National Taiwan University

Machine Learning: HW2

EE3 b03901016
陳昊

October 27, 2016

1 Logistic Regression

The concept of implementation of classifying in this program is based on **discriminative logistic regression**, using **gradient descent** to directly find the bias and weight of each feature of the model. Similar to linear regression, the only different between these two is the design of loss function, logistic regression calculate the **cross entropy**.

Algorithm 1 Compute Loss and Update parameters

Define model $y = sigmoid(b + \sum w_i x_i)$
Initialize all parameters $b,\ w_i,\ L,\ Grad\_of\_w_i = 0$
for $i < iteration$ **do**
    $L \leftarrow 0$
    for models in all data **do**
        $y \leftarrow b$
        for each parameter **do**
            $y \leftarrow y + w_i * x_i$
            $L \leftarrow L + cross\_entropy(\widehat{y}^n - y)$
        for each parameter **do**
            Compute G_i = gradient of w_i
            $Grad\_of\_w_i \leftarrow Grad\_of\_w_i + G_i$
        Update each parameter by **Adagrad**
return Final model

(This method of gradient descent include **stochastic gradient descent** and **Adagrad**)

2 Random Forest

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks. It correct for decision trees' habit of overfitting to their training set.

2.1 Tree bagging

To construct the training model of each tree in random forest, we randomly sample part of the training data and use it to train a tree. In addition to the randomization of the sample data, training features of of each tree are also randomized. In another word, different trees contain different training data and training features.

To have better performance, the data-size and features number of a `Tree` should be modified, in my design, the size of a `Tree` is randomly picked in range [2000, 2500], and the sample features number is in range [11, 14]. Another important design is the set of split-points of each feature, I compute the mean and variance of each feature base on training data, and sample several points using Gaussian distribution.

2.2 Build Tree

Now use the data and features that are already bagged to build a tree model. First, I construct a class `Node` and a class `Tree`, each `Node` contains some members, such as `feature`, `split-point`, `label`...etc; while a `Tree` contains several `Node`s.

When input a training data to a node, it will randomly pick some features, and each feature has a set of split points, to calculate the **Gini index** of each spilt-point. Assume there are m features, and for each feature there are n split-points, then it will calculate the **gini-index** and get $m * n$ different values, then pick the feature with the split-point having the highest score(i.e. Decrease gini-index most) and set the `Node` with that feature and the split-point. Recursively do this process until it cannot split any more. So when it iterate to the leaf of the tree, the label of the node will be set to 0 or 1.

2.3 Ensemble

Since we've build a lot of `Tree`s, and each of them have different structure, they combine a random forest. Then how do we decide the final classification of an input data? Just simply vote!

Input each testing data through all the `Tree`s, then each `Tree` will return a corresponding value (0 or 1) due to the model structure itself. If more trees say the testing is in class 0, we assume it is 0; while if more trees say it is 1, we assume it to be 1.

2.4 Implementation

```
#! /usr/bin/env python

import sys
import math
import random as rd

train_file = sys.argv[1]

########## setting ############################
Tree_Num = 512
split_Num = 100
```

```
data_range = [2500, 2500]
f_range = [11, 11]
validation = False
n_fold = 10

########## global var ########################
gb = 0
def resetgb():
    global gb
    gb = 0

########## define class ######################
class Node(object):
    def __init__(self, data):
        self.data = data
        self.feature = None
        self.splitpoint = None
        self.left = None
        self.right = None
        self.label = None
        self.index = 0

    def create_child(self, data, flag):
        if flag == 'l':
            self.left = Node(data)
        elif flag == 'r':
            self.right = Node(data)

    def setfeature(self, feature):
        self.feature = feature

    def setsplitpoint(self, splitpoint):
        self.splitpoint = splitpoint

    def setlabel(self, label):
        self.label = label

    def gini(self):
        f_num = rd.randint(f_range[0], f_range[1])
        f = rd.sample(range(0, 57), f_num)
        data = self.data
        index = [0, 0]
        max_con = 1
        for i in f:
            for j in range(len(f_finite[i])):
                lchild = []
                rchild = []
                l_ones = 0.
                r_ones = 0.
```

```
            for k in range(len(data)):
                if data[k][i] < f_finite[i][j]:
                    lchild.append(data[k])
                    if data[k][57] == 1:
                        l_ones += 1
                else:
                    rchild.append(data[k])
                    if data[k][57] == 1:
                        r_ones += 1
            if len(lchild) == 0 or len(rchild) == 0:
                continue

            l_t = float(len(lchild))
            r_t = float(len(rchild))
            l_zeros = float(l_t - l_ones)
            r_zeros = float(r_t - r_ones)
            GL = 1 - (l_ones / l_t)**2 - (l_zeros / l_t)**2
            GR = 1 - (r_ones / r_t)**2 - (r_zeros / r_t)**2
            contribute = GL * l_t / len(data) + GR * r_t / len(data)
            if contribute < max_con:
                index = [i, f_finite[i][j]]
                max_con = contribute
    if max_con == 1:
        index = [-1, -1]
    self.setfeature(index[0])
    self.setsplitpoint(index[1])
    return index

def __split(self):
    index = self.gini()
    data = self.data
    if index == [-1, -1]:
        a = 0.
        out = -1
        for i in range(len(data)):
            a += data[i][57]
        a /= len(data)
        if a > 0.5:
            out = 1
        else:
            out = 0
        self.setlabel(out)
        return
    else:
        lchild = []
        rchild = []
        for i in range(len(data)):
            if data[i][index[0]] < index[1]:
                lchild.append(data[i])
```

```
            else:
                rchild.append(data[i])
        self.create_child(lchild, 'l')
        self.create_child(rchild, 'r')
        self.left.__split()
        self.right.__split()

    def split(self):
        if self.label == None:
            self.__split()

    def find(self, sample):
        if self.label == None:
            if sample[self.feature] < self.splitpoint:
                return self.left.find(sample)
            else:
                return self.right.find(sample)
        else:
            return self

    def setindex(self):
        global gb
        self.index = gb
        gb += 1
        if self.label == None:
            self.left.setindex()
            self.right.setindex()
    def printindex(self):
        print self.index
        if self.label == None:
            self.left.printindex()
            self.right.printindex()

    def recursive_outfile(self, ofile):
        ofile.write(str(self.index) + ',')
        if self.label == None:
            ofile.write(str(self.left.index) + ',')
            ofile.write(str(self.right.index) + ',')
            ofile.write(str(-1) + ',') # self.label
        else:
            ofile.write(str(-99) + ',')
            ofile.write(str(-99) + ',')
            ofile.write(str(self.label) + ',')
        ofile.write(str(self.feature) + ',')
        ofile.write(str(self.splitpoint))
        ofile.write('\n')
        if self.label == None:
            self.left.recursive_outfile(ofile)
            self.right.recursive_outfile(ofile)
```

```
class Tree(Node):
    def __init__(self, d_set):
        self.d_set = d_set
        self.root = Node(d_set)
        self.root.split()
        self.root.setindex()
        #self.root.printindex()
        resetgb()
    def compare(self, sample):
        a = self.root.find(sample)
        return a.label
    def recursive_out(self, ofile):
        ofile.write('-')
        ofile.write('\n')
        self.root.recursive_outfile(ofile)
        ofile.write('-')
        ofile.write('\n')

############### parse training data ####################
train_table = []
with open(train_file, 'r') as f:
    for line in f:
        temp = []
        tokens = line.split(',')
        for i in range(len(tokens)-1):
            temp.append(float(tokens[i+1]))
        train_table.append(temp)

############### build split points ####################
col_ave = [0] * 57
for i in range(57):
    for j in range(len(train_table)):
        col_ave[i] += train_table[j][i]
    col_ave[i] /= len(train_table)
col_sigma = [0] * 57
for i in range(57):
    for j in range(len(train_table)):
        col_sigma[i] += (train_table[j][i] - col_ave[i])**2
    col_sigma[i] /= len(train_table)
    col_sigma[i] = col_sigma[i]**0.5

f_finite = []
for i in range(57):
    temp = [0] * split_Num
    for j in range(split_Num):
        temp[j] = rd.gauss(col_ave[i], col_sigma[i])
    f_finite.append(temp)
```

```
############### Grow Trees ##############################
Trees = []
for i in range(Tree_Num):
    d_num = rd.randint(data_range[0], data_range[1])
    d_set = rd.sample(range(0, 4001), d_num)
    data = []
    for j in d_set:
        data.append(train_table[j])
    print "Growing Tree:", i+1
    # n-fold validation
    if validation == True:
        T = None
        max_test = 0
        for j in range(n_fold):
            restset = []
            validset = []
            v_num = int(d_num) / 10
            v_set = rd.sample(d_set, v_num)
            for k in d_set:
                if v_set.count(k) == 0:
                    restset.append(train_table[k])
                else:
                    validset.append(train_table[k])
            temp = Tree(restset)
            acc = 0.
            for k in validset:
                value = temp.compare(k)
                if value == k[57]:
                    acc += 1
            acc /= len(validset)
            if acc > max_test:
                T = temp
                max_test = acc
            print acc
        print max_test
        Trees.append(T)
    else:
        Trees.append(Tree(data))

############## Calculate Training score ###############
cnt = 0.
for i in range(len(train_table)):
    value = 0.
    out = -1
    for j in range(len(Trees)):
        value += Trees[j].compare(train_table[i])
    if value / Tree_Num > 0.5:
        out = 1
    else:
```

```
        out = 0
    if out == train_table[i][57]:
        cnt += 1
print "Training Acc: ", cnt / len(train_table)


############## Output File ############################
ofile = open(sys.argv[2], 'w')
for i in range(len(Trees)):
    Trees[i].recursive_out(ofile)
print "model builded !!"
```

3 Experiment

The best score I got in **Logistic Regression** is 0.91667, which is not an satisfying score. While using **Random Forest**, I got better score, which has an best score of 0.96333. The number of `Tree`s in random forest effect the prediction result significantly, but the effect decrease when the number is larger and larger.

Tree-size	1	4	16	64	128	512	1024	4096
Training Acc	0.96626	0.98825	0.98925	0.99425	0.99650	0.99547	0.99500	0.99525
Public Acc	0.90000	0.92000	0.96000	0.96333	0.96333	0.96333	0.96333	0.96333

From the table, we can find that there are no significant change when the number of tree is larger than 128, perhaps just error cause by randomization. Consider the runtime and efficiency, 128 `Tree`s is enough for the model.