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

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

and the order of operations in $\text{SWP}[1:m]$ 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, $\text{SWP}[1:n]A = -A^{-1}$.

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.

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

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

(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 $\beta = (\beta_j, j = 1, ..., p)^\top$.

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

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

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 construct a matrix $\mathbf{Z} = [\mathbf{XY}]$, 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

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

where $\text{RSS} = \|\mathbf{Y} - \mathbf{X}\hat{\beta}\|_{\ell_2}^2$ 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 $\tilde{A} = \text{GJ}[k]A$, 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., GJ$[1 : m]$ means we apply Gauss-Jordan for $k = 1 : m$. Being a row operation, Gauss-Jordan is linear, i.e., $\tilde{A} = \text{GJ}[k]A$ amounts to $\tilde{A} = G_k A$ for a matrix G_k. Thus

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

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

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

$$\text{GJ}[1 : m] \left[\begin{array}{cc|cc} A_{11} & A_{12} & I_1 & 0 \\ A_{21} & A_{22} & 0 & I_2 \end{array}\right] = \left[\begin{array}{cc|cc} I_1 & A_{11}^{-1}A_{12} & A_{11}^{-1} & 0 \\ 0 & A_{22} - A_{21}A_{11}^{-1}A_{12} & -A_{21}A_{11}^{-1} & I_2 \end{array}\right].$$

We can write GJ$[1 : m] = \text{GJ}[A_{11}]$, which is a matrix version of Gauss-Jordan. If we compare Gauss-Jordan GJ$[1 : m]$ with the sweep operator 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 GJ$[1 : m] = \text{GJ}[A_{11}]$, we have SWP$[1 : m] = \text{SWP}[A_{11}]$. 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

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

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 $\Delta\beta_j \to \langle \mathbf{R}, \mathbf{X}_j \rangle / \|\mathbf{X}_j\|^2$. Then update $\beta_j \to \beta_j + \Delta\beta_j$, and $\mathbf{R} \to \mathbf{R} - \mathbf{X}_j \Delta\beta_j$.

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.

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

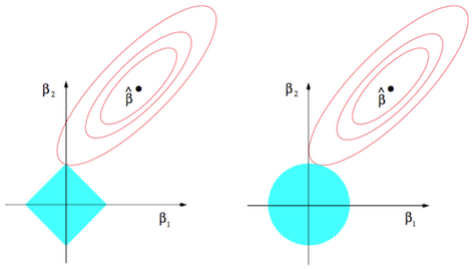

Figure 1: Lasso. Source: web.

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 $\|\beta\|_{\ell_1} \leq t$ 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 $\|\beta\|_{\ell_1} \leq t$. 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 $(\mathbf{X}_j, j = 1, ..., p)$.

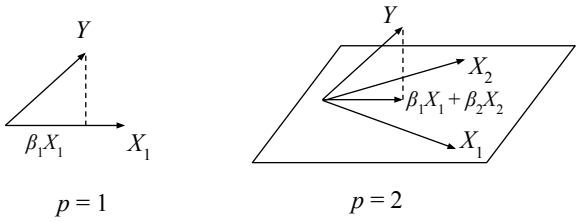

Figure 2: Least squares projection

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 $\sum_{j=p+1}^n \text{E}(\delta_j^2) = (n-p)\sigma^2$.

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

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

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

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

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.

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), we have Q and $Q^\top$ are inverse of each other.

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.

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

Note that since the orthogonal transformation preserves the length of vectors, we know

$$|x_1^*| = |x_1| = \sqrt{x_{11}^2 + x_{12}^2 + \ldots + x_{1n}^2},$$

which means the value of x_{11} is determined

$$x_{11}^* = \pm |x_1| .$$

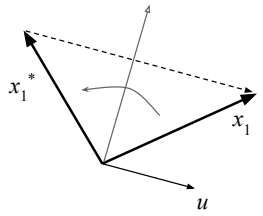

Figure 3: Household Reflection

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

To find a transformation H which can rotate vector x_1 to x_1^*, one simple way is to construct an isosceles triangle where x_1^* is a reflection of x_1:

$$x_1^* = x_1 - 2\langle x_1, u\rangle u = x_1 - 2uu^\top x_1 = H_1 x_1,$$

where

$$u = \frac{x_1 - x_1^*}{|x_1 - x_1^*|}, \quad H_1 = I - 2uu^\top.$$

To transform the original matrix $\mathbf{X}_{n\times p}$ into a upper triganle matrix, we can repeat this procedure p times, the orthogonal matrix $H = H_p \dots H_2 H_1$. Hence, $H\mathbf{X} = R$. let $Q = H^T$, we obtain the QR decomposition of $\mathbf{X}$

$$\mathbf{X} = QR.$$

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
```

Linear regression using QR decompostion. We rotate the matrix $(\mathbf{XY})$ by some orthogonal matrix Q,

$$\begin{bmatrix}\mathbf{X} & \mathbf{Y}\end{bmatrix} \xrightarrow{Q^\top} \begin{bmatrix}R & \mathbf{Y}^*\end{bmatrix} = \begin{bmatrix}R_1 & \mathbf{Y}_1^* \\ 0 & Y_2^*\end{bmatrix},$$

where R_1 is a upper triangular matrix.

To solve the least square,

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

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

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
```

2.10 Multivariate statistics

Expectation of a random matrix. Consider a random matrix X. Suppose X is $m \times n$, and the elements of X are x_{ij}, $i = 1, ..., m$ and $j = 1, ..., n$. Usually we write $X = (x_{ij})_{m\times n}$ or simply $X = (x_{ij})$. We define

$$\text{E}(X) = (\text{E}(x_{ij})),$$

i.e., taking expectations element-wise. Let A be a constant matrix of appropriate dimension, then $\text{E}(AX) = A\text{E}(X)$. Let B be another constant matrix of appropriate dimension, then $\text{E}(XB) = \text{E}(X)B$. The proof follows the linear property of expectation. Let $Y = AX$, then $y_{ij} = \sum_k a_{ik}x_{kj}$, and $\text{E}(y_{ij}) = \text{E}(\sum_k a_{ik}x_{kj}) = \sum_k a_{ik}\text{E}(x_{kj})$. Thus $\text{E}(Y) = (\text{E}(y_{ij})) = (\sum_k a_{ik}\text{E}(x_{kj})) = A\text{E}(X)$.

Variance-covariance matrix of a random vector. Let X be a random vector. Let $\mu_X = \text{E}(X)$. We define

$$\text{Var}(X) = \text{E}[(X - \mu_X)(X - \mu_X)^\top].$$

Then the (i, j)-th element of $\text{Var}(X)$ is $\text{Cov}(x_i, x_j)$. Let A be a constant matrix of appropriate dimension, then $\text{Var}(AX) = A\text{Var}(X)A^\top$. This is because

$$\begin{aligned}
\text{Var}(AX) &= \text{E}[(AX - \text{E}(AX))(AX - \text{E}(AX))^\top] \\
&= \text{E}[(AX - A\mu_X)(AX - A\mu_X)^\top] \\
&= \text{E}[A(X - \mu_X)(X - \mu_X)^\top A^\top] \\
&= A\text{E}[(X - \mu_X)(X - \mu_X)^\top]A^\top \\
&= A\text{Var}(X)A^\top.
\end{aligned}$$

We can also define $\text{Cov}(X, Y) = \text{E}[(X - \mu_X)(Y - \mu_Y)^\top]$, then

$$\begin{aligned}
\text{Cov}(AX, BY) &= \text{E}[(AX - A\mu_X)(BY - B\mu_Y)^\top] = \text{E}[A(X - \mu_X)(Y - \mu_Y)^\top B^\top] \\
&= A\text{E}[(X - \mu_X)(Y - \mu_Y)^\top]B^\top = A\text{Cov}(X, Y)B^\top
\end{aligned}$$

2.11 Multivariate normal

Let $X \sim \mathrm{N}(\mu, \Sigma)$. The density is

$$f(x) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

For simplicity, we shall assume that $\mu = 0$ in the following. Otherwise we can simply let $X' = X - \mu$ to centralize X.

Conditional. Let us partition X into X_1, X_2.

$$\begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \sim N\left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}\right). \tag{1}$$

The conditional distribution

$$[X_2|X_1] \sim \mathrm{N}(\Sigma_{21}\Sigma_{11}^{-1}X_1, \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}).$$

The computation can be accomplished by the sweep operator. The above is a population version of regression.

2.12 Principal component analysis

Eigen decomposition. Σ can be decomposed into $\Sigma = Q\Lambda Q^T$, where $Q = (q_1, q_2 \dots q_n)$ are orthonormal vectors, i.e., $\langle q_i, q_j\rangle = \delta_{ij}$, where $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ otherwise. $\Lambda = \mathrm{diag}(\lambda_1, ..., \lambda_n)$ is a diagonal matrix. Q forms a set of basis. We can expand X in Q, so that $X = \sum_i z_i q_i = QZ$, and z_i are the coordinates of X in the basis Q, i.e., $z_i = \langle X, q_i\rangle$, so $Z = Q^T X$. So $Q^TQ = QQ^T = I$. $|\Sigma| = |\Lambda|$. The density of Z is

$$\begin{aligned} f(z) &= \frac{1}{(2\pi)^{n/2}|\Lambda|^{1/2}} \exp\left(-\frac{1}{2}(QZ)^T\Sigma^{-1}QZ\right) \\ &= \frac{1}{(2\pi)^{n/2}|\Lambda|^{1/2}} \exp\left(-\frac{1}{2}Z^T\lambda^{-1}Z\right) \\ &= \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\lambda_i}} \exp\left(-\frac{z_i^2}{2\lambda_i}\right), \end{aligned}$$

where the Jacobian of $X = QZ$ is 1 since the transformation is just a rotation. Thus $z_i \sim \mathrm{N}(0, \lambda_i)$ independently. So $\mathrm{E}(Z) = 0$ and $\mathrm{Var}(Z) = \Lambda$. As a result $\mathrm{E}(X) = 0$ and $\mathrm{Var}(X) = Q\Lambda Q^T = \Sigma$.

The column vectors in Q and the diagonal elements in Λ are eigenvectors and eigenvalues of Σ.

If the top k basis in Q are selected according to λ_i from the biggest to the smallest, we captured the top k pricinple directions of data along which the variability are most significant. The choice of k in practice can be chosen by keeping the reconstruction error of data in an acceptable range.

Vector form of power method. For a vector v, let u be the transformed coordinate in system Q, i.e. $v = Qu$.

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

This says the vector Σv becomes $(\lambda_1 u_1, \lambda_2 u_2, \cdots, \lambda_p u_p)^\top$ in system Q.

If we repeat this process n times,

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

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

- Repeat the following steps:

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

  Update $v = \Sigma\tilde{v}$.

To get q_2 using this method, we choose a vector $v \perp q_1$ that is perpendicular to q_1. To get q_3, choose v to be perpendicular to both q_1 and q_2, i.e. $v \perp q_1$ and $v \perp q_2$. So on and so forth, we can get all the vectors in Q.

Matrix form of power method. Suppose initially we have a random matrix $V = (v_1, v_2, \cdots, v_p)$, whose orthogonalized version is $\tilde{V} = (\tilde{v}_1, \tilde{v}_2, \cdots, \tilde{v}_p)$ where each of these column vectors is a unit vector and every two of them are orthogonal to each other.

Power method is a iterative algorithm to compute eigen decomposition.

- Repeat the following steps:

  Compute $\tilde{V}$, the orthogonalized V.

  Update $V = \Sigma\tilde{V}$.

We use QR decomposition to perform orthogonalization.

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

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)
```

3 Classification based on generalized linear regression

3.1 Logistic regression

Consider a dataset with n samples, where $X_i^\top = (x_{i1}, \cdots, x_{ip})$ is the features vector, $y_i \in \{0, 1\}$ is the class label for $i = 1, 2, \ldots, n$.

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 $y_i \sim \text{Bernoulli}(p_i)$, $\Pr(y_i = 1) = p_i$. And we assume

$$\log \frac{p_i}{1 - p_i} = X_i^\top \beta.$$

Let $\eta_i = X_i^\top \beta$ be the score, then the probability p_i is simply a mapping from $\mathbb{R}$ to $(0, 1)$

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

where the function $f(\eta_i) = \dfrac{1}{1 + e^{-\eta_i}}$ is the sigmoid function.

Estimating β. 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

$$l(\beta) = \log L(\beta) = \sum_{i=1}^{n} y_i X_i^\top \beta - \log(1 + e^{X_i^\top \beta}).$$

Gradient ascent. There is no close form solution for β. To find the maximum, let's first take derivative

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

And we use gradient ascent to iteratively update β along the gradient direction.

$$\beta_{t+1} = \beta_t + \gamma \sum_{i=1}^{n} (y_i - p_i) x_i,$$

where γ is usually called learning rate. Intuitively, the idea is to accumulatively memorizes the training examples on which the learner is not doing well, i.e. $y_i \neq p_i$. It is very much like getting your homework graded and learning from the mistakes.

Newton-Raphson Method. A more efficient ways is to update β using Newton-Raphson method.

Solving $h(x) = 0$. As the first case, to solve the equation $h(x) = 0$, we take the first order Taylor expansion

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

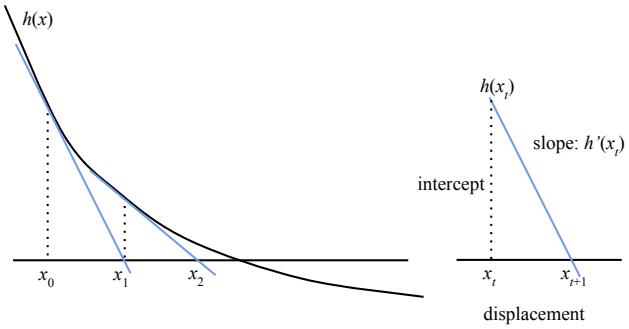

Figure 4: Newton-Raphson

Each iteration, it solves a linear surrogate function to the original one.

Solving $\max f(x)$. In this case, we actually want to solve $f'(x) = 0$. Using the equation above, use Newton-Raphson method, we have

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

It essentially maximize a quadratic approximation of the original function at x_t.

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

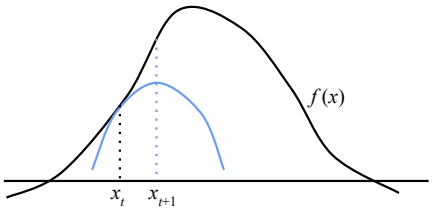

Figure 5: Newton-Raphson Second Order

Vector form. When the variable is a vector $x = (x_1, x_2, \cdots, x_n)^\top$, to solve $h(x) = 0$,

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

If function $h(x) = (h_1(x), h_2(x), \cdots, h_n(x))^T$ is n-dimensional, the derivative is a $n \times n$ matrix:

$$h'(x_t) = \left(\frac{\partial h_i(x)}{\partial x_j} \right)_{i,j}.$$

In the case of maximization, suppose $f(x) = f(x_1, x_2, \ldots, x_n)$,

$$f'(x) = \left(\frac{\partial f}{\partial x_i} \right)_i, \quad f''(x) = \left(\frac{\partial f}{\partial x_i \partial x_j} \right)_{i,j},$$

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

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

Solving 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 then can update β by

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

Relation to Linear Regression. 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 $z_i = X_i^\top \beta_t + \dfrac{y_i - p_i}{w_i}$, $\tilde{X}_i = \dfrac{X_i}{\sqrt{w_i}}$, $\tilde{z}_i = \dfrac{z_i}{\sqrt{w_i}}$, we can further rewrite the equation above as follows.

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

This means we can actually view β_{t+1} as the solution to the weighted least squares problem

$$\texttt{WLS}(z_i, x_i, w_i),$$

or the ordinary least squares with

$$\tilde{z}_i = \tilde{X}_i^{T} \beta_{t+1}.$$

Using Newton-Raphson method to solve logistic regression can be summarized as follows.

- Start with β_t,
- $\eta_i = X_i^\top \beta_t$.
- $p_i = \text{sigmoid}(\eta_i)$,
- $w_i = p_i(1-p_i)$,
- $z_i = \eta_i + \dfrac{y_i - p_i}{w_i}$,
- $\tilde{X}_i = \dfrac{X_i}{\sqrt{w_i}}, \tilde{z}_i = \dfrac{z_i}{\sqrt{w_i}}$,

- $\beta_{t+1} = \texttt{regression}(\tilde{z}_i, \tilde{X}_i)$.

This procedure is also 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
```

3.2 Neural network

In the logistic regression, we transform the data x by a linear transformation $x^\top \beta$ and determine the class label by a non-linear sigmoid function. When x are not good features, however, we can construct new features z from x through some non-linear transformation. And the class label can be inferred based on the transformed features.

- $y_i \sim \text{Ber}(p_i)$
- $p_i = \text{sigmoid}(z_i^T \beta)$

Now the question becomes, how to find the transformed features z_i. This can be problem-dependent and usually requires domain knowledge in real-world applications. Here, for simplicity, we assume z_i is constructed by a logistic regression on x_i.

- $z_{ik} = \text{sigmoid}(X_i^\top \alpha_k)$.

Now, we obtained a very simple two-layer neural network.

Loss Function. Since the top-most layer is still a logistic regression, we can define the loss function as the negative log likelihood. Our objective is to minimize this loss.

$$\ell = -\frac{1}{n} \sum_{i=1}^{n} (y_i \log p_i + (1 - y_i) \log(1 - p_i))$$

Alternatively, we can also use the least squares error.

$$\ell = \sum_{i=1}^{n} (y_i - p_i)^2$$

Feedforward. To get the final output from the network, we need to transform the data input data x through the layers in the network. This is called feedforward. The parameters in each layer is called weights and bias.

Backpropagation. In our network, the parameters we want to update are β and α_k's.

Suppose the top layer loss function is ℓ, we want to calculate the following two derivatives:

- $\dfrac{\partial \ell}{\partial \beta}$,
- $\dfrac{\partial \ell}{\partial \alpha} = \dfrac{\partial \ell}{\partial z} \dfrac{\partial z}{\partial \alpha}$.

Note that when calculate the gradient of parameters in the lower layer, we are actually using the chain rule where the loss is propagated back from upper layers. Hence, this is called backpropagation.

Then we use these gradient to update the parameters.

- $\beta_{t+1} = \beta_t + \dfrac{\partial \ell}{\partial \beta}$,
- $\alpha_{k,t+1} = \alpha_{k,t} + \dfrac{\partial \ell}{\partial \alpha}$.

In the implementation, we want to abstract the operations. Suppose X and y are input and output of one layer, W and b are weights and bias in this layer, respectively, the backpropagation operation of the layer simply computes three values:

- $\frac{dy}{dW}$ and $\frac{dy}{db}$ are the derivative to the parameters in this layer.
- $\frac{dy}{dX}$ is the derivative to the input of this layer, which will be propagated back to lower layers.

Training. In the training phase, we apply gradient descent which iteratively update the parameters in the network. Conceptually, each iteration consists of three operations:

- feed forward to get current network output
- backpropagation to get gradients
- update the paramters along the gradients

Here we also implement a 'accuracy()' function to calculate the classification accuracy of our network.

3.3 Support vector machine

Primal Form. Perceptron is an algorithm which separates two classes by the sign of transformed data, i.e. $y_i = \text{sign}(X_i^\top \beta)$. However, there can be multiple β's which separates the data. The fundamental idea of SVM is to find the separating hyperplan with a large margin.

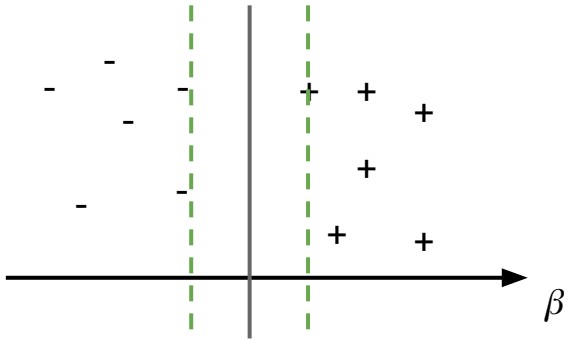

Figure 6: SVM

The goal is to find the β, so that

(1) for positive examples $y = +$, $x^\top \beta \geq 1$,

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

Here we use +1 and -1, becasuse we can always scale β to make the margin 1.

The decision boundary is essentially decided by the training examples that lies on the margin. Let u be an unit vector who has the same direction as β.

$$u = \frac{\beta}{|\beta|}.$$

Suppose x is an example on the margin (a.k.a. support vector), the projection of x on u is

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

Hence, the SVM can be formulated as an optimization problem as follows:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}|\beta|^2, && (2)\\ \text{subject to} \quad & y_i X_i^\top \beta \geq 1, \forall i. && (3) \end{aligned}$$

This is called the primal form.

Hinge loss. When the data is not separatable, SVM can be defined as follows which allows limited cross margin ξ.

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}|\beta|^2 + c\sum_{i=1}^n \xi_i, && (4)\\ \text{subject to} \quad & y_i X_i^\top \beta \geq 1 - \xi_i, \forall i. && (5) \end{aligned}$$

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 $\sum_{i=1}^n \max(0, 1 - y_i X_i^\top \beta)$ is usually called the hinge loss.

From this perspective, SVM can be think of an approximation to the loss of logistic regression hinge loss with a ℓ_2 regularization.

$$\begin{aligned} \ell_{\text{logistic}}(\beta) &= \sum_{i=1}^n \log(1 + e^{-y_i X_i^\top \beta}),\\ \ell_{\text{svm}}(\beta) &= \sum_{i=1}^n \max(0, 1 - y_i X_i^\top \beta) + \frac{\lambda}{2}|\beta|^2. \end{aligned}$$

The following code segment plots the two loss functions.

SVM by gradient descent. Given the loss function $L(\beta) = \sum_{i=1}^n \max(0, 1 - y_i X_i^\top \beta) + \frac{\lambda}{2}|\beta|^2$, we can solve β by gradient descent.

Take derivative, we have

$$\frac{\partial L}{\partial \beta} = \sum_{i=1}^n 1(1 - y_i X_i^\top \beta < 1) \cdot (-y_i x_i) + \lambda\beta,$$

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

The code from logistic regression can be reused with a minor change in the gradient calculation.

Dual form. Historically, people transform the primal form to a dual form. The idea is to view the problem from the geometry perspective.

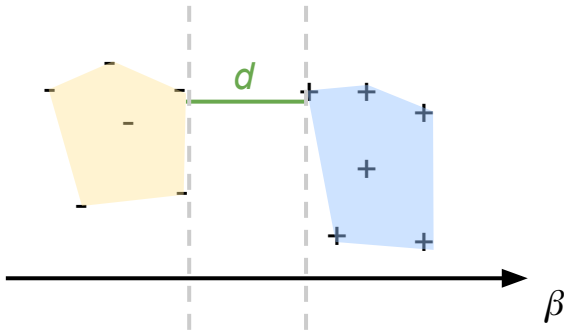

Figure 7: default

Consider the convex hull of the positive and negative population, the margin between the two population is therefore 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 population (in the region of the convex hull). The margin is

$$\min |x_+ - x_-|^2 .$$

It can be rewritten as

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

After we solve for c, non-zeros c_i's are support vectors, i.e. data examples on the boundary.

3.4 Adaboost

Let's first examine AdaBoost from the perspective of loss function. Recall that we have seen loss functions for logistic regression and SVM.

$$\begin{aligned} \ell_{logistic} &= \sum_{i=1}^{n} \log(1 + e^{-y_i X_i^\top \beta}), \\ \ell_{hinge} &= \sum_{i=1}^{n} \max(0, 1 - y_i X_i^\top \beta). \end{aligned}$$

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

$$\ell_{AdaBoost} = \sum_{i=1}^{n} e^{-y_i \sum_{k=1}^{m} \beta_k h_k(x_i)},$$

where $h_k(x_i) \in \{+, -\}$ are weak classifiers (aka. week learners or base learners).

The following code segment plots these loss functions.

Training. Examine the loss function

$$\ell_{AdaBoost} = \sum_{i=1}^{n} e^{-y_i \sum_{k=1}^{m} \beta_k h_k(x_i)}.$$

Intuitively, the set of $h_k()$'s forms a committe, each member has a voting weight β_k. The classification decision is made based on the voting of committe members, sign $(\sum_k \beta_k h_k(x_i))$.

When training a AdaBoost classifier, we usually use a sequential method where we select committee members one after another so that training examples are getting separated gradually.

Suppose a committee with k classifiers, we want to add a new member h_{new}. The votes are

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

After adding a member, the loss function becomes:

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

Take derivative

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

Choose a new member h_{new}. At the point we haven't choose a new member, $\beta_{\text{new}} = 0$, the above gradient can be written as

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

where

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

Here we want to choose the weak classifier h_{new} which gives the maximum drop in loss, i.e. we want to maximize $\sum_{i=1}^{n} w_i y_i h_{\text{new}}(x_i)$.

Intuitively, we can consider the original training examples being weighted by w_i's (voting from current committee). The correctly classified examples receive lower weights, whereas the wrongly classified examples get higher weights. Then the ideal h_{new} is the one which can do well in this weighted examples among all candidates. This new weak classifier therefore comes to resolve the issues that the current committee cannot handle well.

Determine the voting weight β_{new}. To find β_{new}, we set the derivative to 0,

$$\frac{\partial \ell}{\partial \beta_{\text{new}}} = 0,$$

$$\sum_{i=1}^{n} w_i e^{-y_i \beta_{\text{new}} h_{\text{new}}(x_i)} \cdot y_i h_{\text{new}}(x_i) = 0,$$

$$\sum_{i \in \text{correct}} w_i e^{-\beta_{\text{new}}} = \sum_{i \in \text{wrong}} w_i e^{\beta_{\text{new}}},$$

$$\sum_{i \in \text{correct}} w_i = \sum_{i \in \text{wrong}} w_i e^{2\beta_{\text{new}}}.$$

If we define error rate as $\epsilon = \frac{\sum_{i \in \text{wrong}} w_i}{\sum_i w_i}$, β_{new} can be written 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 also explains the name of AdaBoost, where "Ada" means adapative.

Remarks and a bit of history. AdaBoost was originally developed for answering a theoretical question about "whether strong learning = weak learning" (starting from the perspective of game theory instead of loss function). Then it turns out to be very useful in practice. AdaBoost seldom overfits. Most of the time, the loss function tries to increase the margin. One can be saved from overfitting only by training the classifier sequentially. The key lies in the fixing weights for the already learned classifiers. If, otherwise, you minimize the loss function using gradient descent, the resulting classifier will overfit.

The original idea of boosting was non-adpative.

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

(2) Read another 100 examples, classify using h_1, then learn a new classifier h_2 with the mistake;

(3) Read another 100 examples, classify using h_1, h_2, then learn h_3 with the mistake.

The committee simply perform a majority vote by $h_1 + h_2 + h3$. One can use this method to get a lot of committees and further form bigger committees. Still, this gives you $\sum_k h_k$.

Later people discovered this adpative boosting, which gives a voting weight β_k to each classifier h_k.

3.5 Classification and clustering

3.6 Mixture model, EM and k-means

4 Monte Carlo

4.1 Random number generators

4.2 Monte Carlo integration

4.3 Markov chain

The Markov property is such that given the present, the future is independent of the past. Let the state space be $\mathcal{X}$, a Markov chain on $\mathcal{X}$ is determined by the transition probability $K(x, y) = P(X_{t+1} = y | X_t = x, X_{t-1}, ..., X_0) = P(X_{t+1} = y | X_t = x)$.

Let $p^{(t)}(x)$ be the marginal distribution of X_t. Then $p^{(t+1)}(y) = P(X_{t+1} = y) = \sum_x P(X_{t+1} = y, X_t = x) = \sum_x P(X_{t+1} = y|X_t = x)P(X_t = x) = \sum_x p^{(t)}(x)K(x, y)$. Let K be the matrix $(K(x, y))$. Let $p^{(t)}$ be the row vector $(p^{(t)}(x))$. Then $p^{(t+1)} = p^{(t)}K$. By induction, $p^{(t)} = p^{(0)}K^t$.

Under very general conditions, $p^{(t)} \to \pi$, the stationary distribution, so that $\pi = \pi K$, i.e., $\pi(y) = \sum_x \pi(x)K(x, y)$.

The two-step transition $K^{(2)}(x, y) = P(X_{t+2} = y|X_t = x) = \sum_z P(X_{t+2} = y, X_{t+1} = z|X_t = x) = \sum_z P(X_{t+2} = y|X_{t+1} = z, X_t = x)P(X_{t+1} = z|X_t = x) = \sum_z K(x, z)K(z, y)$. Thus $K^{(2)} = K^2$. In general, $K^{(t)} = K^t$.

Population immigration. The way to visualize the Markov chain is to consider a population of say 1 million people moving around in the state space $\mathcal{X}$. Then $p^{(t)}(x)$ can be considered the number of people in state x at time t. $p^{(t)}$ can be considered the distribution of the population at time t. π is the population distribution in the limit. $K(x, y)$ can be considered the fraction of the people in x who will go to y. The above calculations can then be interpreted in terms of numbers of people and fractions.

A special case is the reversible Markov chain where $\pi(x)K(x, y) = \pi(y)K(y, x)$ for every (x, y). It is also called the detailed balance condition. If a chain is reversible with respect to π, then π is the stationary distribution, because $\sum_x \pi(x)K(x, y) = \sum_x \pi(y)K(y, x) = \pi(y) \sum_x K(y, x) = \pi(y)$. The reversibility says that at stationarity, the number of people moving from x to y is the same as the number of people moving from y to x. If this is true for every pair of (x, y), then the numbers of people in different states will not change. That is, detailed balance implies overall balance.

Suppose we want to sample from a target distribution π, we may design an iterative algorithm that implements the Markov transition $P(X_{t+1} = y|X_t = x) = K(x, y)$, which changes x into y by some simple modification. If K maintains π, i.e., $\pi = \pi K$, then in the long run X_t can be considered a random sample from π. There are two major paradigms for designing the Markov chain.

4.4 Metropolis algorithm

The idea is to start from a base chain $B(x, y)$. Suppose $X_t = x$. We then propose to move to y by sampling y from $B(x, y)$. We accept this proposal with probability $\min[1, (\pi(y)B(y, x))/(\pi(x)B(x, y))]$. If the proposal is accepted, then $X_{t+1} = y$. Otherwise, $X_{t+1} = x$.

The intuition behind this algorithm is as follows. Suppose a population of 1 million people move around in the state space. If the population distribution is already π, then we want the distribution to remain π after the Markov transition. Now consider two states x and y. According to $B(x, y)$, the number of people who propose to move from x to y is $\pi(x)B(x, y)$. The number of people who propose to move from y to x is $\pi(y)B(y, x)$. Suppose $\pi(x)B(x, y) \geq \pi(y)B(y, x)$. Then we can allow all the $\pi(y)B(y, x)$ people who propose to move from y to x to make the move. But for those $\pi(x)B(x, y)$ people who propose to move from x to y, we can only allow $\pi(y)B(y, x)$ people to make the move in order to maintain the detailed balance. That is, for each person who proposes such a move, we allow this person to make the move with probability $(\pi(y)B(y, x))/(\pi(x)B(x, y))$. This will maintain the detailed balance between every two states, and thus it will maintain the stationarity.

As an example, consider sampling from $\pi(x) \propto \exp(-x^2/2)$ defined on all integers. Suppose the base chain is a random walk on the integers, that is, if $X_t = x$, we propose to change it to $y = x + 1$ or $y = x - 1$ with probability $1/2$. We accept this proposal with probability $\min[1, \exp(x^2/2 - y^2/2)]$. Here $B(x, y) = B(y, x) = 1/2$. If the proposal is accepted, $X_{t+1} = y$. Otherwise, $X_{t+1} = x$. This example illustrates that we only need to know $\pi(x)$ in proportionality.

4.5 Langevin dynamics

4.6 Simulated annealing

4.7 Gibbs sampler

The Gibbs sampler is designed to sample from multi-dimensional distribution $p(x_1, ..., x_d)$. The algorithm is very simple. In each iteration, for $i = 1, ..., d$, we sample $X_i \sim p(x_i|x_{-i})$, where the notation x_{-i} denotes the *current* values of all the other components except i. Such a move preserves the target distribution. Suppose $X \sim p(x)$, $x = (x_1, ..., x_d)$. Then $p(x) = p(x_{-i})p(x|x_{-i})$. In stationarity, $X_{-i} \sim p(x_{-i})$, and $[X_i|X_{-i}] \sim p(x_i|x_{-i})$. If we change X_i by another independent copy from the same distribution, we are not going to change the joint distribution.

As an example, consider sampling from a bivariate normal distribution $p(x, y)$ with correlation ρ. Let (X_t, Y_t) be the current values of (X, Y), then we sample $X_{t+1} = \rho Y_t + \epsilon_1$, where $\epsilon_1 \sim \mathrm{N}(0, 1 - \rho^2)$. After that we sample $Y_{t+1} = \rho X_{t+1} + \epsilon_2$, where $\epsilon_2 \sim \mathrm{N}(0, 1 - \rho^2)$, and both ϵ_1 and ϵ_2 are independent of anything else.

We can visualize the Gibbs sampler for a special case where the target distribution is a uniform distribution over a two-dimensional shape (such as a big island). Suppose one million people start from a certain point. Then at each iteration, we let people randomly relocate horizontally and then vertically. Then people will start to diffuse, and gradually will reach the uniform distribution on the island. At this point, horizontal and vertical re-shuffling of people will not change the population distribution.

4.8 Particle swarm optimization