**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

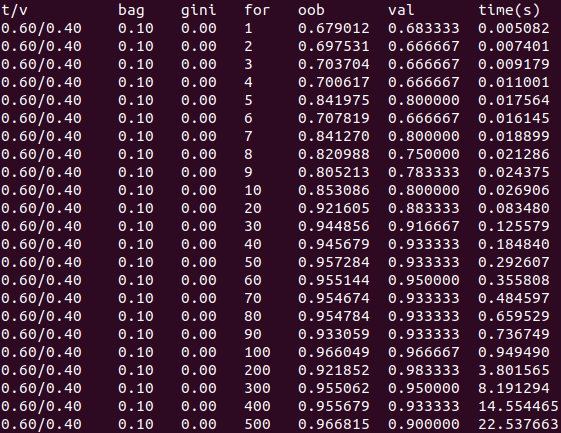
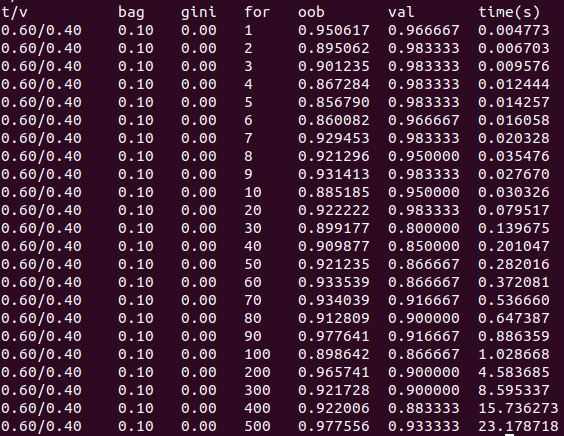
obb: out-of-bag accuracy

val: validation accuracy

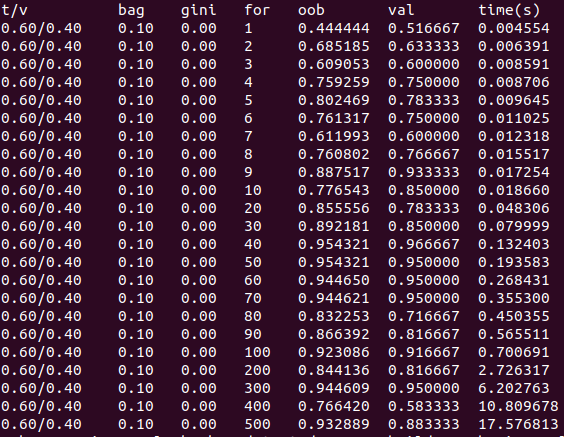
time: execution time

**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 (p1/2).

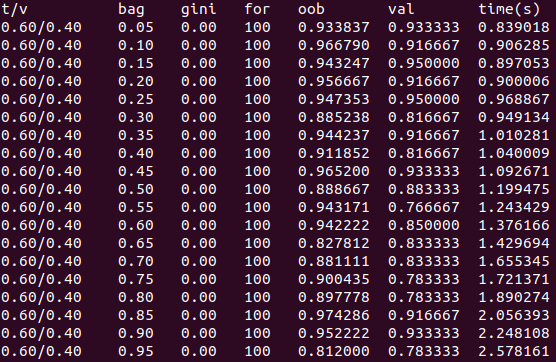
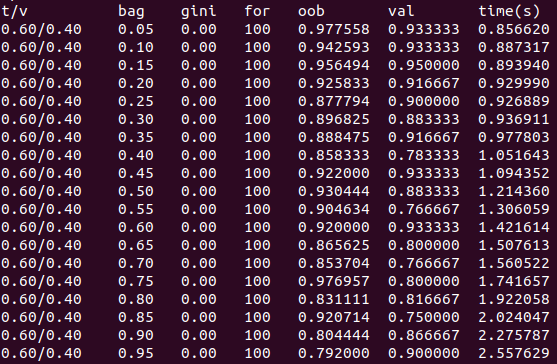
The result showed that the test with bagging still did well on the accuracy (same as the test without bagging) and also better on time, so we decide to use attribute bagging for the later tests. In other cases, the accuracy may become lower after using attribute bagging. But the # of attributes is too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



This test has a bagging rate lower than p1/2 (1 of 4) attributes. We can see that the accuracy become unstable then.

**Data bagging**

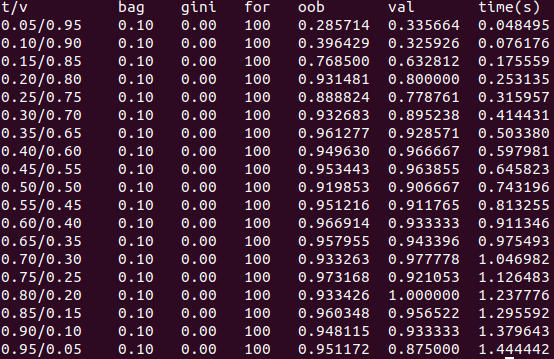
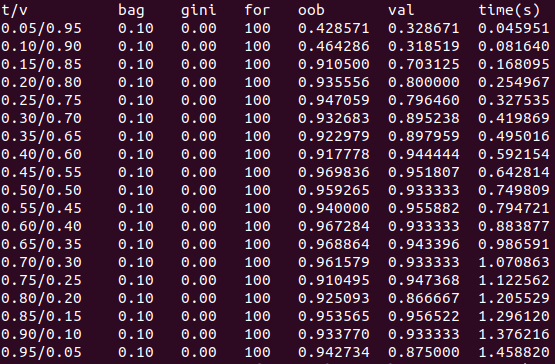
I did the same test twice.

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**

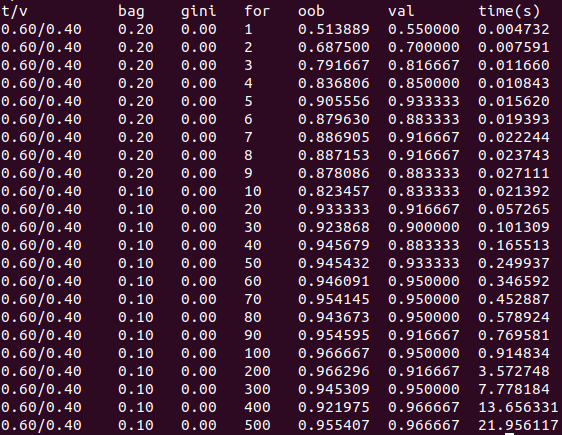
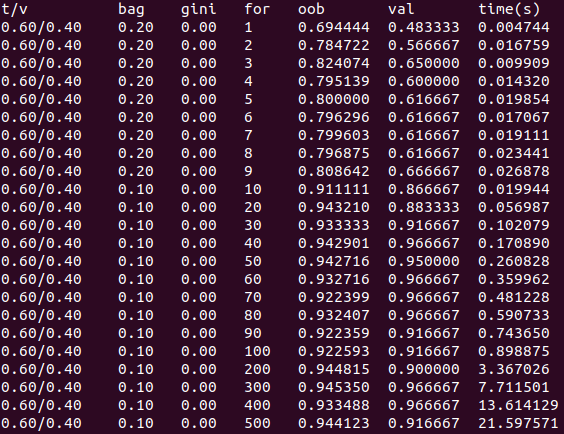
The out-of-bag error may alter with the size of training subset, so we focus on the validation accuracy on this part. (also, the same test twice)



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)

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);

}