Report: Assignment #2

ITE4005, **Data Science.**
2016025305, **Jihun Kim**<jihunkim@hanyang.ac.kr>.

Build a decision tree, and then classify the test set using it.

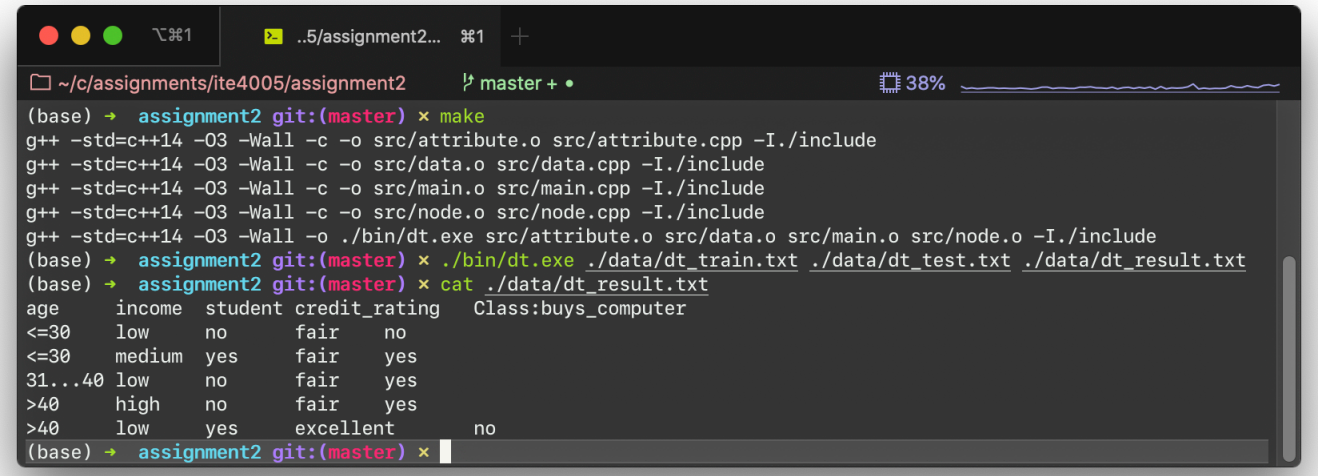
Getting Started

Development Environment

* **OS:** macOS 10.15.4 (x86_64-apple-darwin19.4.0)
* **Language:** C++14
* **Compiler:** Apple clang version 11.0.3 (clang-1103.0.32.59)

Compile & Run

```
$ cd /path/to/repo
$ make
$ ./bin/dt.exe ./data/dt_train.txt ./data/dt_test.txt ./data/dt_result.txt
$ vi ./data/dt_result.txt
```



Implementation

I implemented this assignment using C++. In this report, only important parts are documented.

Attribute Representation

`class attribute<Val>` represnets a single attribute, where **`Val`** is the type the value can be. It inherits **`class attribute_base`** for polymorphism. For simplicity, all values of attribute just reside in the class, so the only way to access attribute values is using **`id`**, whose type is **`attribute_base::val_id`**.

```
class attribute_base {
public:
  typedef size_t val_id;

protected:
  val_id _id;
  std::string _name;

public:
  explicit attribute_base(val_id id) : _id(id) {}
  attribute_base(val_id id, std::string name) : attribute_base(id) { _name = std::move(name); }
  virtual ~attribute_base() = default;

  val_id id() const { return _id; }
  std::string name() const { return _name; }
  void set_name(std::string name) { _name = std::move(name); }

  virtual val_id get_id(void* _Nonnull value) = 0;
  virtual val_id read_value(std::istringstream& iss) = 0;
  virtual void write_value(std::ostream& ofs, val_id id) const = 0;
};
```

```
template<typename Val>
class attribute : public attribute_base {
  std::unordered_map<Val, val_id> _val_to_id;
  std::vector<Val> _id_to_val;

public:
  explicit attribute(val_id id) : attribute_base(id) {}
  attribute(val_id id, std::string name) : attribute_base(id, name) {}
  ~attribute() = default;

  val_id get_id(void* _Nonnull value) override;
  val_id read_value(std::istringstream&) override;
  void write_value(std::ostream& ofs, val_id id) const override;
};
```

Data Representation

Data are consist of attributes, and a label. Both attributes and label are combined with values, except test data doesn't have the value of label. As I mentioned ealier, data doesn't contain the actual value, but only id. This makes data be implemented without template although attributes are not.

```
struct data {
  std::unordered_map<attribute_base*, attribute_base::val_id> attrs;
  std::pair<attribute_base*, attribute_base::val_id> label;
};
```

Node Representation

Nodes are basic building blocks of trees. Each node has a pointer to attribute (**`_attr`**), and id-to-child_node map (**`_children`**), and the id of the value of label(**`label`**). Note that attr is annotated as **`_Nullable`**. If its value is **`nullptr`**, it denotes that this node does not have any children. In that situation, expansion is impossible since there are no attribute to apply.

```
1 struct node {
2   attribute_base* _Nullable _attr{ nullptr };
3   std::unordered_map<attribute_base::val_id, node* _Nonnull> _children;
4   attribute_base::val_id _label{};
5
6   explicit node() = default;
7   ~node();
8   attribute_base::val_id infer(const data& d) const;
9 };
```

Tree Construction

Now we'll start to build a decision tree. The function below constructs tree by selecting attribute to apply, and split all the training data it has, and call itself recursively.
It halts by the following conditions:

(1) **`(line 16)`** Split seems no effect: No more reproduction needed.
(2) **`(line 16)`** No more attribute left: Same as (1).
(3) **`(line 12)`** Label of all data are same: Same as (1).
(4) **`(line 17)`** Gain ratio is too small: Maybe it's because of small amount of outlier.
(5) **`(line 22)`** Split is to small: No sufficient support.

(4), (5) can increase performance by preventing overfitting, however it can rather decrease the performance. Anyway I decided to use it.

```
 1 node* construct_tree(const std::vector<data>& vec_data, std::unordered_set<attribute_base*> attrs) {
 2     node* n = new node();
 3
 4     auto count = count_label(vec_data);
 5     size_t max_count = 0;
 6     for (auto& c : count) {
 7         if (max_count < c.second) {
 8             max_count = c.second;
 9             n->_label = c.first;
10         }
11     }
12     if (count.size() == 1)
13         return n;
14
15     auto selected_attr = select_attribute_gain_ratio(vec_data, attrs);
16     if (selected_attr.second.size() <= 1 || attrs.size() <= 1
17         || compute_gain_ratio(vec_data, selected_attr.second) < .1)
18         return n;
19     n->_attr = selected_attr.first;
20     attrs.erase(selected_attr.first);
21     for (auto& i : selected_attr.second) {
22         if (i.second.size() > vec_data.size() * .05)
23             n->_children[i.first] = construct_tree(i.second, attrs);
24     }
25     return n;
26 }
```

Infering Phase

Inferring from test data is somewhat easy. In inferring phase, the function just need to check whether the node has children **(line 2)**, and recursively call its child **(line 6)**. This simplicity is because all labels were saved at tree construction phase.

```
1 attribute_base::val_id node::infer(const data& d) const {
2     if (_attr == nullptr)
3         return _label;
4     auto val = d.attrs.at(_attr);
5     try {
6         return _children.at(val)->infer(d);
7     } catch (std::out_of_range& e) {}   // Not learned from train data
8     return _label;
9 }
```

Result

319 out of 346 (92.1%) of samples in **dt_test1.txt** were correctly classified when trained with **dt_train1.txt**.