Bios/CS 534 Project 2

Chenxi Cai

March 28, 2018

1 Problem 1

With perceptron function, the error rate is

$$0.033333 \tag{1}$$

Python codes of problem 1:

```python
import numpy as np
import pandas as pd

#define functions of prediction
def predict(x,w,y):
    '''
    :param x: one row of data samples
    :param w: data weight
    :param y: one row of data labels
    :return: predict label
    '''
    y_p = np.dot(x,w)*y
    return y_p
#define function of perceptron
def perceptron(X,w,Y,lrate):
    '''
    :param X: data samples
    :param w: data weight
    :param Y: data labels
    :return: weight vector as a numpy array
    '''
    flag = True
    while(flag):
        total_error = 0
        for i, x in enumerate(X):
            x = np.insert(x,2,1) # Add bias = 1 in x
            y_p = predict(x,w,Y[i])
            if (y_p <= 0):
                total_error += 1
                w = w + lrate*x*Y[i] #new weight,lrate = learning rate
        if (total_error == 0):
            flag = False #loop until all training data are predict correct
    return w

#prediction of testing data and error rate
def Error_rate(X2,w,Y2):
    '''
    :param X2: Testing data samples
    :param w: data weight
    :param Y: Testing data labels
    :return: error_rate
    '''
    s = Y2.shape[0]
    n = 0
```

```
    for i, x in enumerate(X2):
        x2 = np.insert(x,2,1)
        y2_p = predict(x2,w,Y2[i])
        if (y2_p <= 0):#error_rate = #points whose predict class is different form original one/#total points
            n += 1
    return n/float(s)

#read files
f = pd.read_csv('/Users/ccai28/Desktop/hw2_data_1.txt',sep='\t', header = None, skiprows = 1)

#read training data
X1 = f.loc[:69,[0,1]].values
Y1 = f.loc[:69,[2]].values

#read testing data
X2 = f.loc[70:,[0,1]].values
Y2 = f.loc[70:,[2]].values

#set learning rate and initial weight
lrate = 1
w = np.ones(3)

#new weight after training
w = perceptron(X,w,Y,lrate)

#Error rate
error_rate = Error_rate(X2,w,Y2)

print('perceptron: \nerror rate = {:f} \n'.format(error_rate))
```

2 Problem 2

With a function of Adaboost, the error rate is shown in Table below

Iteration	3	5	10	20
Error rate	0.167	0.167	0.133	0.133

Plot of Error rate vs iteration is shown as Fig. 1

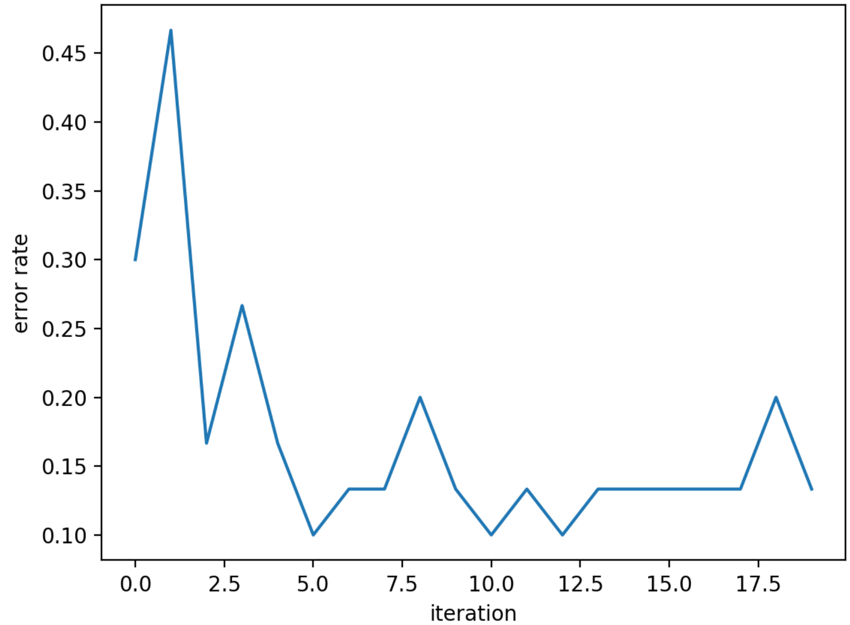

Figure 1: Plots of Adaboost error rate

Python codes of problem 2:

```python
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt

#define functions of classifier
def lineclassify(value,x,label):
    '''
    :param value: double,the standard value to compare with
    :param x:  training data,double,data axis value
    :label: 0,left; 1,right
    :left : x <= value: predict y is -1; x > value: predict y is 1
    :right: x <= value: predict y is 1; x > value: predict y is -1
    :return: predict y,int(-1 or 1)
    '''
    if (label == 0):
        if (x <= value):
            y_p = -1
        else:
            y_p = 1
    elif (label == 1):
        if (x <= value):
            y_p = 1
        else:
            y_p = -1

    return y_p


#Classify all training data with one weak classfier (xij)(loop all training data with specific xij)
def Weighterror(value,X,Y,W,label):
    '''
```

```
    :param value: double,the standard value to compare with
    :param X:  training data,list (n*1),data set in one axis (eg.X[:,0])
    :param Y:  training data,list (n*1),all data labels
    :param W: list (n*1),weight set of data
    :label: 0,left; 1,right
    :left : x <= value: predict y is -1; x > value: predict y is 1
    :right: x <= value: predict y is 1; x > value: predict y is -1
    :return: double, weighterror = sum(misclassified data's value on one axis * weight)
    '''
    weighterror = 0
    for i, x in enumerate(X):
        y_p = lineclassify(value,x,label)
        y_dot = y_p*Y[i]

        if (y_dot <= 0):
            weighterror += W[i]

    return weighterror

#The sum of weight
def Sumweight(W):
    '''
    :param W: list (n*1),weight set of data (all equal to 1/N),N is number of training data
    :return: double,sum of weight
    '''
    sumweight = 0
    for i, w in enumerate(W):
        sumweight += W[i]
    return sumweight

#new weight after iteration
def changeweight(value, X_line ,Y,W, label ,a):
    '''
    :param value: double,the best value to compare with
    :param X:  training data,list (n*1),data set in one axis (eg.X[:,0])
    :param Y:  training data,list (n*1),all data labels
    :param W: list (n*1),weight set of data
    :para a: alpha wi = wi*exp(a)
    :label: 0,left; 1,right
    :left : x <= value: predict y is -1; x > value: predict y is 1
    :right: x <= value: predict y is 1; x > value: predict y is -1
    :return: list ,new weight
    '''
    for i,x in enumerate(X_line):
        y_p = lineclassify(value, X_line[i],label)
        y_dot = y_p*Y[i]

        if (y_dot <= 0):
        # print W[j],alpha[i]
            W[i] = W[i]*np.exp(a)#calculate new wi

        else:
            continue
    return W

#Find the best xij to classify traning data (loop for every possible xij)
def Buildline(X,Y,W):
    '''
    :param X: training data,list (n*2),data set in all axis (X[xi,x2])
    :param Y: training data,list (n*1),all data labels
    :param W: list (n*1),weight set of data
    :return: list,best xij with smallest weighterror and its label in l/r, X1/X2
             x_best = [best value,[l(0)/r(1),X1(0)/X2(1)]
    :label: left:0 ; right:1; X1:0 ; X2:1
    '''
```

```
    weight_x1_l = []
    weight_x1_r = []
    weight_x2_l = []
    weight_x2_r = []
    weight_x = []
    x_best = []
    X_list = []

    X1 = X[:,0] #all x1
    X2 = X[:,1] #all x2

    X_list.append(X1)
    X_list.append(X1)
    X_list.append(X2)
    X_list.append(X2) #X_list = [[X1],[X1],[X2],[X2]]


    for i, x1 in enumerate(X1):
        weighterror1_l = Weighterror(X1[i],X1,Y,W,0) #weighterror of X1,left
        weight_x1_l.append(weighterror1_l)

        weighterror1_r = Weighterror(X1[i],X1,Y,W,1)
        weight_x1_r.append(weighterror1_r) #weighterror of X1,right

    weight_x.append(weight_x1_l) #put (l,x1) in list, index = 0
    weight_x.append(weight_x1_r) #put (r,x1) in list, index = 1


    for i, x2 in enumerate(X2):
        weighterror2_l = Weighterror(X2[i],X2,Y,W,0) #weighterror of X2,left
        weight_x2_l.append(weighterror2_l)
        weighterror2_r = Weighterror(X2[i],X2,Y,W,1) #weighterror of X2,right
        weight_x2_r.append(weighterror2_r)


    weight_x.append(weight_x2_l) #put (l,x2) in list, index = 2
    weight_x.append(weight_x2_r) #put (r,x2) in list, index = 3
                                  #weight_x = [[weight_x1_l], [weight_x1_r], [weight_x2_l], [weight_x2_r]]

    weight_x_min_list = np.min(weight_x, axis=1) #minimum of every (l/r,x1/x2) list
                                                      #weight_x_min_list = [min(weight_x1_l),min(weight_x1_r),min(weight_x2_l),min(weight_x2_r)]

    weight_x_min = min(weight_x_min_list)#minimum in all list, smallest weighterror

    index_list = np.argmin(weight_x_min_list) #index of four lists who has the smallest weighterror
                                                 #index_list = the index in weight_x_min_list
                                                 #= the index in weight_x = best value with smallest weighterror in which list, (l/r,x1/x2),between 0 and 3

    label_list = [[0,0],[1,0],[0,1],[1,1]] #build label list to represent all (l/r,x1/x2)
                                               #label_list = [[l,X1],[r,X1],[l,X2],[r,X2]]
    label = label_list[index_list] #find the label(0 or 1) of smallest weighterror

    index = weight_x[index_list].index(weight_x_min)#index of best value,0-69

    best_value = X_list[index_list][index]

    x_best.append(best_value)
    x_best.append(label) #x_best = [best value,[l(0)/r(1),X1(0)/X2(1)]
```

```python
                        #best value = x_best[0], l/r = x_best[1][0],X1/X2 = x_best[1][1]

    return x_best

#Build Adaboost training
def Adaboost(iteration,X,Y,W,X_test,Y_test):
    '''
    :param iteration: times loop weak lineclassify
    :param X: list (n*2),training data set in all axis (X[xi,x2])
    :param Y: list (n*1),training data, all data labels
    :param X_test: list (n*2),testing data set in all axis (X[xi,x2])
    :param Y_test: list (n*1),testing data, all data labels
    :param W: list (n*1),initial weight set of data (all equal to 1/N),N is number of training data
    :return: list, error rate, contains every iteration error rate
    '''
    error_rate = []

    Y_testp = np.zeros(Y_test.shape[0])
    for i in range(iteration):
        n = 0
        x_best = Buildline(X,Y,W)

        value, label_index ,line_index = x_best[0],x_best[1][0],x_best[1][1] #value, label_index ,line_index = best value, l/r, X1/X2
        X_line = X[:,line_index] #x1 best: X_line = X[:,0];x2 best: X_line = X[:,1];

        weighterror = Weighterror(value,X_line,Y,W,label_index)

        sumweight = Sumweight(W)
        error = weighterror/float(sumweight)  #calculate error
        print ("errorm:")
        print error

        #print weighterror,sumweight,error
        a = np.log((1-error)/float(error))  #calculate alpha
        #print error,a,value_best[i]

        W = changeweight(value,X_line,Y,W,label_index,a) #new weight

        s_test = X_test.shape[0] #number of data in X_test
        for j in range(s_test):
            x_test = X_test[:,line_index][j]
            Y_testp[j] += AdaClassify(a,x_best,x_test)

            y_dot = Y_testp[j] *Y_test[j]
            if (y_dot <= 0):
                n += 1
        error_rate.append(n/float(s_test)) #error rate of every iteration
        print error_rate[i]

    return error_rate

#Adaboost classify(combination of weak classifier)
def AdaClassify(alpha,value_best,x_test):
    '''
    :param alpha: to build adaboostc classifier: sum(alpha[i]*Gi(x))
    :param value_best: list,= [[best value1,[l(0)/r(1),X1(0)/X2(1)],[[best value2,[l(0)/r(1),X1(0)/X2(1)]]...]
    :param x_test: element ,testing data of one axis
    :return: predict y with alpha
    '''
    value, label_index ,line_index = value_best[0],value_best[1][0],value_best[1][1]  #value, label_index ,line_index = best value, l/r, X1/X2
    y_testp = lineclassify(value,x_test,label_index)*alpha #adaboost classifier
```

```

    return y_testp

#read files
f = pd.read_csv('/Users/ccai28/Desktop/hw2_data_1.txt',sep='\t', header = None, skiprows = 1)

#read training data
X = f.loc[:69,[0,1]].values
Y = f.loc[:69,[2]].values

#read testing data
X_test = f.loc[70:,[0,1]].values
Y_test = f.loc[70:,[2]].values

#number of training data
s = X.shape[0]

#set initial weight
W = np.full((s,1),1/float(s))


#iteration
iteration = 20

error_rate = Adaboost(iteration,X,Y,W,X_test,Y_test)
print error_rate[iteration-1]

x_plot = list(range(iteration))

plt.plot(x_plot, error_rate)
plt.xlabel('iteration')
plt.ylabel('error rate')
plt.show()
```

3 Problem 3

By using SVM in sklearn, the best parameter and error rate of three kernel methods are shown in Table below

	Radical kernel	Sigmoid kernel	Polynomial kernel
Best parameter (gamma/degree)	0.02	0.06	2
Error rate	0.443	0.493	0.426

Plots of Scores vs gamma or degree of three kernels are shown as Fig. 2, 3, 4

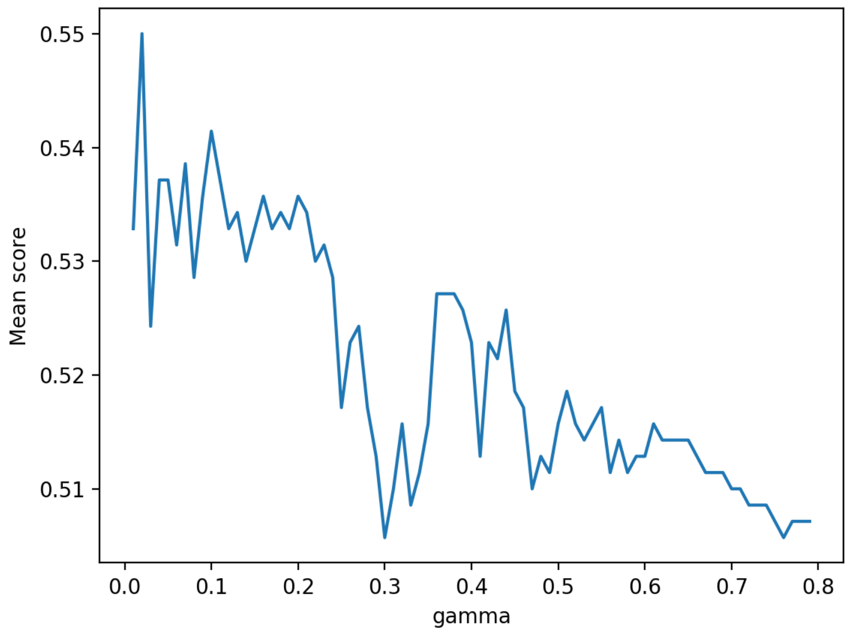

Figure 2: Plots of radical kernel

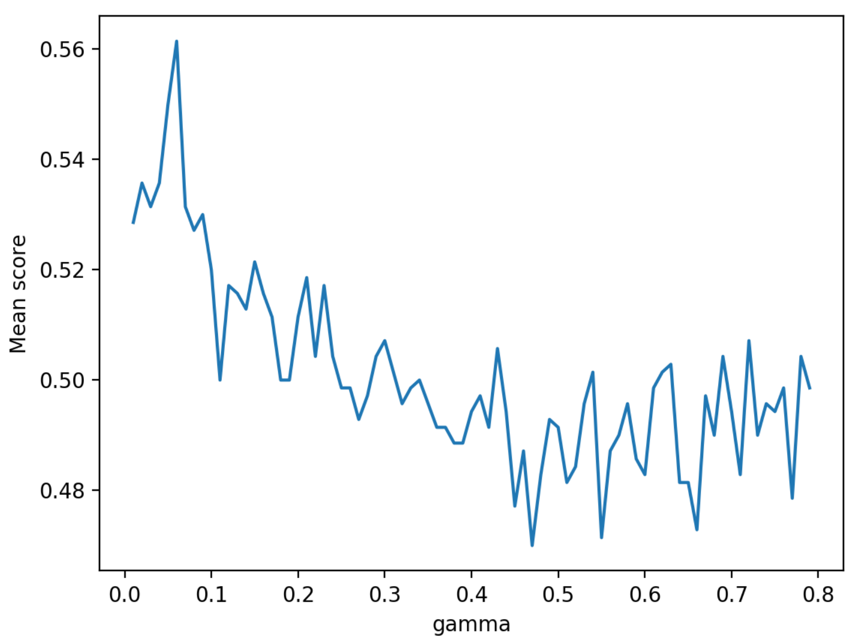

Figure 3: Plots of sigmoid kernel

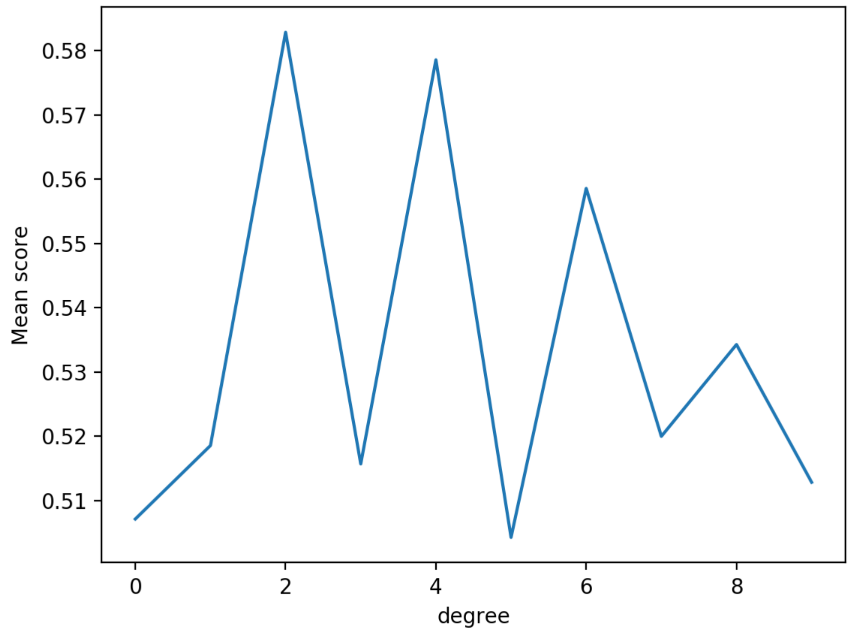

Figure 4: Plots of polynomial kernel

Python codes of problem 3:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.svm import SVC
from sklearn.grid_search import GridSearchCV

def SVM(param_grid, cv, kernel):
    '''
    :param param_grid: parameter in SVM grid
    :param cv : cv-fold
    :param kenel: string, thr name of kernel
    :return: svm grid
    '''
    grid = GridSearchCV(SVC(kernel=kernel), param_grid, cv = cv)
    return grid

def Best_estimator(grid, X, Y):
    '''
    :param grid: SVM grid
    :param X : Training data X
    :param Y : Training data Y
    :return: best estimator after fiting training data. It has smallest error
    and best parameter(gamma or degree)
    '''
    grid.fit(X, Y)
    best_estimator = grid.best_estimator_
    print("The best classifier is: ", best_estimator)
    return best_estimator

def plot(grid, C_range, para_range, s):
    '''
    :param grid: SVM grid
```

```
    :param C_range : range of C parameter
    :param para_range : range of gamma or degree parameter
    :param s : string, name of parameter('gamma' or 'degree')
    :return: plot of score vs parameter
    '''
    scores = [x[1] for x in grid.grid_scores_]

    #plot
    plt.plot(para_range, scores)

    plt.legend()
    plt.xlabel(s)
    plt.ylabel('Mean score')
    plt.show()

def Error_rate(X,Y,X2,Y2,best_estimator):
    '''
    :param best_estimator: best_estimator with best parameter
    :param X: list (n*20), training data set in all axis (X[xi,x2])
    :param Y: list (n*1), training data, all data labels
    :param X2: list (n*20), testing data set in all axis (X[xi,x2])
    :param Y2: list (n*1), testing data, all data labels
    :return: error_rate
    '''
    s = X2.shape[0]
    n = 0
    best_estimator.fit(X,Y)
    Y_p = best_estimator.predict(X2)
    for i, y in enumerate(Y_p):
        if (y != Y2[i]):#error_rate = #points whose predict class is different form original one/#total points
            n += 1
    return n/float(s)


#read files
f = pd.read_csv('/Users/ccai28/Desktop/hw2_data_2.txt',sep='\t', header = None, skiprows = 1)

#read training data
X = f.loc[:699,:19].values
Y = f.loc[:699,20].values

#read testing data
X_test = f.loc[700:,:19].values
Y_test = f.loc[700:,20].values


#set parameter for Gridsearch
C_range = [1]
gamma_range = np.arange(0.01, 0.8,0.01)
degree_range = np.arange(0, 10,1)

param_grid_g = dict(gamma=gamma_range)
param_grid_d = dict(degree=degree_range)
cv = 10

#radical kernel
grid_r = SVM(param_grid_g,cv,'rbf')
best_estimator_r = Best_estimator(grid_r,X,Y)
plot(grid_r,C_range,gamma_range,'gamma')

error_rate_r = Error_rate(X,Y,X_test,Y_test,best_estimator_r)
print('radical kernel: \nerror rate = {:f} \n'.format(error_rate_r))


#sigmoid kernel
grid_s = SVM(param_grid_g,cv,'sigmoid')
```

```
best_estimator_s = Best_estimator(grid_s,X,Y)
plot(grid_s,C_range,gamma_range,'gamma')

error_rate_s = Error_rate(X,Y,X_test,Y_test,best_estimator_s)
print('sigmoid kernel: \nerror rate = {:f} \n'.format(error_rate_s))

# polynomial kernel
grid_p = SVM(param_grid_d,cv,'poly')
best_estimator_p = Best_estimator(grid_p,X,Y)
plot(grid_p,C_range,degree_range,'degree')

error_rate_p = Error_rate(X,Y,X_test,Y_test,best_estimator_p)
print('polynomial kernel: \nerror rate = {:f} \n'.format(error_rate_p))
```

4 Problem 4

The plot of OOB error rate and number of trees are shown in Fig. 5,

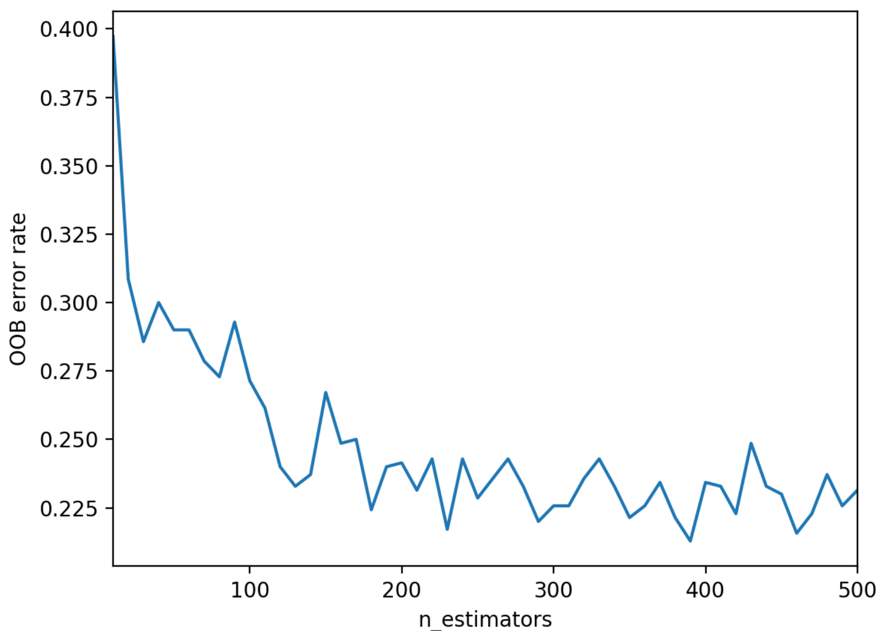

Figure 5: plots of OOB error rate vs number of trees

The selected number of trees is 210, and the error rate is

$$0.223333 \tag{2}$$

The feature ranking is:

rank	index of variables	importances
1	4	(0.129395)
2	9	(0.119601)
3	14	(0.101843)
4	0	(0.041882)
5	10	(0.041192)
6	2	(0.041054)
7	12	(0.039885)
8	1	(0.039104)
9	16	(0.038682)
10	3	(0.038664)
11	11	(0.038254)
12	7	(0.038123)
13	5	(0.037836)
14	19	(0.037576)
15	18	(0.037312)
16	8	(0.036664)
17	6	(0.036404)
18	13	(0.036342)
19	15	(0.035886)
20	17	(0.034302)

Python codes of problem 4:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from sklearn.ensemble import RandomForestClassifier


def Error_rate(X2,Y2,forest):
    '''
    :param forest: forest with best parameter
    :param X2:  (n*20),testing data set in all axis (X[xi,x2])
    :param Y2:  (n*1),testing data, all data labels
    :return: error_rate
    '''
    s = X2.shape[0]
    n = 0
    Y_p = forest.predict(X2)
    for i, y in enumerate(Y_p):
        if (y != Y2[i]):#error_rate = #points whose predict class is different
         form original one/#total points
            n += 1
    return n/float(s)

#read files
f = pd.read_csv('/Users/ccai28/Desktop/hw2_data_2.txt',sep='\t', header = None
    , skiprows = 1)

#read training data
X = f.loc[:699,:19].values
Y = f.loc[:699,20].values

#read testing data
X_test = f.loc[700:,:19].values
Y_test = f.loc[700:,20].values

# Range of n_estimators values to explore.
min_estimators = 10
max_estimators = 500
step = 10
```

```
number_estimators = (max_estimators - min_estimators)/step #how many trees are tried

print number_estimators

error_rate = []
n_estimators = []

#use Random Forest
#oob_score : bool (default=False) :Whether to use out-of-bag samples to estimate the generalization accuracy.
for i in range(min_estimators, max_estimators + step,step):
    clf = RandomForestClassifier(oob_score=True)#warm_start=True
    clf.set_params(n_estimators=i)
    clf.fit(X, Y)

    # Record the OOB error for each n_estimators=i setting.
    oob_error = 1 - clf.oob_score_
    n_estimators.append(i)
    error_rate.append(oob_error)
    print i,oob_error

last_oob_error = error_rate[number_estimators] #oob error of n = 500

stabilize_estimator = 0
#number of trees based on when the OOB error rate first stabilizes
for i in range(number_estimators):
    diff_rate = abs((error_rate[i] - last_oob_error)/float(last_oob_error))
    print i, diff_rate
    if (diff_rate <=0.002):
        stabilize_estimator = n_estimators[i]
        print stabilize_estimator
        break

print stabilize_estimator #the stablize tree number

# Generate the "OOB error rate" vs. "n_estimators" plot.
plt.plot(n_estimators, error_rate)

plt.xlim(min_estimators, max_estimators)
plt.xlabel("n_estimators")
plt.ylabel("OOB error rate")
plt.legend()
plt.show()

# Build a forest and compute the feature importances
forest = RandomForestClassifier(n_estimators = stabilize_estimator)
forest.fit(X,Y)

#importance and feature
importances = forest.feature_importances_
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")
for f in range(X_test.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

error_rate = Error_rate(X_test,Y_test,forest)
print('Random Forest: \nerror rate = {:f} \n'.format(error_rate))
```

5 Problem 5

By running gradient boosting with deviance loss on the training data, the error rate is

$$0.193333 \tag{3}$$

The feature ranking is:

rank	index of variables	importances
1	9	(0.254537)
2.	4	(0.241189)
3	14	(0.215034)
4	1	(0.034982)
5	10	(0.031280)
6	3	(0.023650)
7	7	(0.023579)
8	6	(0.020974)
9	2	(0.020797)
10	18	(0.020200)
11	5	(0.017700)
12	8	(0.015834)
13	0	(0.015096)
14	17	(0.012983)
15	13	(0.011701)
16	11	(0.010080)
17	19	(0.008941)
18	16	(0.008900)
19	15	(0.007335)
20	12	(0.005209)

Python codes of problem 5:

```
import pandas as pd
import numpy as np

from sklearn.ensemble import GradientBoostingClassifier

def Error_rate(X_test, Y_test, grid):
    '''
    :param grid: grid after fit by training data
    :param X_test: (n*20),testing data set in all axis (X[xi,x2])
    :param Y_test: (n*1),testing data, all data labels
    :return: error_rate
    '''
    s = X_test.shape[0]
    n = 0
    Y_p = grid.predict(X_test)
    for i in range(s):
        if (Y_p[i] != Y_test[i]):#error_rate = #points whose predict class is
    different form original one/#total points
            n += 1
    score = grid.score(X_test, Y_test)
    return 1-score

#read files
f = pd.read_csv('/Users/ccai28/Desktop/hw2_data_2.txt',sep='\t', header = None
    , skiprows = 1)

#read training data
X_train = f.loc[:699,:19].values
Y_train = f.loc[:699,20].values

```

```
#read testing data
X_test = f.loc[700:,:19].values
Y_test = f.loc[700:,20].values

#use gradient boosting with deviance loss
grid = GradientBoostingClassifier(loss = 'deviance')
grid.fit(X_train, Y_train)

#importance and feature
importances = grid.feature_importances_
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")
for f in range(X_test.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

error_rate = Error_rate(X_test, Y_test, grid)
print('gradient boosting: \nerror rate = {:f} \n'.format(error_rate))
```

6 Problem 6

With MARS, the error rate is

$$0.24333 \tag{4}$$

Python codes of problem 6:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

from pyearth import Earth

def Error_rate(X_test, Y_test, model):
    '''
    :param X_test: (n*20),testing data set in all axis (X[xi,x2])
    :param Y_test: (n*1),testing data, all data labels
    :return: error_rate
    '''
    s = X_test.shape[0]
    n = 0
    Y_p = model.predict(X_test)
    y_median = median(Y_p)
    for i in range(s):
        if (Y_p[i] <= y_median):
            y_p = -1
        else:
            y_p = 1
        if (y_p != Y_test[i]):#error_rate = #points whose predict class is different form original one/#total points
            n += 1
    return n/float(s)

def median(Y_test):
    '''
    :param Y_test:  (n*1),predict Y, all data labels
    :return: median value of all predict Y
    '''
    y_median = np.median(Y_test)
    return y_median

#read files
f = pd.read_csv('/Users/ccai28/Desktop/hw2_data_2.txt',sep='\t', header = None, skiprows = 1)
```

```

#read training data
X_train = f.loc[:699,:19].values
Y_train = f.loc[:699,20].values

#read testing data
X_test = f.loc[700:,:19].values
Y_test = f.loc[700:,20].values

#MARS from earth class
model = Earth() #MARS
model.fit(X_train, Y_train)

#error_rate
error_rate = Error_rate(X_test, Y_test, model)
print('MARS: \nerror rate = {:f} \n'.format(error_rate))
```