Treelite: toolbox for decision tree deployment

Hyunsu Cho
University of Washington
chohyu01@cs.washington.edu

Mu Li
Amazon Web Services
mli@amazon.com

ABSTRACT
This paper introduces a brand new tree library treelite1. The library is a toolbox to facilitate easy deployment of models and accelerate prediction performance. It has a Python wrapper that allows users to integrate it as part of their workflow. Treelite is able to read tree ensemble models that are trained by any tree libraries, including XGBoost [1], LightGBM [2], and scikit-learn [3]. Treelite is also designed to minimize dependencies at the time of deployment. It used to be the case that one had to ship his tree model with the original tree library that trained it; with treelite, it is no longer. Finally, treelite allows for optimizations that improve prediction performance without changing any detail of the model.

1 INTRODUCTION
Treelite is designed with deployment scenarios in mind. Deployment involves two machines: the host machine and the target machine. See Figure 1 for a pictorial representation of the workflow. The host machine is the machine that 1) has treelite installed and 2) stores the model of interest. The target machine on the other hand is the machine on which actual predictions will be made. We make no assumption as to whether the host and target machines will be the same or distinct. Most likely they will be distinct, as we may want to avoid installing treelite and other dependencies on the target system. The two machines are to exchange a single C file that contains all relevant information about the tree model. No other object is exchanged. The target machine requires only a functional C compiler to make predictions; it needs neither treelite nor other tree libraries (XGBoost, LightGBM, and others).

2 DESIGN OF TREELITE
2.1 Interoperability with many tree libraries
Treelite offers multiple front-end interfaces to work with other tree libraries. First, there is a dedicated interface to import models produced by XGBoost [1], LightGBM [2], and scikit-learn [3]. In particular, treelite provides a seamless integration with XGBoost.

In addition, treelite provides the model builder API that lets the user programmatically specify his model. This is useful if the user uses other packages to train the model. The user is asked to specify every test condition as well as every leaf output.

2.2 Extensible modular design
As shown in Figure 1, there is a clear separation between the front-end (parts of treelite that interact with other tree libraries) and the back-end (parts of treelite that generate deployable C files). The modular design facilitates future extension. A crucial piece in the design is a common schema for decision tree ensembles. The front-end and the back-end shall communicate in no way other than the common schema. The schema is designed to accommodate both random forests and gradient boosted trees.

2.3 Faster prediction with rule compilation
The given tree ensemble model is converted into a C program by “compiling” decision rules into nested if-else conditions. Each test node is converted into a pair of if-else statements as follows:

```
if (__builtin_expect( /* comparison test for the test node */ ) {  
  /* ... code for the left child node ... */
} else {
  /* ... code for the right child node ... */
}
```

The left and right child nodes are then recursively expanded into C code until every leaf node is expanded.

By compiling rules, we enable compile-time optimizations that are specific to the model being examined. Previously, a model would be loaded from a file at runtime, and the prediction logic would be oblivious to any information specific to that particular model. With rule compilation, however, the compiler has access to every bit of information from the particular model being compiled, which it can use to further optimize the resulting machine code. As an early demonstration, treelite offers two optimizations2.

2.3.1 Annotate conditional branches. We predict the likelihood of each condition by counting the number of data points from the training data that satisfy that condition. If a condition is true at least 50% of the time (over the training data), the condition is labeled as “expected to be true”; otherwise, it is labeled as “expected to be false.” Both GCC and clang compilers provide the compiler intrinsic __builtin_expect specifying the likely outcome of a condition. This helps the compiler make more intelligent decisions about the ordering of branches, thereby improving branch prediction.

2.3.2 Use integer thresholds for conditions. This optimization replaces all thresholds in the test nodes with integers so that each threshold condition performs integer comparison instead of the usual floating-point comparison. The thresholds are said to be “quantized” into integer indices. On such platforms as x86-64, replacing floating-point comparisons with integer ones improves performance by 1) reducing executable code size and 2) improving data locality.

3 RELATED WORKS
There are several algorithmic approaches to accelerate prediction. Early stopping [4] finds that, for “easy” data points, we can use only first 25-75% of member trees without loss of accuracy. Tree indexing [5] embeds trees as objects in a spatial database and uses a R-tree-like structure to reduce tree traversals to sub-linear complexity. Neither early stopping nor tree indexing were incorporated in this

---

1Hosted at https://github.com/dmlc/treelite; documentation at http://treelite.io

2Notice that the model information is wholly preserved; the optimizations only affect the manner at which prediction is performed.
work. Tree indexing will not work well with high-dimensional data, as spatial access trees, like KD-trees, suffer from the curse of dimensionality [6].

Other works consider system-level optimizations to speed up prediction without changing models or prediction accuracy. Several existing works implement the If-Then-Else approach [2, 7–10], in which each decision tree is translated into a sequence of if-then-else blocks. A recent blog post by Facebook [11] utilizes distribution of the training data to annotate branches. A high-speed corner detector implementation [9] also annotates branches, this time utilizing runtime performance behavior of the program over data samples. A fast document ranker [8] expresses interleaved traversal of decision tree ensembles as simple logical bitwise operations, thereby eliminating branches and control dependencies. Approaches known as Pred and VPred [7] eliminates control dependencies by unrolling feature-threshold comparisons.

There are deployment solutions geared towards deploying models trained with particular software packages. LightGBM [2] has recently added support for converting its own models into C++ code. The sklearn-protobuf package [10] offers a suite of Python scripts to convert many scikit-learn models. The latter supports models other than decision tree ensembles, such as Support Vector Machine and multi-layer perceptrons.

4 BENCHMARK

One dedicated EC2 instance of type m4.16xlarge was used to run a benchmark. For each dataset described in Table 1, we trained a 1600-tree ensemble using XGBoost [1] and then another 1600-tree ensemble using LightGBM [2]. After obtaining the models, we made predictions on batches of varying sizes that were sampled randomly from the data. Fifty samples for each size was taken. For treelite, we imported each tree model (either trained by XGBoost or LightGBM), exported it as a shared library, and then loaded it onto memory to make predictions. For XGBoost and LightGBM, we simply loaded the respective model by calling appropriate API calls. Throughput is computed as the number of lines predicted per second.

### Table 1: Characteristics of three datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>allstate(^3)</th>
<th>higgs(^4)</th>
<th>yahoo(^5)</th>
</tr>
</thead>
<tbody>
<tr>
<td># of features</td>
<td>4,227</td>
<td>28</td>
<td>699</td>
</tr>
<tr>
<td>Training set size</td>
<td>7M</td>
<td>10M</td>
<td>473,134</td>
</tr>
<tr>
<td>Validation set size</td>
<td>3M</td>
<td>100K</td>
<td>71,083</td>
</tr>
</tbody>
</table>

Overall, treelite outperforms both XGBoost and LightGBM by a factor between 2x and 4x. The LightGBM model for allstate is particularly remarkable, in which treelite is 7.5x faster than LightGBM. Note that any significant performance improvement is observed only for sufficiently large batches of data points (at least 10,000 or more). So treelite is currently suited for performing large batch prediction. In general, the bigger the batch is, the overall throughput would be.

Figure 1: How to deploy tree models with treelite

Figure 2: Throughput measured over varying sizes of data batches. The error bars represent 95% confidence interval.
REFERENCES


