HW2 Gradient_Boosting_K_Class

October 14, 2018

1 T1

1.1 The first part is the implement of the algorithm as a class in python. I use the sub-model as regression tree with max depth 2.

1.2 the second part is the apply the algorithm on the dataset specified in the HW1

1.3 I also plot the Accuracy vs number of iterations in the plot.

1.4 and print out the Accuracy & number of iterations

In [1]:
```
# coding: utf-8
%matplotlib inline
# import the basic packages for the problem
# no fancy packages.
import numpy as np
from sklearn import tree
# this package is used to plot the result
import matplotlib.pyplot as plt

# Here I define a class as an agent to do
# Gradient Boosting for K-Class Classification
class K_GBoost():
    def __init__(self,X,y,M,element):
        # Input:
        # X: X train matrix dimention: n by d
        # y: y train: n by 1
        # M: integer, the loop number
        # init: initial f vector: K by 1
        self.X = X
        self.y = y
        self.Max_iter = M
        self.K = len(element)
        self.element = element


    @staticmethod
    # this function is weighted sum
```

```
# to compute fk
def f_val(f_set, X,k):
    f = 0
    for i in range(len(f_set)):
        f += f_set[i](X,k,i)
    return f

@staticmethod
# this method is aimed to compute the P matrix
def p(k,f):
    # f.dtype = "float64"
    # print(f)
    return np.exp(f) / np.sum(np.exp(f),1).reshape(-1,1)

@staticmethod
# this funtion is aimed to convert the y vector
# into the indicator vector
def y_indicator(y, k, element):
    z = np.array(y)
    targat = element[k]
    index = z == targat
    z[index] = 1
    rev = np.array([not i for i in index]).reshape(-1,)
    z[rev] = 0
    return z
@staticmethod
# this method is try to compute the current f:
# i.e. f_km
def cur_f(reg_tree, gamma, X, reg_room):
    pred = reg_tree.predict(X)
    f_val = np.zeros(X.shape[0])
    for i, val in enumerate(pred):
        in_index = reg_room == val
        f_val[i] = gamma[in_index]
    return f_val

def fit(self):
    ## Here preallocate some 2-D list or Matrix we will use later
    # initial the F and P matrix, both are n by k
    F = list()
    # k_f_set is a collection of classifier for the k class
    # add the init classifier to the f_set, return 0 column
    for i in range(self.K):
        k_f_set = [lambda X,k,m: np.zeros(X.shape[0])] * self.Max_iter
        F.append(k_f_set)
    assert len(F) == self.K and len(F[0]) == self.Max_iter

    P = np.zeros((self.X.shape[0], self.K))
```

```
# these is to store the tree, gamma, rooms
reg_tree_set = [[0]*self.Max_iter for i in range(self.K)]
gamma_set = [[0] * self.Max_iter for i in range(self.K)]
reg_rooms_set = [[0] * self.Max_iter for i in range(self.K)]

# Set the P_k matrix in every iteration
for m in range(self.Max_iter):

    f = np.zeros((self.X.shape[0], self.K))
    for k in range(self.K):
        f[:,k] = self.f_val(F[k],self.X,k)
    for k in range(self.K):
        P = self.p(k, f)

    ## update f
    for k in range(self.K):
        # k_f_set = F[k]
        # f = self.f_val(k_f_set, self.X)
        # assert len(f) == self.X.shape[0]
        # # set pk
        # P[:,k] = self.p(f)
        pk = P[:,k].T

        y_k = self.y_indicator(self.y, k, self.element)
        rk = y_k - pk

        # Use a simple tree in each sub-classifier Max depth is 2.
        reg_tree_set[k][m] = tree.DecisionTreeRegressor(max_depth=2)
        reg_tree_set[k][m] = reg_tree_set[k][m].fit(self.X, rk)

        # all prediction value
        reg_X_train = reg_tree_set[k][m].predict(self.X)
        # reg_X_train = np.array(reg_X_train,dtype = "float16")
        # reg_room is the distinct regression value in increasing manner
        reg_rooms_set[k][m] = np.unique(reg_X_train)
        # preallocation the gamma vector
        gamma_set[k][m] = np.zeros(len(reg_rooms_set[k][m]))
        # compute the gamma_jkm
        for j, room_num in enumerate(reg_rooms_set[k][m]):
            cur_cluster = reg_rooms_set[k][m][j]
            # index of X_train fell into the j-th room
            index_X_in = reg_X_train == cur_cluster
            nu = sum(rk[index_X_in])
            de = sum(np.abs(rk[index_X_in])*(1-np.abs(rk[index_X_in])))
            gamma_set[k][m][j] = ((self.K - 1) / self.K) * (nu/de)
        # store the sub-f_km in the F matrix
        F[k][m]=lambda X,k,m: self.cur_f(reg_tree_set[k][m], gamma_set[k][m], X,
# Store the F matrix as global in the class
```

```
            self.f_model = F

        ## this method is to predict the f(X) given X matrix
        def predict(self,X):
            pred = np.zeros((X.shape[0],self.K))
            for k in range(self.K):
                k_model = self.f_model[k]
                pred[:,k] = self.f_val(k_model,X,k)
            return pred
```

```
In [2]: if __name__ == '__main__':

            # fix the seed
            np.random.seed(1)
            ## generate the dataset
            # the dimension and size of the data
            p = 10
            n_train = 2000
            n_test = 200


            # two distribution we sample data from
            mu1 = np.repeat(0.5,p)
            sigma1 = np.eye(p)
            mu2 = np.repeat(-0.5,p)
            sigma2 = np.eye(p)

            # construct the training dataset
            train_x1 = np.random.multivariate_normal(mu1,sigma1,n_train)
            train_y1 = np.repeat(1,n_train)

            train_x2 = np.random.multivariate_normal(mu2,sigma2,n_train)
            train_y2 = np.repeat(-1,n_train)

            train_x = np.vstack((train_x1,train_x2))
            train_y = np.hstack((train_y1,train_y2))

            print('the size of the training dataset is:\n',train_x.shape)
            print(train_y.shape)

            # constructing the testing dataset
            test_x1 = np.random.multivariate_normal(mu1,sigma1,n_test)
            test_y1 = np.repeat(1,n_test)

            test_x2 = np.random.multivariate_normal(mu2,sigma2,n_test)
            test_y2 = np.repeat(-1,n_test)

            test_x = np.vstack((test_x1,test_x2))
```

```python
test_y = np.hstack((test_y1,test_y2))

print('the size of the testing dataset is:\n',test_x.shape)
print(test_y.shape)

# define the distinct element & class K
element = -np.unique(train_y)
K = len(element)

# Make two List: iteration list and Acu list.
# Used to plot
M = np.arange(21)
Acu = np.zeros(len(M))

for trial,iter in enumerate(M):
    GBoost = K_GBoost(train_x,train_y,iter,element)
    # print("the Gradient Boosting model is training...")
    model = GBoost.fit()
    # print("prediction start...")
    pred = GBoost.predict(test_x)
    # print(pred)
    pred_class = GBoost.p(K,pred)
    # print(pred_class)
    # Accuracy
    pred_final = np.zeros((test_y.shape[0]))
    for i,val in enumerate(pred_class):
        if val[0]>val[1]:
            pred_final[i] = 1
        else:
            pred_final[i] = -1
    Acu[trial] = sum(pred_final==test_y)/len(test_y)
# print the iteration and corresponding Acu
print("the iteration List:\n",M)
print("the Accuracy List:\n",Acu)
# plot the result
plt.plot(M,Acu)
plt.xlabel("the iteration number")
plt.ylabel("the Accuracy")
plt.show()
```

```
the size of the training dataset is:
 (4000, 10)
(4000,)
the size of the testing dataset is:
 (400, 10)
(400,)
the iteration List:
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]
```

```
the Accuracy List:
 [0.5    0.6975 0.7175 0.805  0.84   0.85   0.875  0.8725 0.89   0.9025
 0.9025 0.9075 0.9025 0.8975 0.9    0.9    0.9025 0.9125 0.9175 0.9175
 0.9175]
```