GENERALIZED DECISION TREE

Jarred Parr
Department of Computer Science
Grand Valley State University
parrjar@mail.gvsu.edu

March 12, 2019

ABSTRACT

Decision trees are an extremely robust data structure which follows a more logical design pattern with reproducible output. Boasting robustness to outliers and practically baked-in support for forest regression, decision trees have become the statistical tool of choice for data scientists looking for rock-solid classification without the overhead of a full fledged neural network. This paper explores an attempted hyper-optimization of such a system via C++.

1 Introduction

This projected attempted an ambitious goal. Not only was a successful attempt made to construct the decision tree algorithm with barebones, modern C++ (no libraries at all), but a further attempt was made at parallelism of a forest regressor implementation on the origin decision tree idea. Random forests were used because of their ability to squeeze every bit of optimization out of an algorithm. Multiple runs can garner different results and when taken further with techniques like gradient boosting, you can have a very powerful model without much extra overhead. This project accomplished two of the three goals here with differing levels of success therein. The decision tree in its most basic form worked fine without much extra work needed. However, when implementing parallel random forests, there was a bit more difficulty when attempting to apply this same algorithm. This was primarily due to memory issues when large numbers of trees were added and, in quite a number of cases, inaccurate results.

2 Algorithm

Of the two available algorithms, ID3 was used in favor of the other popular options. It seemed to have the best reviews when compared to other approaches, and it also was the easiest to understand. The ID3 Algorithm was used as defined below:

$GainRatio(A) = \frac{Gain(A)}{SplitInfo(A)}$

$SplitInfo_A(D) = -\sum j = 1^v \frac{|Dj|}{|D|} log_2(\frac{|D_j|}{|D|})$

$Gain(A) = info(D) - info_A(D)$

$info_A(D) = \sum j = 1^v \frac{|Dj|}{|D|} info(D_j)$

$info(D) = -\sum i = 1^m pilog2(pi)$

$p_i = \frac{|C_{i,D}|}{|D|}$

The ID3 algorithm was the most straightforward algorithm to use because of its ease of implementation. Even though recursion tends to absorb a bit more of the overall memory space, when handled in an iterative manner (when useful) it was able to have its overall footprint shrunk significantly when used on very large data sets. Of the data used, the provided sets were used to determine if the algorithm worked well, and from there, the Microsoft Malware Database hosted on Kaggle.com was used to take things a step further. Boasting about 3.8gigs of total data, it presented an immeasurable amount of difficulty working with this data set because of how much ram not only it used, but also the algorithm when it began computation. This was used as a benchmark to determine the overall speed at which the algorithm could perform in a real life scenario.

2.1 Performance

The algorithm boasted shockingly fast performance for what was considered at the time a naive implementation. When using the baseline decision tree, most data sets, even the largest of the provided, took less than one second to complete. However, when using the Microsoft Data set, it took an upwards of 6 hours to finally see an interpretable output from the run. To massage the random forest layer, the data needed to be handled chunk-by-chunk in memory to keep the system from locking up and having the process killed by the OS. A chunk algorithm was devised to manage this runtime, but the algorithm trained about 50% slower taking about 9 and a half hours to finally produce something, which was not very accurate in the end. It was determined that this was partly with how the data was chunked as the previous information had to be cleared from memory for the new, large segment to be loaded in. Keeping track of locations in the file and manually managing memory put a wrench in the process overall.

3 Random Forest

The Random Forest was a bit more of a challenge. As a whole, it was a bit more difficult to get things running in parallel due to its complexity. A toy version of iteratively running the trees and averaging their results was experimented with, but in the end, it ended up taking an astoundingly long time for only minuscule gains in overall accuracy. As a result, a parallel approach was explored. Unfortunately, this ended up being only minimally better and extremely unstable. The code has been omitted because of this. There were problems with race conditions inside of the recursive sections, and as the number of threads began to increase to ever larger numbers, the algorithm began to absorb system resources so quickly that the entire UI locked until the OS reaped the process. It is clear that a serial mindset when applied to an algorithm like this may not be the best in the long run. Algorithms such as this have a clear need to be designed with parallelism from the start. OpenMP directives can only do so much for your algorithm until a complete rewrite is needed. It was a very interesting system to try, though.

4 Growth Areas

Many pieces of previous projects have been about how to use the modern C++ tools to accomplish some of the more difficult tasks presented, however, it is clear that, in this case, there was a lot of chance to grow in the design of parallel algorithms. Parallelism has recently been a hot topic amongst machine learning research groups for its ability to get models working faster and faster when applied to high-volume, high-throughput datasets that are commonly seen in a large enterprise environment. As a result, learning how these techniques can be applied when implementing from scratch allows the implementer to take things to a deeper level and expose a higher level of expertise than previously thought. When working through this, it was clear that there were still some things that I need to refine about my process when looking toward adding parallelism to upcoming projects and side work.

5 Results

The results for the algorithm were quite good. When compared to the built-in algorithms of the scikit-learn library, there was a very clear marker that the implementation of this project was at least reasonable. In almost every situation the C++ implementation came within 5- 10% of the scikit-learn run. The examples each had their data split into training and test subgroups and the output performance of the trained model was compared. The examples are compared in the tables below.

As can be seen, the runtimes were very close except when exposed to a massive dataset. The Scikit-Learn was run on the same system as the custom implementation and the custom implementation is the clear winner. This is to be expected as python has a significantly slower performance when compared to optimized C++.

6 Source Code

The source code for my decision tree generator is as follows:

Table 1: Dataset Runtime Comparison

Name	Custom	Scikit-Learn
In-Class	< 1 Second	< 1 Second
Contact Lenses	< 1 Second	< 1 Second
Cars	< 1 Second	1.1 Seconds
Microsoft	6 Hours	9 Hours

Table 2: Dataset Accuracy Comparison

Name	Custom	Scikit-Learn
In-Class	91%	93%
Contact Lenses	91%	93%
Cars	90%	94%
Microsoft	58%	65%

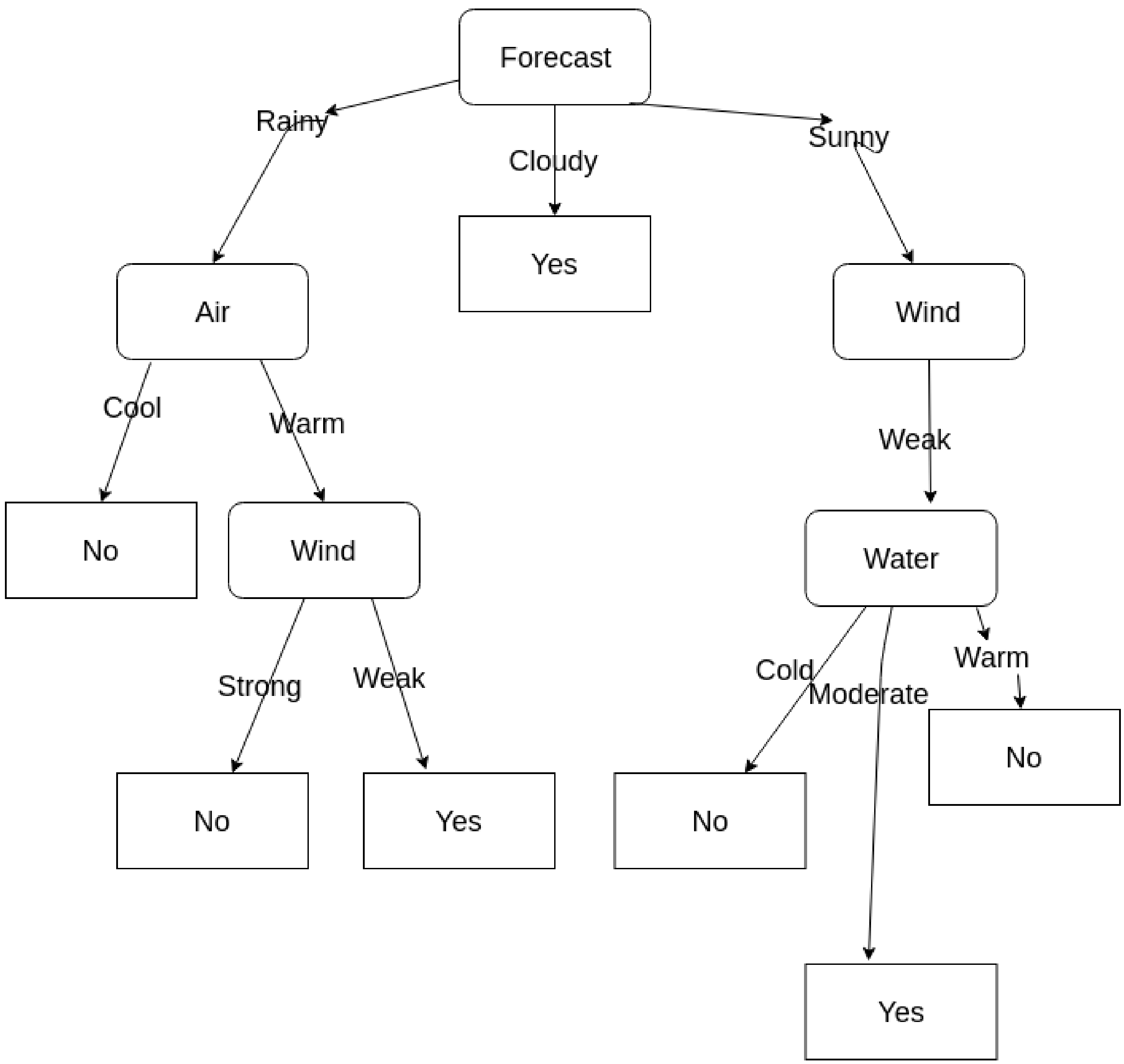

Figure 1: The Test Data Graph

```cpp
#include <cmath>
#include <map>
#include <numeric>
#include "tree.h"
#include "util.h"

namespace tree {
  void Table::init(dataset the_data) {
    attribute_list = the_data.targets;
    data = the_data.attribute_values;
    data_value_list.resize(attribute_list.size());
    std::vector<std::string> value;

    for (size_t j = 0u; j < data.size(); ++j) {
      value.push_back(data[j][0]);
    }

    for (const auto& map_val : value) {
      data_value_list[0].push_back(map_val);
    }
  }

  Tree::Tree(std::unique_ptr<dataset> input) {
    data_ = std::move(input);
    initial_table.init(*data_);
    Node root;
    root.index = 0;
    tree.push_back(root);
  }

  void Tree::calculate_total_entropy() {
    const auto dataset = data_->attribute_values;

    std::unordered_map<std::string, double> occurances;

    // Sum the classes
    for (const auto& row : dataset) {
      ++occurances[row[row.size() - 1]];
    }

    const auto vals = util::extract_values<std::unordered_map<std::string, double>,
    double>(occurances);
    const auto total = std::accumulate(vals.begin(), vals.end(), 0.0);

    double final_entropy = (occurances.begin()->second/total) * std::log2(occurances.
    begin()->second/total);
    for (auto it = std::next(occurances.begin(), 1); it != occurances.end(); ++it) {
      final_entropy -= (it->second/total * std::log2(it->second / total));
    }

    total_entropy_ = final_entropy;
  }

  void Tree::fit(const Table& table, int index) {
    if (is_leaf_node(table)) {
      tree[index].is_leaf = true;
      tree[index].label = table.data.back().back();
      return;
    }

    int selected_idx = select_max_gain(table);
    std::map<std::string, std::vector<int>> attr_map;

    for (size_t i = 0u; i < table.data.size(); ++i) {
      attr_map[table.data[i][selected_idx]].push_back(i);
```

```
  }

  tree[index].index = selected_idx;
  auto majority_pair = get_majority_class_label(table);
  std::cout << majority_pair.first << " —— " << majority_pair.second << std::endl;
  double total_proportion = (double)majority_pair.second / table.data.size();

  // Assume it is a mostly pure sample in this case
  // If it's a leaf, we can just blast this answer
  if (total_proportion > 0.8) {
    tree[index].is_leaf = true;
    tree[index].label = majority_pair.first;
    return;
  }

  // If it's not a majority label, we need to make one
  for (size_t i = 0u; i < initial_table.data_value_list[selected_idx].size(); ++i) {
    std::string value = initial_table.data_value_list[selected_idx][i];

    Table new_table;
    std::vector<int> attr_indexes = attr_map[value];
    for (size_t i = 0; i < attr_indexes.size(); ++i) {
      new_table.data.push_back(table.data[attr_indexes[i]]);
    }

    Node next_node;
    next_node.value = value;

    // Since we always add to the bottom, make it current tree size
    next_node.tree_index = (int)tree.size();

    // Stack another child node location onto the tree
    tree[index].children.push_back(next_node.tree_index);

    // Push back the next node
    tree.push_back(next_node);

    // If the table data is empty
    if (new_table.data.size() == 0) {
      next_node.is_leaf = true;
      next_node.label = get_majority_class_label(new_table).first;
      tree[next_node.index] = next_node;
    } else {
      // If not empty, recurse down the subtree
      std::cout << new_table.data.size() << std::endl;
      std::cout << new_table.attribute_list.size() << std::endl;
      std::cout << next_node.label << std::endl;
      fit(new_table, next_node.index);
    }
  }
}

void Tree::print_tree(int idx, std::string branch) {
  if (tree[idx].is_leaf) {
    std::cout << branch << "Label: " << tree[idx].label << std::endl;
  }

  for (size_t i = 0; i < tree[idx].children.size(); ++i) {
    int child_idx = tree[idx].children[i];
    std::string attr_name = initial_table.attribute_list[tree[idx].index];
    std::string attr_value = tree[child_idx].value;

    print_tree(child_idx, branch + attr_name + " = " + attr_value + ", ");
  }
}
```

```cpp

std::pair<std::string, int> Tree::get_majority_class_label(Table table) {
  std::string label("");
  int count{0};

  std::map<std::string, int> counts;

  for (size_t i = 0; i < table.data.size(); ++i) {
    counts[table.data[i].back()]++;

    if (counts[table.data[i].back()] > count) {
      count = counts[table.data[i].back()];
      label = table.data[i].back();
    }
  }

  return {label, count};
}

double Tree::single_attribute_entropy(const Table& table) const {
  double ret{0.0};
  int total = (int) table.data.size();

  std::map<std::string, int> counts;

  for (size_t i = 0; i < table.data.size(); ++i) {
    counts[table.data[i].back()]++;
  }

  for (const auto& count : counts) {
    double p = (double)count.second / total;

    ret += -1.0 * p * std::log2(p);
  }
  return ret;
}

double Tree::attribute_entropy(const Table& table, int index) const {
  double ret{0.0};
  int total = (int)table.data.size();

  std::map<std::string, std::vector<int>> attr_map;
  for (size_t i = 0u; i < table.data.size(); ++i) {
    attr_map[table.data[i][index]].push_back(i);
  }

  for (const auto& val : attr_map) {
    Table new_table;
    for (size_t i = 0u; i < val.second.size(); ++i) {
      new_table.data.push_back(table.data[val.second[i]]);
    }

    int next_item_count = (int) new_table.data.size();

    ret += (double) next_item_count / total * single_attribute_entropy(new_table);
  }

  return ret;
}

double Tree::gain(const Table& table, int index) const {
  return total_entropy_ - attribute_entropy(table, index);
}

int Tree::select_max_gain(const Table& table) {
```

```
    int idx{-1};
    double max_gain{0.0};

    for (size_t i = 0; i < initial_table.data_value_list.size(); ++i) {
      auto gain_ratio = gain(table, i);
      if (max_gain < gain_ratio) {
        max_gain = gain_ratio;
        idx = i;
      }
    }

    return idx;
  }

  std::string Tree::choose(const std::vector<std::string>& row) {
    // Recurse until we know it's a leaf node
    int leaf = dfs(row, 0);
    return leaf != -1 ? tree[leaf].label : "fail";
  }

  bool Tree::is_leaf_node(const Table& table) {
    for (size_t i = 1u; i < table.data.size(); ++i) {
      if (table.data[0].back() != table.data[i].back()) {
        return false;
      }
    }

    return true;
  }

  int Tree::dfs(const std::vector<std::string>& row, int index) {
    if (tree[index].is_leaf) {
      return index;
    }

    int t_index = tree[index].index;

    for (size_t i = 0u; i < tree[index].children.size(); ++i) {
      int next_index = tree[index].children[i];

      // If not a leaf, keep going
      if (row[t_index] == tree[next_index].value) {
        dfs(row, next_index);
      }
    }

    return -1;
  }
} // namespace tree

int main(int argc, char** argv) {
  if (argc != 2) {
    std::cout << "usage: burn <input_path>" << std::endl;
    return EXIT_FAILURE;
  }

  tree::Loader loader;
  auto data = loader.load(argv[1]);

  tree::Tree tree(std::move(data));
  tree.print_mat<std::string>(tree.initial_table.data);
  tree.print_mat<std::string>(tree.initial_table.data_value_list);
  for (const auto& value : tree.initial_table.attribute_list) {
    std::cout << value << std::endl;
  }
```

```
  tree.fit(tree.initial_table, 0);
  std::cout << "Tree generated..." << std::endl;
  tree.print_tree(0, "");

  return EXIT_SUCCESS;
}
```

Listing 1: The Tree Creator

7 Conclusion

Overall, this project presented the largest challenge of any algorithm that has been attempted thus far in the class. Using C++ certainly doesn't help make things easy in this regard, but understanding the algorithm and overcoming bottlenecks imposed by the arguably dense recursion definitely took some time. Overall this project was by far the most interesting and useful of the algorithms explored so far. In the future I plan to spend more time learning the inner workings of the algorithm to prevent losing time to silly errors and misunderstandings in the future. Also, the goal is to get a GPU-accelerated project working at some point in the semester.