Introduction to AI Programming Assignment #4 Report

Student ID: 0716085　Name: 賴品樺

Description:

I use "**Optical Recognition of Handwritten Digits Data Set**" (https://archive.ics.uci.edu/ml/machine-learning-databases/optdigits/) in this program experiment. In this data set, it can be divided 2 different type of dataset, **optdigits** and **optdigits-orig**.

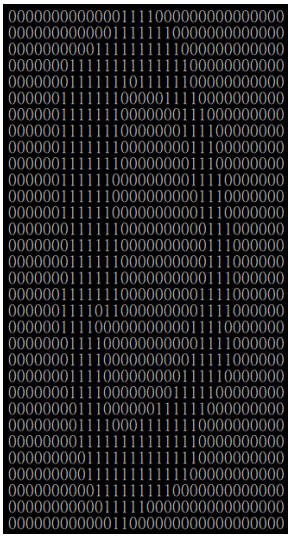
Each data in **optdigits** has 65 integers. The front 64 integers are represented to 8x8 picture and every integer is in range 0~16 (white~black). The last integer is the value of this picture. That is there is 64 different attributes.

Below is an example of **optdigits**:

00 00 10 14 08 01 00 00
00 02 16 14 06 01 00 00
00 00 15 15 08 15 00 00
00 00 05 16 16 10 00 00
00 00 12 15 15 12 00 00
00 04 16 06 04 16 06 00
00 08 16 10 08 16 08 00
00 01 08 12 14 12 01 00　　value = 8

Each data in **optdigits-orig** has 1025 integers. The front 1024 integers are represented to a 32x32 picture and every integer is 0 (white) or 1(black). The last integer is the value of this picture.

Below is an example of **optdigits-orig**:


value = 0

In this assignment, I mainly use **optdigits**. I use **optdigits.tra** as training dataset and **optdigits.tes** as validation subset.
NOTICE: Before execute the program, be sure the dataset files is in the same directory as the program.

Code interpretation:

The decision tree ***DT*** is composed of some ***Node*** type nodes.
In class ***Node*** there has 8 components. ***branch_t*** and ***branch_f*** are ***Node*** pointer pointing to the child of true condition and the child of false condition. ***dataset*** is a vector<***Sample***> that record the training data in present node and ***Sample*** is a struct recording 8x8 picture and the value. ***impurity*** and ***threshold*** are float variables recording gini impurity of present node and the threshold limit value to differentiate ***branch_t*** and ***branch_f***. ***attr*** is an integer recording the attribute that is the condition to do the partition. ***leaf*** is an integer recording the value of result if this object is leaf node. ***index*** is an integer recording the index of node, the rule of the index is that if present node has index k, then ***branch_t***'s index is *2k* and ***branch_f***'s index is *2k+1*. By this rule, we can understand the relationship between each node.
To build the decision tree ***DT***, initializing ***DT*** as the root node of this tree which has all training data from **optdigits.tra**. After call function ***BuildDecisionTree*** and pass the ***DT*** as argument, the function will recursively divide training data until the node is pure (impurity is zero). The rule of dividing is finding the optimal information gain of threshold and attribute pair.
To build a random forest, I only do the sample bagging part. I implement sample bagging by ***SampleBagging*** function. The experiment of random forest is based on the different in number of tree and the number of samples of each tree.

Testing result & Observation:

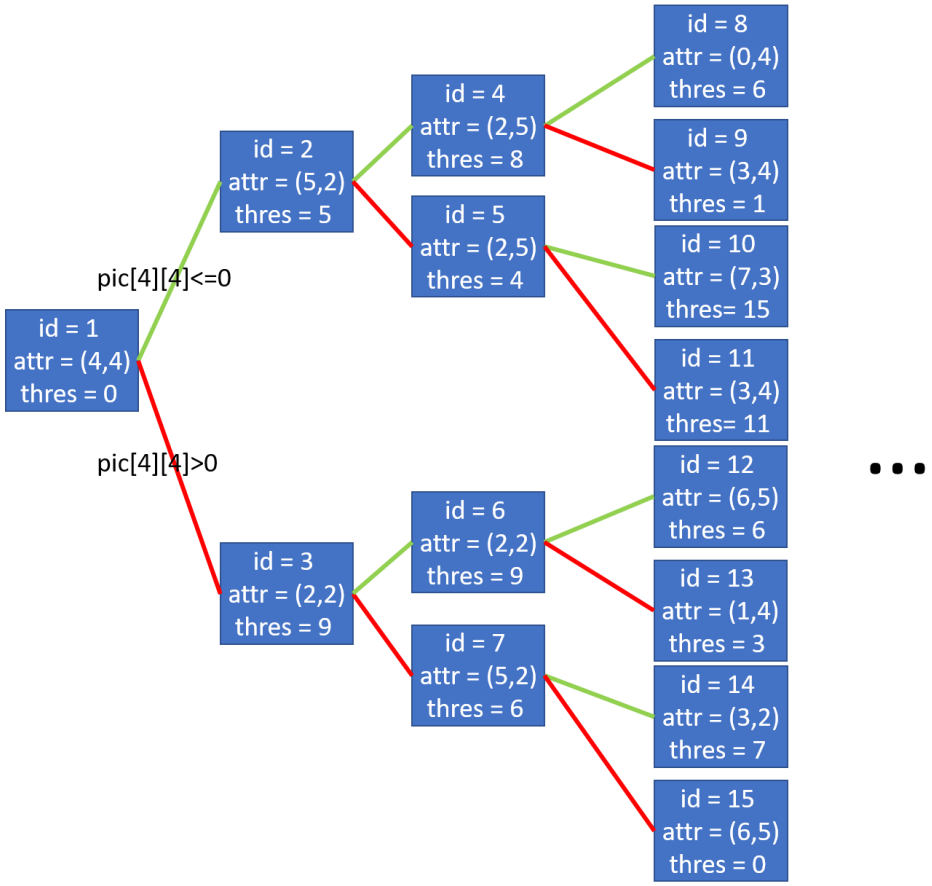
Decision Tree:
optdigits.tra has 3823 samples, **optdigits.tes** has 1797 samples.
Case **optdigits.tra**: 3823 samples correct, hit rate = 100.00%.
Case **optdigits.tes**: 1538 samples correct, hit rate = 85.56%.
The 100% hit rate of **optdigits.tra** case can't judge anything because I say a node is leaf node if it is pure and don't put limit on the depth of the tree. The only thing that can tell from this 100% is this program doesn't

have logical error. After doing the check, the maximum depth of this tree is 15.

The thing we need to concern is **optdigits.tes** case. To understand the accuracy of my C++ program, I also use DecisionTreeClassifier (python-sklearn) function with max depth 15 to test.

Case **optdigits.tes**: 1532 samples correct, hit rate = 85.25%. (PYTHON)

The accuracy of my C++ program is almost equal to the accuracy of using PYTHON library sklearn. (PYTHON code **optdigits.py** appends after **main.cpp** in **Appendix** section)



Random Forest:

Each tree has sample_rate * dataset_size samples.

10 trees, each tree has sample_rate **= *0.5***:

Case **optdigits.tra**:

1st test: 3804 samples correct, hit rate = 99.50%

2nd test: 3810 samples correct, hit rate = 99.66%

3rd test: 3805 samples correct, hit rate = 99.53%

Case **optdigits.tes**:

1st test: 1642 samples correct, hit rate = 91.37%

2nd test: 1645 samples correct, hit rate = 91.54%

3rd test: 1644 samples correct, hit rate = 91.49%

We can see the result is greater than the result in **Decision Tree**, in fact, **Decision Tree** is the case of 1 tree only and sample_rate = 1.0. The reason why **optdigits.tra** case's performance reduces is because the validation set isn't same as training set in **Random Forest**.

15 trees, each tree has sample_rate **= *0.5***:
Case **optdigits.tra**: 3815 samples correct, hit rate = 99.79%
Case **optdigits.tes**: 1644 samples correct, hit rate = 91.49%
20 trees, each tree has sample_rate **= *0.5***:
Case **optdigits.tra**: 3817 samples correct, hit rate = 99.84%
Case **optdigits.tes**: 1668 samples correct, hit rate = 92.82%
10 trees, each tree has sample_rate **= *0.7***:
Case **optdigits.tra**: 3820 samples correct, hit rate = 99.92%
Case **optdigits.tes**: 1637 samples correct, hit rate = 91.10%
15 trees, each tree has sample_rate **= *0.7***:
Case **optdigits.tra**: 3822 samples correct, hit rate = 99.97%
Case **optdigits.tes**: 1642 samples correct, hit rate = 91.37%
20 trees, each tree has sample_rate **= *0.7***:
Case **optdigits.tra**: 3822 samples correct, hit rate = 99.97%
Case **optdigits.tes**: 1631 samples correct, hit rate = 90.76%

Generally, the **Random Forest** perform better than **Decision Tree**, however in **Random Forest**, it seems that the number of tree and sample_rate don't have positive correlation to the hit rate. I think the reason why is that the rising of number of tree and sample_rate may cause the probability of extreme samples being chosen, which can reduce the hit rate.

Experiment of optdigits-orig dataset:

I do some little change on **main.cpp** to fit the type of **optdigits-orig** dataset and use **optdigits-orig.tra** as training dataset and **optdigits-orig.cv** as validation subset. Below is the result,

Decision Tree:
optdigits-orig.tra has 1934 samples, **optdigits-orig.cv** has 946 samples.
Case **optdigits-orig.tra**: 1934 samples correct, hit rate = 100.00%.
Case **optdigits-orig.cv**: 814 samples correct, hit rate = 86.05%.

Random Forest:
Each tree has sample_rate * dataset_size samples.
10 trees, each tree has sample_rate **= *0.5***:

Case **optdigits-orig.tra**: 1921 samples correct, hit rate = 99.32%.
Case **optdigits-orig.cv**: 893 samples correct, hit rate = 94.40%.
10 trees, each tree has sample_rate **= *0.7***:
Case **optdigits-orig.tra**: 1933 samples correct, hit rate = 99.95%.
Case **optdigits-orig.cv**: 889 samples correct, hit rate = 93.97%.

Appendix:

main.cpp:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <vector>
#include <algorithm>
#include <queue>
#include <fstream>

using namespace std;
struct Sample{
    int pic[64];
    int value;
};
class Node{
public:
    Node *branch_t, *branch_f;
    vector<Sample> dataset;
    float impurity, threshold;
    int index;
    int attr, leaf; // index of pic, if impurity = 0.0 leaf != -1
    Node();
    Node(vector<Sample> & DataSet);
};
struct SampleSorter{
    int attribute;
    SampleSorter(int ATTR) : attribute(ATTR) {}
    bool operator() (Sample s1, Sample s2) { return (s1.pic[attribute] <
s2.pic[attribute]);}
};
struct Children{
```

```
    float impurity_t, impurity_f; //
branch_true_gini_impurity,branch_false_gini_impurity
    vector<Sample> sample_t, sample_f;
    int attribute;
    float threshold;
};

void ReadDataset(const char* filename, vector<Sample> & dest);
void SampleBagging(vector<vector<Sample> > & dest, vector<Sample>
& train_set, float sample_rate);
int Predict(Node* node, int picture[64]);
void PrintTree(Node* node);
void BuildDecisionTree(Node* node);
Children Partition(vector<Sample> & dataset, int & attribute, float &
threshold);
float GiniImpurity(int digit_count[10], int node_size);
float InfoGain(float & parent_impurity, Children & children);

int main(int argc, char const *argv[]){
    int ToA = 1;
    printf("DecisionTreeClassifier(1),RandomForestClassifier(2): ");
    scanf("%d",&ToA);
    if(2 < ToA || ToA < 1){
        printf("[x] Unexpected input!!!\n");
        return -1;
     }
    vector<Sample> digits_train,digits_test;
    ReadDataset("optdigits.tra",digits_train);

    int n_tree = 1;
    float sample_rate = 1.0;
    if(ToA == 1)
        printf("DecisionTreeClassifier selected\n");
    else{
        printf("RandomForestClassifier selected\nInput number of tree:
");
        scanf("%d",&n_tree);
        printf("Input sample_rate(subset_size = train_set_size *
```

```
sample_rate): ");
            scanf("%f",&sample_rate);
        }
        // Sample bagging
        vector<vector<Sample> > sets(n_tree);
        SampleBagging(sets, digits_train, sample_rate);

        // Build tree
        printf("Progress of Build Tree starts, it takes some time, don't kill the
program\n");
        Node *DT[n_tree];
        for(int ni = 0;ni < n_tree;ni++){
            DT[ni] = new Node(sets[ni]);
            BuildDecisionTree(DT[ni]);
        }

        // Predict result by the tree
        ReadDataset("optdigits.tes",digits_test);
        int hit = 0;
        for(int i = 0;i < digits_test.size();i++){
            int vote[10] = {0};
            for(int ni = 0;ni < n_tree;ni++)
                vote[Predict(DT[ni], digits_test[i].pic)]++;
            int ans = digits_test[i].value, res = 0, highest_vote = 0;
            for(int vi = 0;vi < 10;vi++)
                if(vote[vi] > highest_vote){
                    highest_vote = vote[vi];
                    res = vi;
                }
            if(res == ans)
                hit++;
            printf("ans = %d, res = %d\n",ans,res);
        }
        printf("\nHit rate = %d / %d =
%f\n",hit,digits_test.size(),(float)hit/digits_test.size());

        printf("Exit(0),PrintTree(1): ");
        scanf("%d",&ToA);
```

```
    for(int ni = 0;ni < n_tree && ToA;ni++){
        printf("Tree %d:\n",ni);
        PrintTree(DT[ni]); // Print the tree by DFS
    }
    return 0;
}

void SampleBagging(vector<vector<Sample> > & dest, vector<Sample>
& train_set, float sample_rate){
    int subset_size = train_set.size() * sample_rate;
    srand(time(NULL));
    for(int ni = 0;ni < dest.size();ni++){
        if(subset_size == train_set.size()){
            dest[ni] = train_set;
            continue;
        }
        vector<bool> selected_sample(train_set.size());
        int hit = 0;
        while(hit < subset_size){
            int index = rand() % train_set.size();
            if(!selected_sample[index]){
                selected_sample[index] = true;
                hit++;
            }
        }
        for(int si = 0;si < selected_sample.size();si++)
            if(selected_sample[si])
                dest[ni].push_back(train_set[si]);
    }
}
void BuildDecisionTree(Node* node){
    if(node->impurity == 0.0){
        node->leaf = node->dataset[0].value;
        return; // pure node: leaf node
    }
    // find optimal partition
    float optimal_infogain = 0.0;
    Children optimal_children;
```

```
    for(int attribute = 0;attribute < 64;attribute++){
        sort(node->dataset.begin(), node->dataset.end(),
SampleSorter(attribute));
        // calculate possible threshold
        vector<float> thres_arr;
        for(int di = 0;di < node->dataset.size() - 1;di++){
            float threshold = (float)(node->dataset[di].pic[attribute] +
node->dataset[di + 1].pic[attribute]) / 2.0;
            if(thres_arr.empty() || threshold != thres_arr.back())
                thres_arr.push_back(threshold);
        }

        for(int ti = 0;ti < thres_arr.size();ti++){
            Children children = Partition(node->dataset, attribute,
thres_arr[ti]);
            if(children.sample_t.empty() || children.sample_f.empty())
                continue;
            //printf("attr = %d, threshold = %f\n%d %d\n", attribute,
thres_arr[ti], children.sample_t.size(), children.sample_f.size());
            //printf("%f %f\n", children.impurity_t, children.impurity_f);
            float infogain = InfoGain(node->impurity, children);
            //printf("infogain = %f\n",infogain);
            if(infogain > optimal_infogain){
                optimal_infogain = infogain;
                optimal_children = children;
            }
        }
    }
    //printf("\n\nOptimal Partition:\n");
    //printf("attr = %d, threshold = %f\n", optimal_children.attribute,
optimal_children.threshold);
    //printf("infogain = %f\n",optimal_infogain);

    // Construct child nodes
    Node *child_t = new Node(), *child_f = new Node();
    child_t->dataset = optimal_children.sample_t;
    child_f->dataset = optimal_children.sample_f;
    child_t->impurity = optimal_children.impurity_t;
```

```
    child_f->impurity = optimal_children.impurity_f;
    child_t->index = node->index << 1;
    child_f->index = (node->index << 1) + 1;
    // Update present node info
    node->attr = optimal_children.attribute;
    node->threshold = optimal_children.threshold;
    node->branch_t = child_t;
    node->branch_f = child_f;
    // Recursively call the function BuildDecisionTree()
    BuildDecisionTree(node->branch_t);
    BuildDecisionTree(node->branch_f);
}
Children Partition(vector<Sample> & dataset, int & attribute, float &
threshold){
    Children children;
    children.attribute = attribute;
    children.threshold = threshold;
    int count_t[10] = {0}, count_f[10] = {0};
    for(int di = 0;di < dataset.size();di++){
        if(dataset[di].pic[attribute] <= threshold){
            children.sample_t.push_back(dataset[di]);
            count_t[dataset[di].value]++;
        }
        else{
            children.sample_f.push_back(dataset[di]);
            count_f[dataset[di].value]++;
        }
    }
    children.impurity_t = GiniImpurity(count_t,children.sample_t.size());
    children.impurity_f =
GiniImpurity(count_f,children.sample_f.size());
    return children;
}
float GiniImpurity(int digit_count[10], int node_size){
    float impurity = 1.0;
    for(int i = 0;i < 10;i++){
        float prob = (float)digit_count[i] / node_size;
        impurity -= (prob * prob);
```

```
    }
    return impurity;
}
float InfoGain(float & parent_impurity, Children & children){
    float infogain = parent_impurity, impurity_t = children.impurity_t,
impurity_f = children.impurity_f;
    int sample_t_size = children.sample_t.size(), sample_f_size =
children.sample_f.size();
    int parent_size = sample_t_size + sample_f_size;
    infogain -= (((float)sample_t_size * impurity_t +
(float)sample_f_size * impurity_f) / parent_size);
    return infogain;
}
int Predict(Node* node, int picture[64]){
    if(node->leaf != -1)
        return node->leaf;
    if(picture[node->attr] <= node->threshold)
        return Predict(node->branch_t, picture);
    return Predict(node->branch_f, picture);
}
void PrintTree(Node* node){
    queue<Node*> q;
    q.push(node);
    while(!q.empty()){
        Node *n = q.front();
        q.pop();
        printf("index, attr, thres, val = %d, (%d,%d), %f, %d\n",n-
>index,n->attr / 8,n->attr % 8,n->threshold,n->leaf);
        if(n->branch_t)
            q.push(n->branch_t);
        if(n->branch_f)
            q.push(n->branch_f);
    }
}
void ReadDataset(const char* filename, vector<Sample> & dest){
    ifstream file(filename);
    Sample tmp;
    if(file.is_open()){
```

```
            string line;
            while(std::getline(file, line)){
                line += ',';
                int i = 0,value = 0;
                for(int li = 0;li < line.length();li++){
                    if(line[li] == ','){
                        if(i < 64){
                            tmp.pic[i] = value;
                            i++;
                        }
                        else{
                            tmp.value = value;
                            i = 0;
                            dest.push_back(tmp);
                        }
                        value = 0;
                    }
                    else if('0' <= line[li] && line[li] <= '9')
                        value = value * 10 + (line[li] - '0');
                }
            }
            file.close();
        }
        else{
            printf("E: File open failed\n");
            exit(1);
        }
}
Node::Node(){
    impurity = 1.0;
    threshold = 0.0;
    attr = -1;
    leaf = -1;
    branch_t = branch_f = NULL;
}
Node::Node(vector<Sample> & DataSet){
    threshold = -1.0;
    attr = -1;
```

```
    leaf = -1;
    index = 1; // root
    dataset = DataSet;
    int count[10] = {0};
    for(int di = 0;di < dataset.size();di++)
        count[dataset[di].value]++;
    impurity = GiniImpurity(count, dataset.size());
    branch_t = branch_f = NULL;
}
```

optdigit.py:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
import numpy as np
import pandas as pd
import sys

def main():
    digits_train = pd.read_csv('optdigits\\optdigits.tra', header = None)
    digits_test = pd.read_csv('optdigits\\optdigits.tes', header = None)
    train_x,train_y = digits_train[np.arange(64)],digits_train[64] # digit
picture,digit
    test_x,test_y = digits_test[np.arange(64)],digits_test[64] # digit
picture,digit

    tree = DecisionTreeClassifier(criterion = 'gini', max_depth = 10,
random_state = 0)
    tree.fit(train_x,train_y)
    res_y = tree.predict(test_x)
    """
    forest = RandomForestClassifier(criterion = 'gini', n_estimators = 10,
random_state = 3)
    forest.fit(train_x,train_y)
    res_y = forest.predict(test_x)
    """
    count = 0
    for i in range(0,len(res_y)):
        print('res = ',res_y[i])
```

```
            print('ans = ',test_y[i])
            if res_y[i] == test_y[i]:
                count += 1
                print('Correct')
            else:
                print('Wrong')
            print('----------------')
        print('{0:d} / {1:d} = {2:.2f}'.format(count,len(res_y),count /
len(res_y)))
if __name__ == '__main__':
    main()
```