Big Data Analytics Programming Assignment 2

Milos Dragojevic - r0780142 - milos.dragojevic@student.kuleuven.be

1 Introduction

The goal of this assignment is to deliver an efficient C++ implementation of an ensemble of decision tree learners with high accuracy and low runtime.

2 Optimising the Decision Tree

An initial naive algorithm implementation calculated the gain for each specific value in each column. This naive implementation exhibited quadratic runtime of 30 minutes. To counter this, an ascending sort is performed on each column in which the best threshold is searched. After the sort, the algorithm was adapted to linearly read the values only once in the table, and to calculate the Gini score and Information Gain only when a value change is detected. This simple optimisation reduced the runtime to 220 seconds.

Another optimisation decision was to implement multi-threading using futures and std::async. This was done at the level of the *buildtree* function as it is recursive and allows to start building the left and right sides of the subtrees in parallel. Multi-threading was also implemented in the $find\_best\_split$ function as to parallelise the process of finding the best threshold in each of the columns. With this full parallelisation, the runtime was reduced to approximately 130 seconds, however a drawback appeared as the memory consumption became huge and the program was firing up to 8000 threads in parallel.

Consequently, a way to limit the multi-threading up to a certain tree depth was implemented, which restricted the thread concurrency and reduced the runtime to 90 seconds. From analysis, it was identified that the core function of the decision tree building is finding the best threshold as for the covtype dataset this function is called more than 550000 times. Within this function, the sort was the most time expensive process as sorting strings is memory consuming as they take by default 32 bytes of memory, and as it is inefficient to compare numbers represented as strings as it requires to convert these to integers to allow for comparison. This led to the next optimisation step where the strings are converted to int type before the sort when searching for the best threshold. As a result, the runtime dropped to 22 seconds.

As the memory usage was still too large to be able to run the bagging of 5 trees within 2 GB of memory, it was decided to convert the full dataset in memory from vectors of vectors of strings to vectors of vectors of integers (int type) and to pass between function pointers to vectors of int. This improvement both reduced the global memory footprint as an int only takes 4 bytes of RAM whilst speeding-up the handling of the dataset. The decision tree could then be built in 12 seconds.

Applying the decision tree on the covtype dataset reveals that one side of the tree presents two times more nodes than the other one and is much deeper (57 levels vs 38 levels). Thus it was determined that to make multi-threading more efficient it should not depend on the absolute tree-depth but more on the amount of information to be processed by the sub-threads. A loop was ran to identify the best cut-off value between starting new threads versus a sequential processing of the *buildtree* function for the sub-nodes. An empirical value of 25000 rows was found to be the best cut-off on the department computers, meaning that above this value it is worth starting new threads while below this value it is more efficient to build the left and right subtrees sequentially. Finally, to maintain memory usage at a level low enough to be able to generate the learning curve when bagging up to 60 trees, multi-threading was disabled for the $find\_best\_split$ function.

Another point to mention is that on the department machines, a flag was added in the CMakeLists.txt file for the optimisation of the code which decreased the runtime further.

Using these optimisation settings, the memory usage for bagging 60 trees was still below the 2GB limit and the algorithm runtime came down to 3 seconds.

3 Effects of Tree Number on Ensemble Accuracy

From figure 1, it can be seen that increasing the number of bootstrap trees in an ensemble model (grey line) results in a higher accuracy than the one for a single tree (blue line) even if each individual bootstrap tree has a lower accuracy (orange dots). Important to note is that there is a major increase in accuracy when going from 2 to 3 bootstrap samples in the ensemble. Further, increasing the number of models beyond 50 does not provide with a substantial accuracy increase.

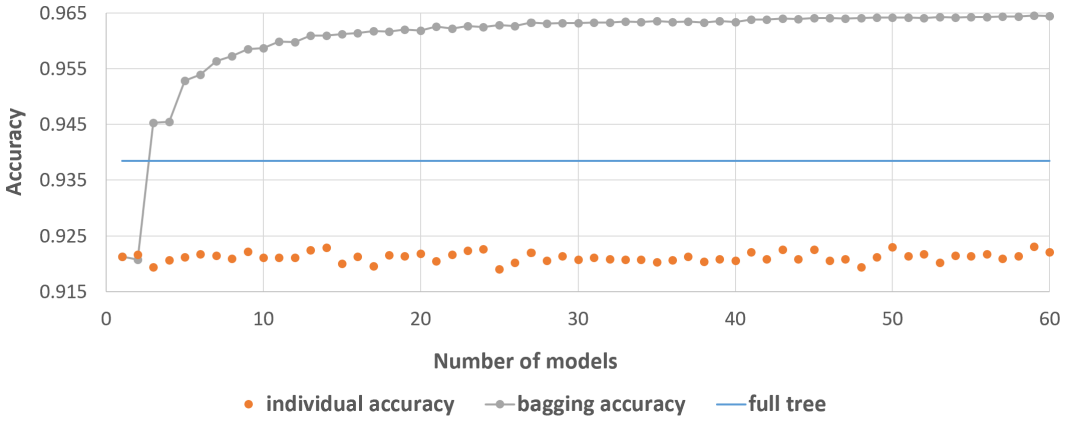

Figure 1: Influence of the number of trees in an ensemble on accuracy

Comparing the bagging performance of the described implementation with sklearn gives quasi-identical accuracy results which proves that the implemented algorithm is correct.

Thus, increasing the number of trees in an ensemble model causes an increase in accuracy up to a certain level because of reduced variance in the implementation as each extra tree will provide a small amount of information which will improve slightly the model.

4 Conclusion

The implementation of CART as an ensemble model led to a performant balance between execution time and accuracy. Accuracy is comparable with the results obtained with sklearn while the execution time is of 3 seconds.

Further research could focus itself on implementing thread pools or on further optimising multi-threading for the $find\_best\_split$ function similarly to what was done for the $buildTree$ function.