ID3, Reduced Error Pruning, and Random Forests - BITS F464 Assignment 2

Introduction

There are several methods to classify data from a dataset. One notable method is to use decision trees which are recursively generated data structures based on the training data.

For this assignment, the task is to classify whether a given instance of a person's details can be used to deduce if they earn more or less than 50,000 USD per annum.

This project is a modern, C++11 implementation of a Decision Tree.

Installation

This project has dependencies on the SQLite3 library and SQLiteCpp for interfacing SQL with C++ code. The project has been built with CMake. The libraries can be downloaded with:

```
sudo apt install libsqlite3-dev libsqlite3-0
 cmake
```

The SQLiteCpp must be built from source. Instructions for doing so can be found here: https://github.com/SRombauts/SQLiteCpp/

Once the library has been installed, clone or unzip the files to your favourite folder and run:

```
mkdir build
cd build
cmake ..
make -j4
```

Executables can be found in `build/`. For testing your setup, run `./test_p` which will calculate the score of the decision tree computed by a standard ID3 algorithm.

Code Base

The code depends on two main header files:

DataEngine.h

The ID3 algorithm requires several lookups to find data satisfying a certain attribute to classify it accurately. The DataEngine header file encapsulates all of the common lookups required in the ID3 algorithm, such as getting the count of an instance satisfying certain properties and the list of distinct attributes of the dataset among many others.

The DataEngine works by using an instance of an SQLite3 database loaded in RAM. As the given database is only 5MB, this can be done without having to worry about running out of memory.

DecisionTree.h

This header file contains the functions to generate a DecisionTree. The class encapsulates a structure called DecisionTreeNode which specifies the data contained in each node of the Decision Tree.

The struct definition of DecisionTreeNode is as follows:

```
enum NodeType {
    TerminalNode,
    AttributeNode,
    RootNode
};

struct DecisionTreeNode {
    std::vector<DecisionTreeNode*> children; // each node of the DecisionTreeNode has a vect
    NodeType type; // useful for letting us know whether we are at the "Top" (root) or "Bott
    ItemPair attributePair; // attributes and values are stored as pairs, like <"outlook", "
};
```

Given a dataset, the DecisionTree first stores the data in its member DataEngine object. which in turns initialises the database in RAM. After this, the DecisionTree builds the tree starting from its member root node.

Using `traverseTree(..)`, the list of the deduced rules is linearized and printed to standard output. This can be output redirected to a file.

Reading the deduced rules from a file greatly speeds up testing the DecisionTree as training the DecisionTree can take several hours depending on the computational power of the target computer. However, reading a list of rules from the file and computing the DecisionTree can be done in a matter of seconds on a reasonably fast computer.

The DecisionTree class has been completely created from scratch and can handle any generic dataset with any number of continuous and discrete values.

test.cpp

This file is used for testing the performance of a given DecisionTree based on its Accuracy, Precision, F-Value, and Recall.

This works by taking a rule base, that is the list of deduced rules from a file and building a DecisionTree from the deduced rules. The first argument passed to the program is used as the path to the file containing the list of rules.

If the last argument if the constructor of the DecisionTree is set to `false`, the tree is built from the rules defined in the rule base. If it is set to `true`, the tree is built by training it from the training instances. This can be upto 100 times slower than just building the tree from the file.

The properties of the DecisionTree, such as the schema of the SQL database, the list of discrete and continuous attributes, and the path of the training and test data is specified in this file.

The pre-built `rule_base` file contains the list of the rules deduced from the standard ID3 algorithm.

prune_rules.cpp

This file initialises the tree from the `rule_base` file which contains the rules deduced from the training of the decision tree.

The tree is then aggressively pruned, very heavily favouring nodes which improve the overall accuracy of the tree. Pruning is done by a custom BFS algorithm which checks if the accuracy of the tree is improved by deleting the node and replacing it with a TerminalNode having the result of the majority instances of the tree at that point. In case the accuracy is improved, the node is deleted and replaced with a TerminalNode. Otherwise, the node is kept at the same position and tree is recursively traversed till all the nodes have been checked.

This algorithm requires the most execution time of the three. In return, it also provides the best accuracy at a startling **96%**!

Random Forests

These have been created using three separate files:

random_dataset.cpp

Randomly selects N elements from the dataset and creates M number of files with these values. In this case, 128 files are randomly produced, each consisting of 512 training instances. The datasets are stored in random_forest/adultTraining*.

build_random_trees.cpp

Builds decision trees from the random datasets created in the previous file. The tree is then linearized and written back to a file so it can be quickly accessed the next time. These files are stored under random_rule_bases/random_rule_base*.

test_forest.cpp

This starts off by reading the generated rule bases in random_rule_bases/random_rule_base* and building all the decision trees.

The random forest is constructed in an `std::vector` of pointers to DecisionTreeNode root nodes. The Random Forest evaluates a given instance by taking a vote among the 128 generated decision trees. The majority vote is taken as the result of the provided instance.

The Random Forest method provides a good combination of short execution time and good performance. However, the performance does not improve much by increasing the number of decision trees in the forest or increasing the number of training instances used to produce each decision tree.

Results

All of the following decision trees were tested on a modern laptop with a i5-6300U and an SSD. The code has been compiled using the `-std=c++11` and `-O2` g++ optimisation flags to improve performance. The calculations for the statistics shown below are by comparing the output of the decision tree with the expected value in `adult.test` file.

Standard ID3

The Standard ID3 algorithm shows some promising results. It scores an accuracy of 81%, which is higher than expected. In the case, the ID3 algorithm deduces over 9000 rules. The running time of the algorithm could be improved by reducing the number of SQLite queries to the RAM database. The program consumes 12MB of RAM with this dataset.

Stats

Accuracy: 0.810359

Precision: 0.618606

Recall: 0.594865

F1 Score: 0.606503

Validation Time: 0m11.050s

Training Time: 108m26.086s

Reduced Error Pruning

The Reduced Error Pruning algorithm shows interesting results. As it aggressively prunes for improving the trees accuracy, it gives excellent accuracy and precision. The Recall and F1 score are quite poor, however. Reduced Error Pruning has a very long execution time and hence, should only be used in instances where long time training time is not a problem.

Stats

Accuracy: 0.988082

Precision: 1

Recall: 0.0363636

F1 Score: 0.0701754

Validation Time: 0m2.042s

Training Time: 108m26.086s + 64m12.762s (the training time includes the time for constructing the tree using standard ID3 in the first place)

Random Forest

The random forest approach to Decision Trees provides some of the best results. It has better results than the standard ID3 algorithm, while being over 20 times faster when it comes to training with 128 trees, each trained on 512 instances. The random forest's performance does not improve by much on increasing the number of trees or the training instances. The same has been cited in many papers about this subject. The only drawback of the random forest approach is that it uses twice as much RAM as the standard ID3 algorithm.

Stats

Accuracy: 0.835857

Precision: 0.705351

Recall: 0.57

F1 Score: 0.630493

Validation Time: 0m13.217s

Training Time: 7m55.311s

Conclusions

ID3 is a very effective machine learning classifying algorithm. The algorithm can be adapted in many ways to improve the training time and accuracy as the problem requires. It can be adapted to solve several problems when large datasets are given. Furthermore, the construction of the tree is easy to visualise and the rules deduced by the tree are simple to understand. To improve performance, the algorithm could be rewritten to support multithreading and to recursively search the best attribute on each thread.