Chapter 4

Algorithm implementation

This chapter gives a broad overview of the design and implementation work that was carried out for this thesis. The aim is to explain the key details and features in order to facilitate future work that builds on this thesis. Readers may also skip this chapter if this is not part of their objective. The topics discussed include:

- Starting point
- Design decisions and implementation features
- Discovered bugs and fixes
- Transferring the code into an SGX enclave

Be aware that the design and implementation parts are not strictly separated. This facilitates justifying certain design choices that were made along the way. There are also some remarks about things that did **not** work out as expected.

4.1 Starting point

With DPBoost ([30], 2020) Li et al. set the cornerstone behind DP-GBDT learning. Apart from some slight changes (described in TODO ref background) this is the algorithm that is used in our implementation. The results of their work are promising, yet should be taken with a grain of salt. For instance Moritz Kirschte found a bug in XY calculations TODO. Further, the results of some plots in their evaluation section are not really helpful and apparent. can I say that? Li et al. do provide source code for their implementation. They are essentially using the LightGBM [27] framework but with several modifications scattered throughout a number of files. As this is a bit convoluted, Théo Giovanna subsequently build a first standalone Python implementation[1] of said algorithm during the course of his master's thesis (submitted Feb. 2021). Unfortunately, this implementation has weaknesses as well, most notably speed. It's just a bit too slow for effective testing and tuning. With the overlying goal of performing DP-GBDT in an enclave, a decision had to be made: Do we either use some unofficial framework to squeeze the existing Python code into an enclave, or, do we use the Python code as reference and rebuild the algorithm in C/C++?

4.2 C++ DP-GBDT - Design and implementation

In hindsight, this was a good decision. It lead to the discovery of several bugs, to substantial runtime improvements, and enabled an easier transfer to SGX. Even though there do exist some

[1] https://github.com/giovannt0/dpgbdt

machine learning libraries for C++, we decided against using one. Mainly because it would complicate the upcoming hardening process. It would have been difficult to analyze and assure the absence of side-channels. As a result, many tasks that only require a single line of code in Python had to be replicated in C++. Performing cross validation is one such example. The only library that was utilized in the algorithm is the C++ standard library [25]. The c++11 standard was chosen as it is the latest with SGX compatibility.

Project structure The project[2] essentially consists of five DP-GBDT versions (figure 4.1) and some additional infrastructure to run/verify/evaluate/... them. This separation is a big advantage, since both hardening and running a program inside an enclave tend to clutter the underlying code to some extent. The outcome of running the different variants is the same: equal input means equal output. Details on running and testing them can all be found in the public repository.

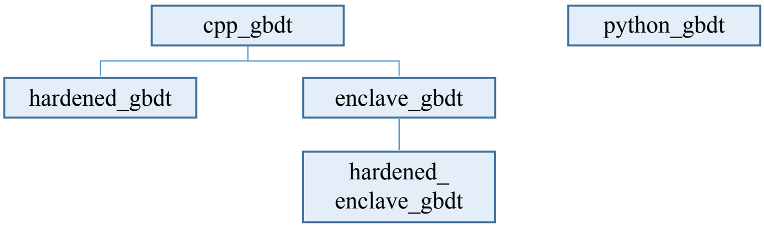

Figure 4.1: Project structure

For the remainder of this section we are mainly referring to `cpp_gbdt`, the non-hardened C++ version. It is roughly 2500 lines of code in total. The algorithm is clearly visible and compiler optimizations and threads lead to solid runtime performance. This version is ideal for experimentation with the underlying algorithm.

Feature selection With future hardening work in mind, it was vital to get a fully functional and correct prototype. To achieve this in the available time frame some compromises had to be made. These came in the form of omitting certain algorithmic options and features that were presented in either the DPBoost paper or in Théo's thesis. The following features are not available:

- Best-leaf first decision tree induction
- "2-nodes" decision tree induction algorithm
- Sequential composition of multiple ensembles
- Generation and use of synthetic insurance datasets
- Multiclass classification

This is of course unfortunate, however, if required they can always be incorporated into the current implementations with relatively small effort.

Datasets and parsing Currently only regression and binary classification prediction tasks are supported. To demonstrate this four datasets are available and ready to use in the project: Abalone [11], YearMSD [12], Adult [13] and BCW [14]. To our knowledge, this is first time detailed and correct DP-GBDT results have been generated for these datasets. This should be a valuable foundation for future experiments and tuning of the algorithm. Further, it is very easy to add other datasets: Given a new dataset (in comma separated form), only about 15 lines of code

[2] https://github.com/loretanr/dp-gbdt/blob/main/TODO.todo

in the `Parser` class are required to start using it. This is just for specifying how the new dataset looks (size, target feature, feature type, regression/classification etc). There's an example of this in appendix B.

Comparability and verifiability This was one of the main challenges during development. How can you ensure that the results are accurate? Even though a reference implementation was available for comparison, there were multiple obstacles:

- Bugs in the reference implementation
- The DP-GBDT algorithm being full of randomness
- The python implementation being painfully slow
- Numerical issues caused by the use of floating point numbers

Let's take a closer look at some of these points. Naturally, being the first of its kind, the reference implementation contained multiple bugs, that were only gradually discovered. It ranges from programming slips to relatively important parts of the algorithm that were misinterpreted. Next, regarding randomness, Python and C++ libraries use different sources and mechanisms for random number generation. This obviously obstructs comparison of Python and C++ DP-GBDT results. Unfortunately, it was not as simple as setting the same seed in both implementations. The problem was solved by enabling deactivation of every bit of randomness. This includes the exponential mechanism, shuffling of the dataset, Laplace Mechanism for leaf clipping, random selection of samples for the next tree and more. Last but not least, a word on numerical details, which were the most aggravating kind of bugs to deal with. Imagine that at some point in the algorithm you need to compute a weighted sum $\sum_{i \in \mathbb{N}} w \cdot a_i$. Being a performance conscious programmer, you unsuspectingly factor out w. You will not notice any problem until you later run the algorithm on some particularly large dataset. In the 4th cross validation fold, from decision tree number 34 onwards, the Python and C++ trees are suddenly not matching anymore. With a few hairs turning gray in the process, you track the difference down to this exact weighted sum. Somewhere deep in the Python sklearn library, this sum was calculated without factoring out w. A friendly reminder that neither addition nor multiplication is associative with IEEE 743 double precision (64-bit) numbers [21]. Because of the nature of the algorithm, there are multiple spots where computation is done on the same values over and over again. For example: all samples have gradients that are constantly updated whenever a tree is built and added to the ensemble. Tiny numerical differences in those gradients can build up to the point where they affect the shape of a new tree. For instance, when searching the next split in an internal node, slightly different gradients lead to a different gain being computed which can result in a different split being chosen. This means different samples continue to the left/right children of that node. As soon as such things happen, the trees in an ensemble will eventually start to look completely different. Since these kind of issues are absolutely ubiquitous in DP-GBDT, the only viable solution was to regularly clip prone values to about 12 decimals.

However, given these hacks, it was possible to eliminate the differences between the python and C++ algorithm. To keep it that way and make future development easier, a verification framework was created. In its core it's a bash script that (compiles) and runs both the Python and (un/hardened) C++ code on a number of different datasets. During execution of the algorithms it will continuously compare intermediate values of the implementations. This way you do not only quickly recognize if a programming mistake was made, but you can also get the exact location where the two implementations started diverging. It will give you convenient and nicely formatted output, an example is shown in appendix B.

Logging and documentation We use fmtlib[3] for logging. You can choose between different logging levels to get output of the desired degree. You can further print finished trees to the terminal. Doxygen[4] was used to generate useful documentation of the algorithm, that will help the programmer quickly getting an overview of the different components of the algorithm.

Runtime improvements Experimentation with a slow implementation is frustrating. The GBDT algorithm has around 20 hyperparameters that need to be optimized to achieve good results (see appendix B for details). Being a compiled language, C++ was naturally already quite a bit faster. There was still much to be gained however. In order to determine the right spots to steer our attention to, frequent profiling was done using Intel vTune[5] and Intel Advisor[6]. For illustration purposes you can find a snippet of the profiling output of the finished (optimized) C++ GBDT algorithm in appendix B. It will also give you an idea of where the current bottlenecks are. Carried out optimizations include multithreading, aggressive compiler optimizations, removing blockers in core/critical functions to incentivise vectorization. The actual performance gains achieved, and put into relation to the other implementation are depicted in chapter 6.3.

4.3 Discovered bugs and fixes

This section summarizes all bugs that were discovered while working on this thesis.

Python bugs List of significant issues encountered (decreasing severity):

- Illegal tree rejection mechanism → big performance boost for small privacy budgets
- Bugs in the parser → performance loss
- `leaf_clipping` never being enabled
- `use_decay` formula wrong

The idea behind tree rejection was the following: In normal GBDT (no differential privacy) the predictions get closer to their actual value with every single tree. However, due to the randomness introduced by DP, this is not always the case for DP-GBDT. In the latter, some of the trees might actually make the current prediction worse. Therefore the author of the Python implementation decided to set aside some samples, that are used each time after a tree is created, to judge whether it's a good/useful tree. Bad trees are rejected accordingly and will not be part of the final model. This rejection mechanism and the reuse of samples of course violates differential privacy. However, as the performance improvements are quite drastic, it may actually be worth paying privacy budget for. This was discussed with E. Mohammadi and M. Kirschte, who will likely explore this idea further in the future. Figure 4.2 shows the drastic difference between the (illegal) usage of the rejection mechanism and the correct way of utilizing all trees. On average around 15 out of 50 trees were rejected. This goes to show how seemingly small tweaks and bugs in the algorithm can have a huge impact on the results and lead to deceptively good results.

All mentioned bugs were of course corrected in the new C++ algorithm. For the sake of completeness this project also includes a fixed version of the Python code.

DPBoost bugs Technically the following point is not a bug, but rather an important aspect that was not addressed by the paper: When trying out different splits to find the one with the

[3] https://github.com/fmtlib/fmt

[4] https://github.com/doxygen/doxygen

[5] https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[6] https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html

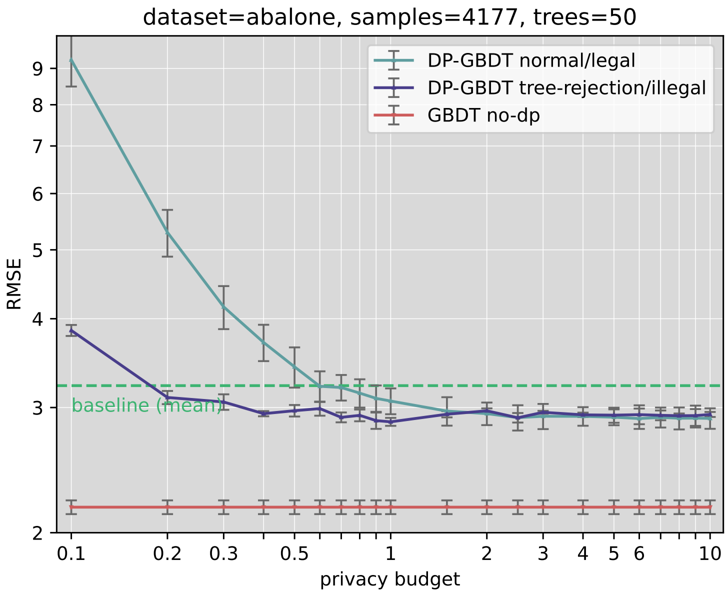

Figure 4.2: Comparison: Tree-rejection vs correct DP-GBDT

highest gain, iterating over the real numerical feature values is a bad idea for the following reason: When an adversary learns the final model (which is an assumption of our differential privacy proof), he can infer what feature values are actually present in the dataset. This problem has to be addressed by either adding enough noise, or by using a grid of fixed-size intervals.

Grid usage Given a fine enough grid spacing, the same splits will be found as if you were iterating over the real feature values. This is depicted in figure 4.3. The key observation is that splits only change whenever an existing feature values is passed.

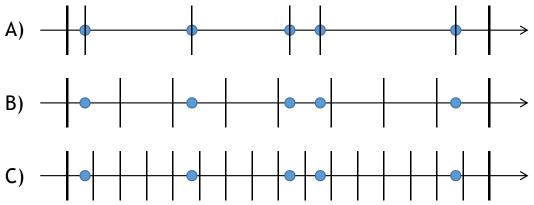

Figure 4.3: Grid step size: A) split on feature values, B) not fine enough, C) catches all splits

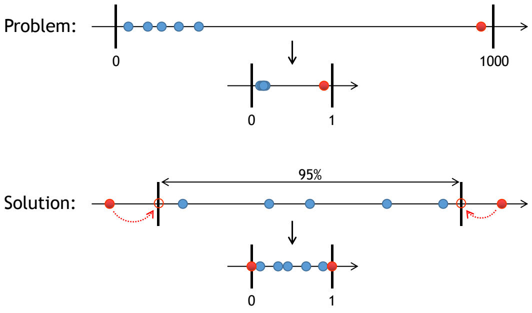

Figure 4.4: Scaling feature values into the grid using 95% confidence interval

Certain datasets however, contain numerical features with values of very different magnitudes. This creates a problem: Imagine a feature A with values in [0,1] and feature B with values in [0,1000]. You can certainly define a grid that is fine enough for feature A, while simultaneously accommodating all feature values present in B. However that would be extremely inefficient. A natural solution would be scaling the feature values into the grid. If the numerical feature values of the entire matrix X are in [0,1], we could choose our grid borders accordingly and not waste as much computation power. This comes with another problem though: A malicious entity M, e.g. an insurance customer or even a malicious service provider, could submit bogus questionnaire data: If M chooses a value of 100'000 for feature A, scaling [0,100'000] into [0,1] would cause the legitimate participants' values to lie extremely close to each other. Possibly even to the point where the grid is not fine enough anymore to consider all possible splits. This way an attacker could essentially eliminate certain features from having any effect in the training. This is illustrated in figure 4.4.

One solution to this problem is the use of confidence intervals. A common percentile to choose is 95%. The idea is essentially to clip any outliers to the chosen percentile. In the case of a 95% percentile, the 2.5% and the 97.5% border of the values need to be determined. Subsequently the outliers above the upper boundary and below the lower boundary are clipped to their respective boundary. The challenging part is doing this in a differentially private fashion. Fortunately this paper [10] proposes multiple options on how to achieve this as well as corresponding R code on Github[7]. For this project the EXPQ method was adopted from the aforementioned work. According to the paper, it's not the most efficient scheme in terms of privacy budget, but it was the simplest one to understand and port to C++. The price for this is of course a small loss of information in your training data, as well as a small amount of privacy budget. More details on this in section 6.2.

In the final implementation you have both options. If you know enough about the values in your dataset, i.e. that your data values are going to be in a certain range, You can manually set constant grid borders, and thus not spend any privacy budget. Or, you enable the feature scaling and enjoy a shorter execution time. Of course, the grid usage can also be entirely turned off for experimentation in a non-real-world deployment setting.

4.4 Transferring DP-GBDT into an SGX enclave

This section outlines the steps required to move the DP-GBDT algorithm into an enclave. At this point, the details of attestation, encryption of customer data etc. are ignored (this will happen in chapter 7). The main goal is to run the core algorithm in the enclave, as a proof of concept and for execution time measurements.

After setting up SGX SDK 2.14 according to docs, we started from the included C++ sample enclave, and gradually replaced sections with our own code. As mentioned earlier, the C++11 standard was chosen for enclave compatibility. Albeit now, at the time of writing (October 21), support for C++14 was added to the SGX SDK [22]. An important step in SGX application development is dividing code into outside/inside enclave execution. `App.cpp` contains the outside code, while `Enclave.cpp` contains the internal code, i.e., the actual DP-GBDT algorithm. The basis of this separation is specified in the `Enclave.edl` file. It defines all function calls that are allowed to cross the enclave border. In our case (see listing 4.1) that means 4 calls in total: Three ecalls that go into the enclave, each of them is called exactly once. Two of them bring the dataset resp. the model parameters into the enclave. The other one does not carry any data and just initiates the training. The only ocall and thus way to get data out of the enclave is the ocall_print_string, which is used to print the final cross validation performance. This is sufficient to see that the algorithm works.

[7] `https://github.com/wxindu/dp-conf-int`

```
untrusted {
  ocall_print_string([in, string] const char *str);
};

trusted {
  ecall_load_dataset_into_enclave([in] struct sgx_dataset *dset);
  ecall_load_modelparams_into_enclave([in] struct sgx_modelparams *mparams);
  ecall_start_gbdt();
};
```

Listing 4.1: Enclave.edl entries

Parsing the dataset(s) requires various I/O functions that are not available inside the enclave. So the parsing (as well as defining the model parameters) is easiest to be taken care of on the outside. Since the interface between outside/inside of the enclave does not allow passing custom datatypes (not even C++ vectors), the dataset and model parameters have to be converted to C-style arrays. Once passed to the inside, they are converted back to C++ data structures. This procedure is illustrated in figure 4.5

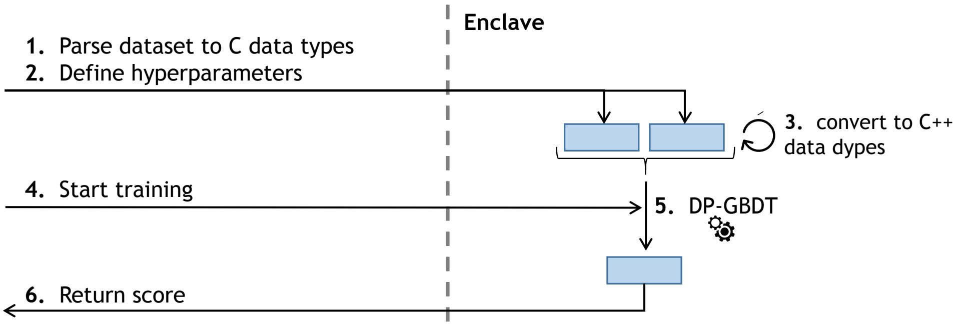
Enclave
1. Parse dataset to C data types
2. Define hyperparameters
3. convert to C++ data dypes
4. Start training
5. DP-GBDT
6. Return score

Figure 4.5: Visualisation of procedures inside/outside of the enclave

The DP-GBDT algorithm itself has to be adjusted everywhere where randomness is involved. Reason being that obtaining randomness inside an enclave is a little different than on the outside. SGX provides a function `sgx_read_rand()` which executes the `RDRAND` instruction to generate hardware-assisted unbiased random numbers [26]. Apart from replacing calls that fetch a random number, several functions, that were previously not much more than a simple standard library call, had to be rewritten from scratch. For example randomly selecting/deleting a subset of rows from a matrix. Of course also the external library code for e.g. the logging had to be removed. Similarly no more multithreading is available. Further, there's an important configuration file `Enclave.config.xml` which specifies settings like available memory size. Modifying this according to your needs is indispensable for lager datasets. For information on changes to the Makefile and other details, consider the code in our repository[8].

[8] https://github.com/loretanr/dp-gbdt/blob/main/TODO.todo