AI Project 2 Report

0516076 施威綸

-

Date: May 5, 2019

Coding Language: C/C++

Environment: Ubuntu 18.04

In this project, I experimented on classification tasks with supervised learning using random forests.

Dataset

I used the dataset “iris” downloaded from UCI Machine Learning Repository.

And the data is in this form:

attr1, attr2, attr3, attr4, label

The data has only 4 attributes and an output label.

Module

The implementations are listed below:

- CART
- Random forest
- Tree/attribute bagging
- Validation
- Speed test

- Adjustable hyper-parameters

Tests

Output fields definitions

t/v: the rate of # of training data / validation data

bag: the rate of data bagging

gini: the purity standard to decide when to stop splitting a node

for: the # of trees in the forest

obb: out-of-bag accuracy

val: validation accuracy

time: execution time

Attribute bagging

There are only 4 attributes in the data, so I had tested on module with (left)/without (right) attribute bagging. I selected 2 of 4 attributes ($p^{1/2}$).

t/v	bag	gini	for	oob	val	time(s)
0.60/0.40	0.10	0.00	1	0.679012	0.683333	0.005082
0.60/0.40	0.10	0.00	2	0.697531	0.666667	0.007401
0.60/0.40	0.10	0.00	3	0.703704	0.666667	0.009179
0.60/0.40	0.10	0.00	4	0.700617	0.666667	0.011001
0.60/0.40	0.10	0.00	5	0.841975	0.800000	0.017564
0.60/0.40	0.10	0.00	6	0.707819	0.666667	0.016145
0.60/0.40	0.10	0.00	7	0.841270	0.800000	0.018899
0.60/0.40	0.10	0.00	8	0.820988	0.750000	0.021286
0.60/0.40	0.10	0.00	9	0.805213	0.783333	0.024375
0.60/0.40	0.10	0.00	10	0.853086	0.800000	0.026906
0.60/0.40	0.10	0.00	20	0.921605	0.883333	0.083480
0.60/0.40	0.10	0.00	30	0.944856	0.916667	0.125579
0.60/0.40	0.10	0.00	40	0.945679	0.933333	0.184840
0.60/0.40	0.10	0.00	50	0.957284	0.933333	0.292607
0.60/0.40	0.10	0.00	60	0.955144	0.950000	0.355808
0.60/0.40	0.10	0.00	70	0.954674	0.933333	0.484597
0.60/0.40	0.10	0.00	80	0.954784	0.933333	0.659529
0.60/0.40	0.10	0.00	90	0.933059	0.933333	0.736749
0.60/0.40	0.10	0.00	100	0.966049	0.966667	0.949490
0.60/0.40	0.10	0.00	200	0.921852	0.983333	3.801565
0.60/0.40	0.10	0.00	300	0.955062	0.950000	8.191294
0.60/0.40	0.10	0.00	400	0.955679	0.933333	14.554465
0.60/0.40	0.10	0.00	500	0.966815	0.900000	22.537663

t/v	bag	gini	for	oob	val	time(s)
0.60/0.40	0.10	0.00	1	0.950617	0.966667	0.004773
0.60/0.40	0.10	0.00	2	0.895062	0.983333	0.006703
0.60/0.40	0.10	0.00	3	0.901235	0.983333	0.009576
0.60/0.40	0.10	0.00	4	0.867284	0.983333	0.012444
0.60/0.40	0.10	0.00	5	0.856790	0.983333	0.014257
0.60/0.40	0.10	0.00	6	0.860082	0.966667	0.016058
0.60/0.40	0.10	0.00	7	0.929453	0.983333	0.020328
0.60/0.40	0.10	0.00	8	0.921296	0.950000	0.035476
0.60/0.40	0.10	0.00	9	0.931413	0.983333	0.027670
0.60/0.40	0.10	0.00	10	0.885185	0.950000	0.030326
0.60/0.40	0.10	0.00	20	0.922222	0.983333	0.079517
0.60/0.40	0.10	0.00	30	0.899177	0.800000	0.139675
0.60/0.40	0.10	0.00	40	0.909877	0.850000	0.201047
0.60/0.40	0.10	0.00	50	0.921235	0.866667	0.282016
0.60/0.40	0.10	0.00	60	0.933539	0.866667	0.372081
0.60/0.40	0.10	0.00	70	0.934039	0.916667	0.536660
0.60/0.40	0.10	0.00	80	0.912809	0.900000	0.647387
0.60/0.40	0.10	0.00	90	0.977641	0.916667	0.886359
0.60/0.40	0.10	0.00	100	0.898642	0.866667	1.028668
0.60/0.40	0.10	0.00	200	0.965741	0.900000	4.583685
0.60/0.40	0.10	0.00	300	0.921728	0.900000	8.595337
0.60/0.40	0.10	0.00	400	0.922006	0.883333	15.736273
0.60/0.40	0.10	0.00	500	0.977556	0.933333	23.178718

The result showed that the test with bagging still did well on the accuracy (same as the test without bagging) and also better on time, so we decide to use attribute bagging for the later tests. In other cases, the accuracy may become lower after using attribute bagging. But the # of attributes is too small here, we cannot see some significant differences.

t/v	bag	gini	for	oob	val	time(s)
0.60/0.40	0.10	0.00	1	0.444444	0.516667	0.004554
0.60/0.40	0.10	0.00	2	0.685185	0.633333	0.006391
0.60/0.40	0.10	0.00	3	0.609053	0.600000	0.008591
0.60/0.40	0.10	0.00	4	0.759259	0.750000	0.008706
0.60/0.40	0.10	0.00	5	0.802469	0.783333	0.009645
0.60/0.40	0.10	0.00	6	0.761317	0.750000	0.011025
0.60/0.40	0.10	0.00	7	0.611993	0.600000	0.012318
0.60/0.40	0.10	0.00	8	0.760802	0.766667	0.015517
0.60/0.40	0.10	0.00	9	0.887517	0.933333	0.017254
0.60/0.40	0.10	0.00	10	0.776543	0.850000	0.018660
0.60/0.40	0.10	0.00	20	0.855556	0.783333	0.048306
0.60/0.40	0.10	0.00	30	0.892181	0.850000	0.079999
0.60/0.40	0.10	0.00	40	0.954321	0.966667	0.132403
0.60/0.40	0.10	0.00	50	0.954321	0.950000	0.193583
0.60/0.40	0.10	0.00	60	0.944650	0.950000	0.268431
0.60/0.40	0.10	0.00	70	0.944621	0.950000	0.355300
0.60/0.40	0.10	0.00	80	0.832253	0.716667	0.450355
0.60/0.40	0.10	0.00	90	0.866392	0.816667	0.565511
0.60/0.40	0.10	0.00	100	0.923086	0.916667	0.700691
0.60/0.40	0.10	0.00	200	0.844136	0.816667	2.726317
0.60/0.40	0.10	0.00	300	0.944609	0.950000	6.202763
0.60/0.40	0.10	0.00	400	0.766420	0.583333	10.809678
0.60/0.40	0.10	0.00	500	0.932889	0.883333	17.576813

This test has a bagging rate lower than $p^{1/2}$ (1 of 4) attributes. We can see that the accuracy become unstable then.

Data bagging

I did the same test twice.

t/v	bag	gini	for	oob	val	time(s)
0.60/0.40	0.05	0.00	100	0.933837	0.933333	0.839018
0.60/0.40	0.10	0.00	100	0.966790	0.916667	0.906285
0.60/0.40	0.15	0.00	100	0.943247	0.950000	0.897053
0.60/0.40	0.20	0.00	100	0.956667	0.916667	0.900006
0.60/0.40	0.25	0.00	100	0.947353	0.950000	0.968867
0.60/0.40	0.30	0.00	100	0.885238	0.816667	0.949134
0.60/0.40	0.35	0.00	100	0.944237	0.916667	1.010281
0.60/0.40	0.40	0.00	100	0.911852	0.816667	1.040009
0.60/0.40	0.45	0.00	100	0.965200	0.933333	1.092671
0.60/0.40	0.50	0.00	100	0.888667	0.883333	1.199475
0.60/0.40	0.55	0.00	100	0.943171	0.766667	1.243429
0.60/0.40	0.60	0.00	100	0.942222	0.850000	1.376166
0.60/0.40	0.65	0.00	100	0.827812	0.833333	1.429694
0.60/0.40	0.70	0.00	100	0.881111	0.833333	1.655345
0.60/0.40	0.75	0.00	100	0.900435	0.783333	1.721371
0.60/0.40	0.80	0.00	100	0.897778	0.783333	1.890274
0.60/0.40	0.85	0.00	100	0.974286	0.916667	2.056393
0.60/0.40	0.90	0.00	100	0.952222	0.933333	2.248108
0.60/0.40	0.95	0.00	100	0.812000	0.783333	2.578161

t/v	bag	gini	for	oob	val	time(s)
0.60/0.40	0.05	0.00	100	0.977558	0.933333	0.856620
0.60/0.40	0.10	0.00	100	0.942593	0.933333	0.887317
0.60/0.40	0.15	0.00	100	0.956494	0.950000	0.893940
0.60/0.40	0.20	0.00	100	0.925833	0.916667	0.929990
0.60/0.40	0.25	0.00	100	0.877794	0.900000	0.926889
0.60/0.40	0.30	0.00	100	0.896825	0.883333	0.936911
0.60/0.40	0.35	0.00	100	0.888475	0.916667	0.977803
0.60/0.40	0.40	0.00	100	0.858333	0.783333	1.051643
0.60/0.40	0.45	0.00	100	0.922000	0.933333	1.094352
0.60/0.40	0.50	0.00	100	0.930444	0.883333	1.214360
0.60/0.40	0.55	0.00	100	0.904634	0.766667	1.306059
0.60/0.40	0.60	0.00	100	0.920000	0.933333	1.421614
0.60/0.40	0.65	0.00	100	0.865625	0.800000	1.507613
0.60/0.40	0.70	0.00	100	0.853704	0.766667	1.560522
0.60/0.40	0.75	0.00	100	0.976957	0.800000	1.741657
0.60/0.40	0.80	0.00	100	0.831111	0.816667	1.922058
0.60/0.40	0.85	0.00	100	0.920714	0.750000	2.024047
0.60/0.40	0.90	0.00	100	0.804444	0.866667	2.275787
0.60/0.40	0.95	0.00	100	0.792000	0.900000	2.557629

We can look at the relationship between OOB accuracy and the bagging rate, the more out-of-bag data we tested, the more stable the OOB accuracy was.

Training / validation data rate

The out-of-bag error may alter with the size of training subset, so we focus on the validation accuracy on this part. (also, the same test twice)

t/v	bag	gini	for	oob	val	time(s)
0.05/0.95	0.10	0.00	100	0.428571	0.328671	0.045951
0.10/0.90	0.10	0.00	100	0.464286	0.318519	0.081640
0.15/0.85	0.10	0.00	100	0.910500	0.703125	0.168095
0.20/0.80	0.10	0.00	100	0.935556	0.800000	0.254967
0.25/0.75	0.10	0.00	100	0.947059	0.796460	0.327535
0.30/0.70	0.10	0.00	100	0.932683	0.895238	0.419869
0.35/0.65	0.10	0.00	100	0.922979	0.897959	0.495016
0.40/0.60	0.10	0.00	100	0.917778	0.944444	0.592154
0.45/0.55	0.10	0.00	100	0.969836	0.951807	0.642814
0.50/0.50	0.10	0.00	100	0.959265	0.933333	0.749809
0.55/0.45	0.10	0.00	100	0.940000	0.955882	0.794721
0.60/0.40	0.10	0.00	100	0.967284	0.933333	0.883877
0.65/0.35	0.10	0.00	100	0.968864	0.943396	0.986591
0.70/0.30	0.10	0.00	100	0.961579	0.933333	1.070863
0.75/0.25	0.10	0.00	100	0.910495	0.947368	1.122562
0.80/0.20	0.10	0.00	100	0.925093	0.866667	1.205529
0.85/0.15	0.10	0.00	100	0.953565	0.956522	1.296120
0.90/0.10	0.10	0.00	100	0.933770	0.933333	1.376216
0.95/0.05	0.10	0.00	100	0.942734	0.875000	1.458820

t/v	bag	gini	for	oob	val	time(s)
0.05/0.95	0.10	0.00	100	0.285714	0.335664	0.048495
0.10/0.90	0.10	0.00	100	0.396429	0.325926	0.076176
0.15/0.85	0.10	0.00	100	0.768500	0.632812	0.175559
0.20/0.80	0.10	0.00	100	0.931481	0.800000	0.253135
0.25/0.75	0.10	0.00	100	0.888824	0.778761	0.315957
0.30/0.70	0.10	0.00	100	0.932683	0.895238	0.414431
0.35/0.65	0.10	0.00	100	0.961277	0.928571	0.503380
0.40/0.60	0.10	0.00	100	0.949630	0.966667	0.597981
0.45/0.55	0.10	0.00	100	0.953443	0.963855	0.645823
0.50/0.50	0.10	0.00	100	0.919853	0.906667	0.743196
0.55/0.45	0.10	0.00	100	0.951216	0.911765	0.813255
0.60/0.40	0.10	0.00	100	0.966914	0.933333	0.911346
0.65/0.35	0.10	0.00	100	0.957955	0.943396	0.975493
0.70/0.30	0.10	0.00	100	0.933263	0.977778	1.046982
0.75/0.25	0.10	0.00	100	0.973168	0.921053	1.126483
0.80/0.20	0.10	0.00	100	0.933426	1.000000	1.237776
0.85/0.15	0.10	0.00	100	0.960348	0.956522	1.295592
0.90/0.10	0.10	0.00	100	0.948115	0.933333	1.379643
0.95/0.05	0.10	0.00	100	0.951172	0.875000	1.444442

In this case, the training module is relatively simple. We do not really need that much data for training. The module performed well on only 20% of training rate.

Forest size (number of trees)

(same test twice)

t/v	bag	gini	for	oob	val	time(s)
0.60/0.40	0.20	0.00	1	0.513889	0.550000	0.004732
0.60/0.40	0.20	0.00	2	0.687500	0.700000	0.007591
0.60/0.40	0.20	0.00	3	0.791667	0.816667	0.011660
0.60/0.40	0.20	0.00	4	0.836806	0.850000	0.010843
0.60/0.40	0.20	0.00	5	0.905556	0.933333	0.015620
0.60/0.40	0.20	0.00	6	0.879630	0.883333	0.019393
0.60/0.40	0.20	0.00	7	0.886905	0.916667	0.022244
0.60/0.40	0.20	0.00	8	0.887153	0.916667	0.023743
0.60/0.40	0.20	0.00	9	0.878086	0.883333	0.027111
0.60/0.40	0.10	0.00	10	0.823457	0.833333	0.021392
0.60/0.40	0.10	0.00	20	0.933333	0.916667	0.057265
0.60/0.40	0.10	0.00	30	0.923868	0.900000	0.101309
0.60/0.40	0.10	0.00	40	0.945679	0.883333	0.165513
0.60/0.40	0.10	0.00	50	0.945432	0.933333	0.249937
0.60/0.40	0.10	0.00	60	0.946091	0.950000	0.346592
0.60/0.40	0.10	0.00	70	0.954145	0.950000	0.452887
0.60/0.40	0.10	0.00	80	0.943673	0.950000	0.578924
0.60/0.40	0.10	0.00	90	0.954595	0.916667	0.769581
0.60/0.40	0.10	0.00	100	0.966667	0.950000	0.914834
0.60/0.40	0.10	0.00	200	0.966296	0.916667	3.572748
0.60/0.40	0.10	0.00	300	0.945309	0.950000	7.778184
0.60/0.40	0.10	0.00	400	0.921975	0.966667	13.656331
0.60/0.40	0.10	0.00	500	0.955407	0.966667	21.956117

t/v	bag	gini	for	oob	val	time(s)
0.60/0.40	0.20	0.00	1	0.694444	0.483333	0.004744
0.60/0.40	0.20	0.00	2	0.784722	0.566667	0.016759
0.60/0.40	0.20	0.00	3	0.824074	0.650000	0.009909
0.60/0.40	0.20	0.00	4	0.795139	0.600000	0.014320
0.60/0.40	0.20	0.00	5	0.800000	0.616667	0.019854
0.60/0.40	0.20	0.00	6	0.796296	0.616667	0.017067
0.60/0.40	0.20	0.00	7	0.799603	0.616667	0.019111
0.60/0.40	0.20	0.00	8	0.796875	0.616667	0.023441
0.60/0.40	0.20	0.00	9	0.808642	0.666667	0.026878
0.60/0.40	0.10	0.00	10	0.911111	0.866667	0.019944
0.60/0.40	0.10	0.00	20	0.943210	0.883333	0.056987
0.60/0.40	0.10	0.00	30	0.933333	0.916667	0.102079
0.60/0.40	0.10	0.00	40	0.942901	0.966667	0.170890
0.60/0.40	0.10	0.00	50	0.942716	0.950000	0.260828
0.60/0.40	0.10	0.00	60	0.932716	0.966667	0.359962
0.60/0.40	0.10	0.00	70	0.922399	0.966667	0.481228
0.60/0.40	0.10	0.00	80	0.932407	0.966667	0.590733
0.60/0.40	0.10	0.00	90	0.922359	0.916667	0.743650
0.60/0.40	0.10	0.00	100	0.922593	0.916667	0.898875
0.60/0.40	0.10	0.00	200	0.944815	0.900000	3.367026
0.60/0.40	0.10	0.00	300	0.945350	0.966667	7.711501
0.60/0.40	0.10	0.00	400	0.933488	0.966667	13.614129
0.60/0.40	0.10	0.00	500	0.944123	0.916667	21.597571

The accuracy had stopped growing after 20~50 trees in the forest. The time got longer if the forest became larger, without improving the accuracy.

Gini index

This test is about when to stop splitting a node. To avoid noise, we can decide to ignore false data when splitting a node. So I set a number to stop dividing dataset when the Gini index is lower than the number.

The number did not affect the result well in this case. Maybe the module is not complex enough to add this feature.

Appendix

random_forest.h:

```
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <vector>
#include <algorithm>

#define LABELSIZE 32
#define BUFSIZE 128
#define DATAMAX 200

struct Data {
	std::vector<float> attr;
	char label[LABELSIZE];
};

struct Node {
	std::vector<Data> dataset;
	int attribute;
	double threshold;
	Node* left;
	Node* right;
	Node* parent;
	int isleaf;
	char label[LABELSIZE];
	Node();
};

class Tree {
public:
	Node *root;
	std::vector<int> used_attr;
public:
	Tree();
	void bagging();
	double giniIndex(std::vector<Data>);
	double impurity(Node*);
	void split(Node*, int, double);
	double selectThreshold(Node*, int);
	void selectAttribute(Node*);
	void buildTree(Node*);
	int isPure(std::vector<Data>);
	int remainAttr();
	void selectLabel(Node*);
	int checkLeaf(Node*);


	void printDataset(); // debug
	void printDataset(Node*);
	void printDataset(std::vector<Data>);
};

int genRandom(int);
void timeStart();
void readData(const char*);
void divideDataset();
void buildForest();
char* traverse(Node*, Data);
char* ensemble(Data);
double correctRate(std::vector<Data>);
void printResult();
```

random_forest.cpp

```
#include "random_forest.h"

using namespace std;

clock_t tstart;

float training_ratio;
float bagging_ratio;
float pure_standard;
int forest_size;
```

```
double correct_rate;

vector<Data> d; // the whole data set
vector<Data> training_sset;
vector<Data> validation_sset;
vector<Data> OOB_sset;

vector<Tree> forest;


Node::Node() {
	attribute = 0;
	threshold = 0;
	left = NULL;
	right = NULL;
	parent = NULL;
	isleaf = 0;
	memset(label, 0, sizeof(label));
}

// build during creating a tree object
Tree::Tree() {
	root = new Node;
	bagging();
	buildTree(root);
}

// select random data into root dataset
// push remain data into out-of-bag subset
// select 2(root of 4) random attribute to be used(can't be used later)
void Tree::bagging() {
	// data bagging
	int bagsize = training_sset.size() * bagging_ratio;
	random_shuffle(training_sset.begin(), training_sset.end(), genRandom);
	int i=0;
	while(i < bagsize) {
		root->dataset.push_back(training_sset[i]);
		i++;
	}
	while(i < training_sset.size()) {
		OOB_sset.push_back(training_sset[i]);
		i++;
	}
	// attribute bagging
	used_attr.push_back(0);
	used_attr.push_back(0);
	used_attr.push_back(1);
	used_attr.push_back(1);
	random_shuffle(used_attr.begin(), used_attr.end(), genRandom);

	return;
}

// calculate gini index of a set of data
double Tree::giniIndex(vector<Data> v) {
	if(v.size() == 0) return 0;

	int cnt[3];
	double index = 1;
	memset(cnt, 0, sizeof(cnt));
	for(int i=0; i<v.size(); i++) {
		if(strcmp(v[i].label, "Iris-setosa") == 0) cnt[0]++;
		else if(strcmp(v[i].label, "Iris-virginica") == 0) cnt[1]++;
		else cnt[2]++;
	}
	for(int i=0; i<3; i++) {
		double pk = (double)cnt[i] / v.size();
		index -= pk * pk;
	}
	// printf("%d, %d, %d, %lf\n", cnt[0], cnt[1], cnt[2], index);
	return index;
}

// calculate the total impurity of a node dataset
double Tree::impurity(Node *n) {
	double g1 = giniIndex(n->left->dataset);
	double g2 = giniIndex(n->right->dataset);
	double n1 = (double)n->left->dataset.size() / n->dataset.size();
	double n2 = (double)n->right->dataset.size() / n->dataset.size();

	// printf("\t\t(%f*%f + %f*%f)\n", n1, g1, n2, g2);

	return (n1*g1 + n2*g2);
}

// only split data set into 2 children
// you have to delete children explicitly if unused
void Tree::split(Node *n, int attr_num, double threshold) {
	n->left = new Node;
	n->right = new Node;
	n->left->parent = n;
	n->right->parent = n;

	for(int i=0; i<n->dataset.size(); i++) {
		float val = n->dataset[i].attr[attr_num];
		if(val <= threshold) {
			n->left->dataset.push_back(n->dataset[i]);
		}
		else {
```

```
                                n->right->dataset.push_back(n->dataset[i]);
                        }
                }
                // for(int i=0; i<n->left->dataset.size(); i++) {
                //      printf("%f, %f, %f, %f, %s\n", n->left->dataset[i].attr[0], n->left->dataset[i].attr[1], n->left->dataset[i].attr[2], n->left->dataset[i].attr[3], n->left->dataset[i].label);
                // }
                return;
}

// return the best threshold value of an attribute
double Tree::selectThreshold(Node *n, int attr_num) {
        vector<float> val;
        double min_impurity = 1;
        double best_threshold;

        for(int i=0; i<n->dataset.size(); i++) {
                val.push_back(n->dataset[i].attr[attr_num]);
        }
        sort(val.begin(), val.end());
        for(int i=1; i<val.size(); i++) {
                split(n, attr_num, (val[i]+val[i-1])/2);
                double n_impurity = impurity(n);
                // printf("\t\t-> impurity: %f\n", n_impurity);
                if(min_impurity > n_impurity) {
                        min_impurity = n_impurity;
                        best_threshold = (val[i]+val[i-1])/2;
                }
                delete n->left;
                delete n->right;
        }
        // printf("\tthreshold: %f\n", best_threshold);

        return best_threshold;
}

// select a best attribute with threshold and split node
void Tree::selectAttribute(Node *n) {
        int best_attribute;
        double best_threshold;
        double min_impurity = 1;

        for(int i=0; i<4; i++) {
                if(used_attr[i]) continue;
                double thold = selectThreshold(n, i);
                split(n, i, thold);
                double n_impurity = impurity(n);
                if(min_impurity > n_impurity) {
                        min_impurity = n_impurity;
                        best_threshold = thold;
                        best_attribute = i;
                }
                delete n->left;
                delete n->right;
        }

        split(n, best_attribute, best_threshold);
        n->attribute = best_attribute;
        n->threshold = best_threshold;
        used_attr[best_attribute] = 1;
        // printf("attribute: %d\n", best_attribute);
        return;
}

// based on the gini index of a dataset to check if it is pure
// determined by the pure_standard(default: 0.0)
int Tree::isPure(vector<Data> v) {
        return (giniIndex(v) > pure_standard) ? 0 : 1;
}

int Tree::remainAttr() {
        int cnt = 0;
        for(int i=0; i<used_attr.size(); i++) {
                if(used_attr[i] == 0) {
                        cnt++;
                }
        }
        return cnt;
}

// read through the dataset and set the most label
void Tree::selectLabel(Node* n) {
        int cnt[3];
        memset(cnt, 0, sizeof(cnt));
        for(int i=0; i<n->dataset.size(); i++) {
                if(strcmp(n->dataset[i].label, "Iris-setosa") == 0) {
                        cnt[0]++;
                }
                else if (strcmp(n->dataset[i].label, "Iris-virginica") == 0) {
                        cnt[1]++;
                }
                else cnt[2]++;
        }
        int maj = max(cnt[0], max(cnt[1], cnt[2]));
        if(maj == cnt[0]) {
                strcpy(n->label, "Iris-setosa");
        }
        else if(maj == cnt[1]) {
                strcpy(n->label, "Iris-virginica");
        }
        else {
                strcpy(n->label, "Iris-versicolor");
```

```
        }
        return;
}

// check the necessity to split the node(data purity, remain
unused attributes)
// if not necessary, set the node to leaf and set label
int Tree::checkLeaf(Node *n) {
        if(n->dataset.size() != 0 && !isPure(n->dataset) &&
remainAttr() != 0) return 0;
        n->isleaf = 1;
        selectLabel(n);
        return 1;
}

// recursively split the nodes
void Tree::buildTree(Node *n) {
        if(checkLeaf(n)) return;
        selectAttribute(n);
        buildTree(n->left);
        buildTree(n->right);
}

// for random_shuffle()
int genRandom(int num) { return rand()%num; }

// set timer for calculating excecution time
void timeStart() {
        tstart = clock();
}

void readData(const char* fpath) {
        FILE* fp;
        if((fp = fopen(fpath, "r")) == NULL) {
                perror("file not exists");
                exit(-1);
        }

        char buf[BUFSIZE];

        while(fgets(buf, BUFSIZE, fp) != NULL) {
                float tmp[4];
                Data dtmp;
                sscanf(buf, "%f,%f,%f,%f,%s", &tmp[0],
&tmp[1], &tmp[2], &tmp[3], dtmp.label);
                dtmp.attr.push_back(tmp[0]);
                dtmp.attr.push_back(tmp[1]);
                dtmp.attr.push_back(tmp[2]);
                dtmp.attr.push_back(tmp[3]);
                d.push_back(dtmp);
        }

        fclose(fp);

        // for(int i=0; i<d.size(); i++) {
        //      printf("%.1f, %.1f, %.1f, %.1f, %s\n",
d[i].attr[0], d[i].attr[1], d[i].attr[2], d[i].attr[3], d[i].label);
        //}
}

// divide the original dataset into training subset and
validation subset
void divideDataset() {
        int train_num = d.size() * training_ratio;
        int validate_num = d.size() - train_num;
        random_shuffle(d.begin(), d.end(), genRandom);
        int i = 0;
        while(i < train_num) {
                training_sset.push_back(d[i]);
                i++;
        }
        while(i < d.size()) {
                validation_sset.push_back(d[i]);
                i++;
        }
}

void buildForest() {
        for(int i=0; i<forest_size; i++) {
                Tree t;
                forest.push_back(t);
        }
}

// traverse through the tree and return the classify result
char* traverse(Node *n, Data data) {
        if(n->isleaf) {
                return n->label;
        }
        // else select a way to go
        if((data.attr[n->attribute] <= n->threshold) && n-
>left) {
                return traverse(n->left, data);
        }
        else if(n->right) {
                return traverse(n->right, data);
        }
        fprintf(stderr, "traverse error\n");
        exit(-1);
}

// return the majority vote of the forest
char* ensemble(Data data) {
        int cnt[3];
        char result[LABELSIZE]; // result for each tree
        char* ret;
        memset(cnt, 0, sizeof(cnt));
        for(int i=0; i<forest.size(); i++) {
                strcpy(result, traverse(forest[i].root,
data));
```

```
                        if(strcmp(result, "Iris-setosa") == 0) {
                                cnt[0]++;
                        }
                        else if(strcmp(result, "Iris-virginica") ==
0) {
                                cnt[1]++;
                        }
                        else cnt[2]++;
        }
        int maj = max(cnt[0], max(cnt[1], cnt[2]));
        if(maj == cnt[0]) {
                ret = strdup("Iris-setosa");
        }
        else if(maj == cnt[1]) {
                ret = strdup("Iris-virginica");
        }
        else {
                ret = strdup("Iris-versicolor");
        }

        return ret;
}

// validation and return the correct rate
double correctRate(vector<Data> sset) {
        int data_n = sset.size();
        int cnt_correct = 0;
        for(int i=0; i<data_n; i++) {
                if(strcmp(sset[i].label,
ensemble(sset[i])) == 0) {
                        cnt_correct++;
                }
        }

        return ((double)cnt_correct / data_n);
}

void printResult() {
        printf("%-4.2f/%-8.2f", training_ratio, 1-
training_ratio); // traning/validation data
        printf("%-7.2f", bagging_ratio); // bagging ratio
        printf("%-7.2f", pure_standard); // gini pure
standard
        printf("%-6d", forest_size); // forest size
        printf("%-10f", correctRate(OOB_sset)); // oob
correct rate
        printf("%-10f", correctRate(validation_sset)); //
validation correct rate
        printf("%-5f", (double)(clock()-tstart) /
CLOCKS_PER_SEC); // execution times
        printf("\n");
}

// argv[1] = data file path
// argv[2] = training_ratio
// argv[3] = bagging_ratio
// argv[4] = pure_standard
// argv[5] = forest_size
int main(int argc, char *argv[]) {
        training_ratio = atof(argv[2]);
        bagging_ratio = atof(argv[3]);
        pure_standard = atof(argv[4]);
        forest_size = atof(argv[5]);

        srand(time(NULL));
        readData(argv[1]);
        divideDataset();
        buildForest();
        // Tree t; t.printDataset();
        printResult();
        // t.printDataset();


        return 0;
}



//******************for debugging****************** //
int level = 0;

void Tree::printDataset() {
        printDataset(root);

}

void Tree::printDataset(Node* n) {
        printf("(%d, %f)\n", n->attribute, n->threshold);
        printDataset(n->dataset);

        level++;
        if(n->left != NULL) {
                printDataset(n->left);
        }
        if(n->right != NULL) {
                printDataset(n->right);
        }
        level--;
        return;
}

void Tree::printDataset(vector<Data> v) {
        for(int i=0; i<v.size(); i++) {
                printf("%f, %f, %f, %f, %s\n", v[i].attr[0],
v[i].attr[1], v[i].attr[2], v[i].attr[3], v[i].label);
        }
        printf("-%d\n", level);
}
```