial l / \partial \alpha_k$ is calculated by chain rule. The gradient descent learning algorithm again learns from mistake or error $y_i - p_i$. The chain rule back-propagates the error to assign the blame in order for β and α to update.

R code:

```
my_NN <- function(X_train, Y_train, X_test, Y_test, num_hidden = 20,
                    num_iterations = 1000, learning_rate = 1e-1)
{
  n        <- dim(X_train)[1]
  p        <- dim(X_train)[2] + 1
  ntest    <- dim(X_test)[1]
  X_train1 <- cbind(rep(1, n), X_train)
  X_test1  <- cbind(rep(1, ntest), X_test)

  alpha <- matrix(rnorm(p * num_hidden), nrow = p)
  beta  <- matrix(rnorm((num_hidden + 1)), nrow = num_hidden + 1)

  acc_train     <- rep(0, num_iterations)
  acc_test      <- rep(0, num_iterations)

  for(it in 1:num_iterations)
  {
    Z  <- 1 / (1 + exp(-X_train1 %*% alpha))
    Z1 <- cbind(rep(1, n), Z)
    pr <- 1 / (1 + exp(-Z1 %*% beta))

    dbeta <-
    beta  <- beta + learning_rate * t(dbeta)

    for(k in 1:num_hidden)
    {
      da <- (Y_train - pr)*beta[k+1]*Z[, k]*(1-Z[, k])
      dalpha <- matrix(rep(1, n), nrow = 1)%*%((matrix(da, n, p)*X_train1))/n
      alpha[, k] <- alpha[, k] + learning_rate * t(dalpha)
```

```
    }
    acc_train[it] <- accuracy(pr, Y_train)
    Ztest         <- 1/(1 + exp(-X_test1 %*% alpha))
    Ztest1        <- cbind(rep(1, ntest), Ztest)
    prtest        <- 1/(1 + exp(-Ztest1 %*% beta))
    acc_test[it]  <- accuracy(prtest, Y_test)
    cat("On iteration ", it, " the training accuracy is ", acc_train[it],
        " and the testing accuracy is ", acc_test[it], sep = "")
    }
  model <- list(alpha = alpha, beta = beta,
                acc_train = acc_train, acc_test = acc_test)
  model
}
```

3.15 Neural net and factor analysis

The data frame $(\mathbf{Z}, \mathbf{X}, \mathbf{Y})$ in neural network is similar to the data frame $(\mathbf{Z}, \mathbf{X})$ in factor analysis. In neural net, the hidden $\mathbf{Z}$ is learned in order to explain $\mathbf{Y}$. In factor analysis, the hidden $\mathbf{Z}$ is learned in order to explain $\mathbf{X}$.

3.16 Rectified linear units and linear spline

In modern neural net, the non-linearity is often the rectified linear unit $\max(0, r)$:

$$y_i \sim \text{Bernoulli}(p_i),$$
$$p_i = \sigma(Z_i^\top \beta) = \sigma(\sum_{k=1}^{d} \beta_k z_{ik}),$$
$$z_{ik} = \max(X_i^\top \alpha_k, 0) = \max(\sum_{j=1}^{p} \alpha_{kj} x_{ij}, 0).$$

Recall the linear spline model $\sum_{j=1}^{p} \beta_j \max(0, x_i - k_j)$. The neural net can be viewed a high-dimensional spline, or piecewise linear mapping. If there are many layers in the neural net, the number of linear pieces is exponential in the number of layers. It can approximate highly non-linear mapping by patching up the large number of linear pieces.

For $\max(0, r)$, we should replace $z_{ik}(1 - z_{ik})$ by $1(z_{ik} > 0)$, which is a binary detector.

3.17 Support vector machine

Consider the perceptron $y_i = \text{sign}(X_i^\top \beta)$, which separates the positive examples and negative examples by projecting the data on vector β, or by a hyperplane that is perpendicular to β. If the positive examples and negative examples are separable, there are be many separating hyperplanes. We want to choose the one with the maximum margin in order to guard against the random fluctuations in the unseen testing examples.

The idea of support vector machine (SVM) is to find the β, so that

(1) for positive examples $y_i = +$, $X_i^\top \beta \geq 1$,

(2) for negative examples $y_i = -$, $X_i^\top \beta \leq -1$.

Here we use +1 and -1, because we can always scale β.

The decision boundary is decided by the training examples that lies on the margin. Those are the support vectors. Let u be an unit vector that has the same direction as β. $u = \frac{\beta}{|\beta|}$.

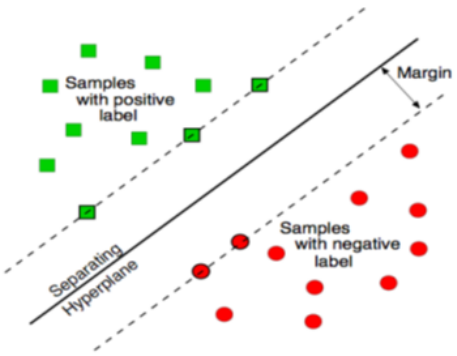

Figure 8: max margin, source web.

Suppose X_i is an example on the margin (i.e., support vector), the projection of X_i on u is

$$\langle X_i, u\rangle = \langle X_i, \frac{\beta}{|\beta|}\rangle = \frac{X_i^\top \beta}{|\beta|} = \frac{\pm 1}{|\beta|}.$$

So the margin is $1/|\beta|$. In order to maximize the margin, we should minimize $|\beta|$ or $|\beta|^2$. Hence, the SVM can be formulated as an optimization problem as follows:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}|\beta|^2, \\ \text{subject to} \quad & y_i X_i^\top \beta \geq 1, \forall i. \end{aligned}$$

Recall $X_i^\top \beta$ is the score, and $y_i X_i^\top \beta$ is the individual margin of observation i. This is the primal form of SVM.

3.18 Hinge loss

When the data are not separable, we may allow limited slacks $\xi_i \geq 0$,

$$\text{minimize} \quad \frac{1}{2}|\beta|^2 + C\sum_{i=1}^n \xi_i, \tag{1}$$

$$\text{subject to} \quad y_i X_i^\top \beta \geq 1 - \xi_i, \forall i. \tag{2}$$

This is equivalent to

$$\text{minimize} \quad \frac{1}{2}|\beta|^2 + C\sum_{i=1}^n \max(0, 1 - y_i X_i^\top \beta),$$

where $\max(0, 1 - y_i X_i^\top \beta)$ is usually called the hinge loss. Recall the loss for perceptron is $\max(0, -y_i X_i^\top \beta)$, which penalizes mistakes or negative margins $y_i X_i^\top \beta$. In comparison, the hinge loss does not only penalize the negative margins $y_i X_i^\top \beta$, it also penalizes margins less than 1.

3.19 SVM and ridge logistic regression

Given the regularized loss function

$$loss(\beta) = \sum_{i=1}^n \max(0, 1 - y_i X_i^\top \beta) + \frac{\lambda}{2}|\beta|^2,$$

we can solve β by gradient descent. The gradient is

$$loss'(\beta) = -\sum_{i=1}^n 1(y_i X_i^\top \beta < 1) y_i X_i + \lambda\beta,$$

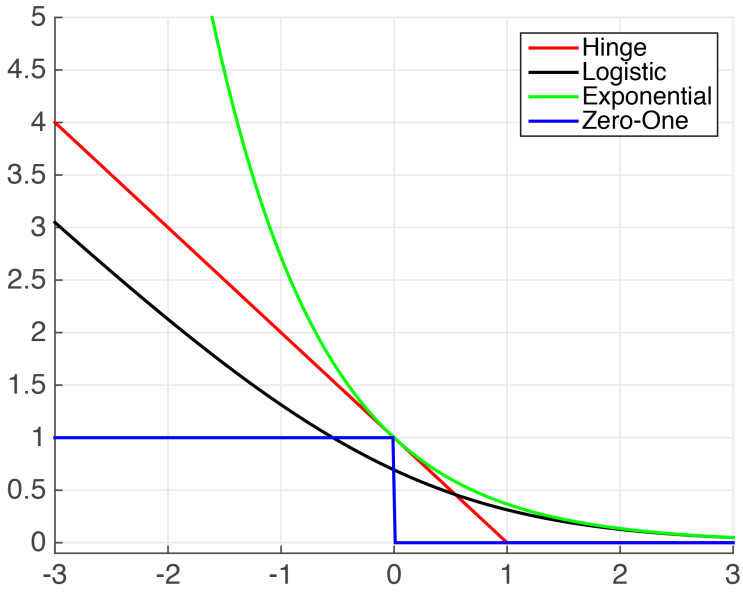

Figure 9: Loss functions, source web.

where $1(\cdot)$ is the indicator function.

This is similar to the ridge logistic regression

$$loss(\beta) = \sum_{i=1}^{n} \log[1 + \exp(-y_i X_i^\top \beta)] + \frac{\lambda}{2}|\beta|^2,$$

whose gradient is

$$loss'(\beta) = -\sum_{i=1}^{n} \sigma(-y_i X_i^\top \beta) y_i X_i + \lambda\beta.$$

R code:

```
my_SVM <- function(X_train, Y_train, X_test, Y_test, lambda = 0.01,
                   num_iterations = 1000, learning_rate = 0.1)
{
  n         <- dim(X_train)[1]
  p         <- dim(X_train)[2] + 1
  X_train1  <- cbind(rep(1, n), X_train)
  Y_train   <- 2 * Y_train - 1
  beta      <- matrix(rep(0, p), nrow = p)

  ntest     <- nrow(X_test)
  X_test1   <- cbind(rep(1, ntest), X_test)
  Y_test    <- 2 * Y_test - 1

  acc_train <- rep(0, num_iterations)
  acc_test  <- rep(0, num_iterations)

  for(it in 1:num_iterations)
  {
    s         <- X_train1 %*% beta
    db        <- s * Y_train < 1
```

```
    dbeta     <-
    beta      <- beta + learning_rate * t(dbeta)
    beta[2:p] <- beta[2:p] - lambda * beta[2:p]

    acc_train[it] <- mean(sign(s * Y_train))
    acc_test[it]  <- mean(sign(X_test1 %*% beta * Y_test))
    }
  model <- list(beta = beta, acc_train = acc_train, acc_test = acc_test)
  model
}
```

3.20 Dual form

The primal form of SVM is max margin, and the dual form of SVM is min distance.

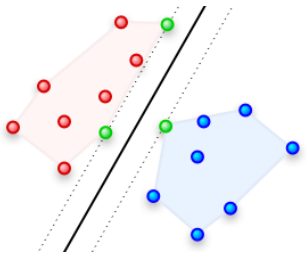

Figure 10: max margin = min distance, source web.

Consider the convex hulls of the positive and negative examples. The margin between the two sets is defined by the minimum distance between two. Let $X_+ = \sum_{i\in+} c_i X_i$ and $X_- = \sum_{i\in-} c_i X_i$ ($c_i \geq 0, \sum_{i\in+} c_i = 1, \sum_{i\in-} c_i = 1$) be two points in the positive and negative convex hulls. The margin is $\min |X_+ - X_-|^2$.

$$\begin{aligned} |X_+ - X_-|^2 &= \left|\sum_{i\in+} c_i X_i - \sum_{i\in-} c_i X_i\right|^2 \\ &= \left|\sum_i y_i c_i X_i\right|^2 \\ &= \sum_{i,j} c_i, c_j y_i y_j \langle X_i, X_j\rangle, \\ \text{subject to} \quad & c_i \geq 0, \sum_{i\in+} c_i = 1, \sum_{i\in-} c_i = 1. \end{aligned}$$

After we solve for c, the non-zeros c_i's are support vectors, i.e., examples on the boundary. c can be solved by sequential minimal optimization, where each step we update two c_i. The dual form enables us to generalize the inner product $\langle X_i, X_j\rangle$ to $K(X_i, X_j)$ where K is a kernel that measures the similarity between X_i and X_j. $K(X_i, X_j) = \langle Z_i, Z_j\rangle$ for some infinite dimensional Z. Hence kernel SVM can be compared to two-layer neural net.

3.21 Adaboost

Adaboost is a committee machine, which consists of a number of base classifiers $h_k(X_i) \in \{+, -\}$, $k = 1, ..., d$. The final classification is a perceptron based on the base classifiers,

$$y_i = \text{sign}\left(\sum_{k=1}^{d} \beta_k h_k(X_i)\right),$$

where β_k can be interpreted as the weight of vote of classifier k. Even if the base classifiers $\{h_k\}$ may be weak, it is still possible to boost them into a strong classifiers.

You may compare the above committee machine to the two-layer neural net we studied before, where $h_k(X_i)$ plays the role of z_{ik}.

3.22 Exponential loss

In AdaBoost, the loss function is in the exponential form:

$$\begin{aligned} \textit{Exponential loss}(\beta) &= \sum_{i=1}^{n} \exp(-y_i \sum_{k=1}^{d} \beta_k h_k(x_i)), \\ \textit{Logistic loss}(\beta) &= \sum_{i=1}^{n} \log(1+\exp(-y_i X_i^\top \beta)), \\ \textit{Hinge loss}(\beta) &= \sum_{i=1}^{n} \max(0, 1-y_i X_i^\top \beta). \end{aligned}$$

Both the hinge loss and the exponential loss are approximations to the logistic loss.

3.23 Coordinate descent

When training adaBoost classifier, we sequentially add members to the committee. It is similar to the matching pursuit version of the coordinate descent for linear regression.

Suppose the current committee has m classifiers, and we want to add a new member h_{new}.

$$\begin{aligned} \text{current committee:} \quad & \sum_{i=1}^{m} \beta_k h_k(X_i), \\ \text{add a new member:} \quad & \sum_{i=1}^{m} \beta_k h_k(X_i) + \beta_{\text{new}} h_{\text{new}}(X_i). \end{aligned}$$

After adding a member, the loss function becomes:

$$loss(\beta_{\text{new}}, h_{\text{new}}) = \sum_{i=1}^{n} e^{-y_i\left(\sum_{k=1}^{m} \beta_k h_k(X_i) + \beta_{\text{new}} h_{\text{new}}(X_i)\right)},$$

where we assume the current members and their weights of votes are fixed. Take derivative

$$\frac{\partial loss}{\partial \beta_{\text{new}}} = \sum_{i=1}^{n} e^{-y_i\left(\sum_{k=1}^{m} \beta_k h_k(X_i) + \beta_{\text{new}} h_{\text{new}}(X_i)\right)} \cdot \left(-y_i h_{\text{new}}(X_i)\right).$$

3.24 Choose a new member

For the current committee without adding a new member, $\beta_{\text{new}} = 0$, the above gradient can be written as

$$\frac{\partial loss}{\partial \beta_{\text{new}}}\Big|_{\beta_{\text{new}}=0} = \sum_{i=1}^{n} e^{-y_i\left(\sum_{k=1}^{m} \beta_k h_k(x_i)\right)} \cdot \left(-y_i h_{\text{new}}(x_i)\right) = -\sum_{i=1}^{n} w_i y_i h_{\text{new}}(x_i),$$

where

$$w_i = e^{-y_i\left(\sum_{k=1}^{m} \beta_k h_k(x_i)\right)}.$$

We can normalize $w_i \leftarrow w_i / \sum_{i=1}^{n} w_i$ to make it a distribution. This distribution focuses on those examples that are not well classified by the current committee.

We want to choose the weak classifier h_{new} by maximizing $\sum_{i=1}^{n} w_i y_i h_{\text{new}}(x_i)$ for the steepest drop in loss. This means we want to train a base classifier on the current distribution of data (X_i, y_i, w_i).

3.25 Determine the voting weight

To find β_{new}, we set the derivative to 0,

$$\begin{aligned}
&\frac{\partial \ell}{\partial \beta_{\text{new}}} = 0, \\
&\sum_{i=1}^{n} w_i e^{-y_i \beta_{\text{new}} h_{\text{new}}(x_i)} \cdot y_i h_{\text{new}}(x_i) = 0, \\
&\sum_{i \in \text{correct}} w_i e^{-\beta_{\text{new}}} = \sum_{i \in \text{wrong}} w_i e^{\beta_{\text{new}}}, \\
&\sum_{i \in \text{correct}} w_i = \sum_{i \in \text{wrong}} w_i e^{2\beta_{\text{new}}}.
\end{aligned}$$

If we define error rate as

$$\epsilon = \frac{\sum_{i \in \text{wrong}} w_i}{\sum_i w_i},$$

β_{new} can be obtained as

$$\beta_{\text{new}} = \frac{1}{2} \log \frac{1-\epsilon}{\epsilon}.$$

It says that the weight is determined by how much error h_{new} made on the weighted data. This explains the name of adaBoost, where "ada" means adapative.

R code:

```
myAdaboost <- function(X_train, Y_train, X_test, Y_test,
                       num_iterations = 200)
{
  n <- dim(X_train)[1]
  p <- dim(X_train)[2]
  threshold <- 0.8

  X_train1 <- 2 * (X_train > threshold) - 1
  Y_train  <- 2 * Y_train - 1

  X_test1 <- 2 * (X_test > threshold) - 1
  Y_test  <- 2 * Y_test - 1

  beta <- matrix(rep(0,p), nrow = p)
  w <- matrix(rep(1/n, n), nrow = n)

  weak_results <- Y_train * X_train1 > 0

  acc_train <- rep(0, num_iterations)
  acc_test  <- rep(0, num_iterations)
  for(it in 1:num_iterations)
  {
    w <- w / sum(w)
    weighted_weak_results <- w[,1] * weak_results
    weighted_accuracy <- colSums(weighted_weak_results)

    e <- 1 - weighted_accuracy
    j <- which.min(e)

    dbeta <-
    beta[j] <- beta[j] + dbeta
    w <- w *
```

```
    acc_train[it] <- mean(sign(X_train1 %*% beta) == Y_train)
    acc_test[it] <- mean(sign(X_test1 %*% beta) == Y_test)
  }
  output <- list(beta = beta, acc_train = acc_train, acc_test = acc_test)
  output
}
```

3.26 Earlier non-adaptive version

An earlier version of boosting is as follows:

(1) Read in a batch of, say, 100 examples, learn a h_1;

(2) Read in another batch of 100 examples, classify them using h_1, keep the examples where h_1 makes mistakes, then learn a new classifier h_2 on these examples;

(3) Read in a third batch of 100 examples, classify them using h_1, h_2, keep the examples where h_1 and h_2 do not agree or both make mistakes, then learn h_3 on these examples.

The committee performs a majority vote by $h_1 + h_2 + h_3$. One can extend the method to include more members. The majority vote corresponds to $\beta_k = 1$. In steps (2) and (3), keeping the examples where the current committee makes mistakes corresponds to computing w_i.

3.27 Adaboost, epsilon-boosting, and neural net

Note that it is possible that the same classifier is selected multiple times, so that β_{new} can be the increment of the coefficient of a classifier that has already been in the committee. In fact, we can always consider β_{new} as an increment if we assume all the classifiers are included from the very beginning except that they all start from zero β.

If we only allow the increment β_{new} to be very small, the learning method becomes epsilon-boosting, which corresponds to gradient descent with ℓ_1 regularization.

Then adaboost corresponds to the two-layer neural net, where $h_k(X_i)$ corresponds to z_{ik}. The computation of w_k corresponds to computing the error $y_i - p_i$. The selection of a new member corresponds to updating α_k. The computation of β_{new} corresponds to updating β in the neural net.

4 Bayesian regression by MCMC

We have studied the Bayesian formulation of ridge regression, where the ℓ_2 regularization corresponds to a prior $p(\beta) \sim \mathrm{N}(0, \tau^2 \mathbf{I})$. In general, the Bayesian method treats the unknown parameter β as a random variable that follows a prior distribution. After observing data Y, the posterior distribution is $p(\beta|Y) = p(\beta, Y)/p(Y) \propto p(\beta)p(Y|\beta)$, which is a combination of prior and likelihood.

We can draw samples from $p(\beta|Y)$ by iterative algorithms called Markov chain Monte Carlo.

4.1 Bayesian variable selection

Consider the regression $\mathbf{Y} = \mathbf{X}\beta + \epsilon$, where $\mathbf{X}$ is $n \times 1$, i.e., $p = 1$. Consider the hypothesis testing $H_0 : \beta = 0$ vs $H_1 : \beta \neq 0$. The least squares estimate $\hat{\beta} = \langle \mathbf{Y}, \mathbf{X} \rangle / |\mathbf{X}|^2 \sim \mathrm{N}(\beta, \sigma^2/|\mathbf{X}|^2)$. The frequentist test is that we reject H_0 if $|\hat{\beta}| > C$ for a cut-off threshold C.

For a Bayesian test, we can put prior distribution $p(H_1) = \rho$ and $p(H_0) = 1 - \rho$. Given H_0, $\beta = 0$. Given H_1, we can assume a prior distribution $p(\beta) \sim \mathrm{N}(0, \tau^2)$. This is often called the spike and slab prior. Then under H_0,

$\hat{\beta} \sim \mathrm{N}(0, \sigma^2/|\mathbf{X}|^2)$, and under H_1, $\hat{\beta} \sim \mathrm{N}(0, \tau^2 + \sigma^2/|\mathbf{X}|^2)$. The posterior ratio is

$$\begin{aligned}\frac{p(H_1|\mathbf{Y})}{p(H_0|\mathbf{Y})} &= \frac{p(H_1)}{p(H_0)} \frac{p(\mathbf{Y}|H_1)}{p(\mathbf{Y}|H_0)} = \frac{p(H_1)}{p(H_0)} \frac{p(\hat{\beta}|H_1)}{p(\hat{\beta}|H_0)} \\ &= \frac{\rho}{1-\rho} \frac{\sqrt{\sigma^2/|\mathbf{X}|^2}}{\sqrt{\tau^2 + \sigma^2/|\mathbf{X}|^2}} \exp\left[\frac{\hat{\beta}^2}{2\sigma^2/|\mathbf{X}|^2} - \frac{\hat{\beta}^2}{2(\tau^2 + \sigma^2/|\mathbf{X}|^2)}\right] = a\end{aligned}$$

In the above derivation, $p(\mathbf{Y}|\hat{\beta}, H_1) = p(\mathbf{Y}|\hat{\beta}, H_0)$, i.e., $\hat{\beta}$ is sufficient for determining H_1 vs H_0.

Then $p(H_1|\mathbf{Y}) = p(H_1|\mathbf{Y})/[p(H_1|\mathbf{Y}) + p(H_0|\mathbf{Y})] = a/(a+1)$. So we can sample H_1 or H_0 by flipping a bias coin with probability $a/(a+1)$. If we get tail, then H_0 is sampled, and $\beta = 0$. If we get head, then H_1 is sampled, and we need to sample β from the posterior distribution, which is obtained by ridge regression, specifically

$$[\beta|H_1, \mathbf{Y}] \sim \mathrm{N}\left(\frac{\hat{\beta}/(\sigma^2 + |\mathbf{X}|^2)}{1/(\sigma^2 + |X|^2) + 1/\tau^2}, \frac{1}{1/(\sigma^2 + |\mathbf{X}|^2) + 1/\tau^2}\right),$$

where the mean is a balance between least squares estimate $\hat{\beta}$ and the prior mean 0, weighted by the precisions (inverses of the variances), and the overall precision is the sum of the precisions of the least squares and the prior.

4.2 From coordinate descent to Gibbs sampler

For $p > 1$, we can assume each component $\beta_j \sim (1-\rho)\delta_0 + \rho\mathrm{N}(0, \tau^2)$, i.e., the spike and slab prior, independently. It is difficult to sample $p(\beta|\mathbf{Y})$ directly. But we can iteratively sample $p(\beta_j|\beta_{-j}, \mathbf{Y})$, where β_{-j} denotes the current value of all the components except j. Let $\mathbf{R}_j = \mathbf{Y} - \sum_{k \neq j} \mathbf{X}_k \beta_k$. We can sample β_j according to the previous subsection, by assuming $(\mathbf{R}_j, \mathbf{X}_j)$ be $(\mathbf{Y}, \mathbf{X})$ in the previous section, by first sampling H_1 vs H_0, and then sample β_j. Such an algorithm is called the Gibbs sampler. It can be considered a stochastic version of the coordinate descent for Lasso path. We may run the algorithm for 10,000 iterations. We throw away 5,000 iterations, and use the rest 5,000 iterations for inference.

We can think of the Gibbs sampler in terms of population migration. Imagine 1 million people start from the same location on an island, and each person runs an independent Gibbs sampler, which iterates random relocation horizontally and vertically. Then eventually the distribution of the population will reach uniform distribution, which is the stationary distribution in this case.

R code:

```
n = 1000
p = 100
s = 10
T = 10000
tau = 3
sigma = 1
rho = s/p;
X = matrix(rnorm(n*p), nrow=n)
beta_true = matrix(rep(0, p), nrow = p)
beta_true[1:s] = rnorm(s)*tau
Y = X %*% beta_true + rnorm(n)*sigma

beta = matrix(rep(0, p), nrow = p)
beta_all = matrix(rep(0, p*T), nrow = p)
R = Y
ss = rep(0, p)
for (j in 1:p)
    ss[j] = sum(X[, j]^2)

for (t in 1:T)
{
    for (j in 1:p)
```

```
    {
        db = sum(R*X[, j])/ss[j]
        b = beta[j]+db
        v0 = sigma^2/ss[j]
        v1 = tau^2 + v0
        a = rho/(1-rho)
        a = a*sqrt(v0)/sqrt(v1)
        a = a*exp(min(b^2/(2*v0)-b^2/(2*v1),100))
        pr = a/(a+1)
        if (runif(1)<pr)
        {
            mu =
            sig =
            bj = rnorm(1)*sqrt(sig)+mu
        }
        else
        {
            bj = 0
        }
        db = bj - beta[j]
        beta[j] = bj
        beta_all[j, t] = beta[j]
        R = R - X[, j]*db
    }
}
print(beta)
```

4.3 Bayesian logistic regression

We may also perform Bayesian logistic regression by assuming a normal prior $p(\beta) \sim \mathrm{N}(0, \tau^2 \mathbf{I})$ as a Bayesian version of ridge logistic regression (which is closely related to SVM). The log-posterior is $\log p(\beta|\mathbf{Y}) = \log p(\beta) + \log p(\mathbf{Y}|\beta)$, where $p(\mathbf{Y}|\beta)$ is the log-likelihood, and $\log p(\mathbf{Y}|\beta) = \sum_{i=1}^n [y_i X_i^\top \beta - \log(1 + \exp(X_i^\top \beta))]$.

4.4 From gradient descent to Langevin dynamics

We can use Langevin dynamics to sample from $p(\beta|\mathbf{Y})$, which iterates

$$\beta_{t+1} = \beta_t + \frac{\gamma}{2}\left[\sum_{i=1}^n (y_i - p_i) X_i - \lambda \beta_t\right] + \epsilon_t,$$

where $\epsilon_t \sim \mathrm{N}(0, \gamma)$. Without ϵ_t, the above algorithm is gradient ascent for ridge logistic regression.

4.5 Metropolis-Hastings correction

If the step size γ is big, we may need to correct the above dynamics by Metropolis-Hastings method. For sampling $\pi(x)$, let $B(x, y) = P(X_{t+1} = y | X_t = x)$ be the base chain. This chain may not converge to π or have π as the stationary distribution. From the population migration perspective, $\pi(x)B(x, y)$ is the number of people who propose to go from x to y, while $\pi(y)B(y, x)$ is the number of people who propose to go from y to x. In order to maintain the stationary distribution π, we want to make the two numbers to be the same. So we want to accept the proposal from x to y by $\min(1, [\pi(y)B(y, x)]/[\pi(x)B(x, y)])$. With such an acceptance probability, the numbers from x to y and from y to x are both $\min(\pi(x)B(x, y), \pi(y)B(y, x))$, so that the stationary distribution is preserved.

For Langevin dynamics, we can show that as $\gamma \to 0$, then the acceptance probability goes to 1.

5 Conclusion: a tightly knitted story

We can knit our story by several threads. The main thread is our data structure and notation $(\mathbf{Z}, \mathbf{X}, \mathbf{Y}, \beta)$, where $\mathbf{X}$ are the observed variables, $\mathbf{Y}$ are the outcomes or labels, $\mathbf{Z}$ are the latent variables or features, β is the parameters to be learned.

For continuous $\mathbf{Y}$, we have least squares regression, which can be computed by the sweep operator or QR decomposition. For binary $\mathbf{Y}$, we have maximum likelihood logistic regression, which is iterated reweighed least squares.

Another thread is regularization. We have ℓ_2 regularization for ridge regression, ridge logistic regression, and SVM, which can be solved directly, or by gradient based methods. We have ℓ_1 regularization for Lasso, which can be solved by coordinate descent or epsilon-boosting, which is closely related to adaboost.

The regularization corresponds to Bayesian prior. To sample the posterior, we can extend coordinate descent for Lasso to Gibbs sampler for Bayesian variable selection. We can extend gradient descent for logistic regression to Langevin dynamics for Bayesian logistic regression.

A third thread is loss function in classification, we have logistic loss, perceptron loss, hinge loss, and exponential loss. The SVM is the max-margin perceptron, and can be compared to ridge logistic regression.

The fourth thread is about the hidden $\mathbf{Z}$. In the unsupervised setting where we do not observe $\mathbf{Y}$, we have factor analysis and principal component analysis, as well as matrix factorization and completion. In the supervised setting where we observe $\mathbf{Y}$, we have neural nets. Assuming rectified linear unit, the neural net can be considered a high-dimensional spline model. The adaboost can be compared to a two-layer neural net. Kernel SVM corresponds to an infinite dimensional $\mathbf{Z}$.