Chapter 2

Background

2.1 Decision trees

Machine learning is generally a two-step process: There is a learning step and prediction step. During the learning phase, a model is built based on the provided training data. During the prediction phase, the previously obtained model is applied to new data in order to forecast the likelihood of a particular outcome. Decision Trees are among the easiest and most popular learning algorithms due to their illustrative nature. They can be used for solving regression and classification problems. This means it can predict both continuous/*numerical* values such as price, salary, etc. and also *discrete/categorical* values, such as gender or nationality. Consider the example in figure 2.1. Using this decision tree, we can categorize an upcoming day according to whether it will be suitable to go rowing. By using the weather outlook, temperature, and wind strength as attributes, the decision tree will return the corresponding classification result (in this case Yes or No).

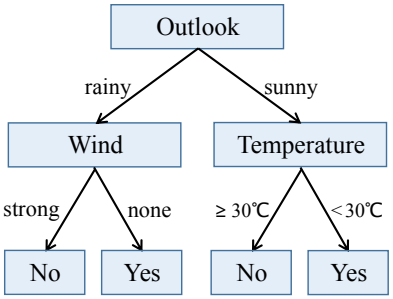

Figure 2.1: Example decision tree

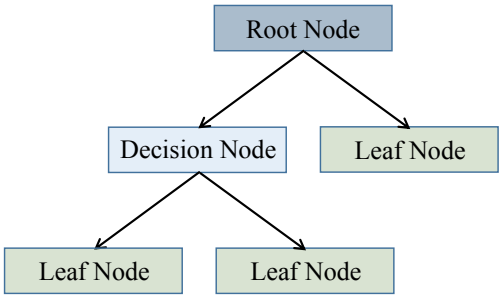

Figure 2.2: Decision tree terminology

Figure 2.2 provides an overview of the terminology related to decision trees. The *root node* is the node that starts the graph. It divides the samples into two parts using the best split that is available on the data. *Splitting* is the process of dividing a node into two (or more) sub-nodes. A *decision/internal node* is formed when a sub-node splits into two (or more) sub-nodes. Nodes where no further splitting takes place are called *leaf/terminal nodes* This is where the predictions of a category or a numerical value are made. In a tree, a node that is divided into sub-nodes is labeled a *parent node*, while its sub-nodes are called *child nodes*.

A decision tree is typically constructed by recursively splitting training samples using the features from the dataset that work best for the specific task at hand. This is done by evaluating certain

metrics, like the the entropy or the *information gain*. There exist many variations of this recursive building algorithm, such as ID3 [56], C4.5 [57] or CART [13].

Definition 2.1 (Information gain [11]) *Let $\boldsymbol{X} = X_1, \ldots, X_n$ be a set of inputs (training samples), where each X_i is of the form $(\boldsymbol{x}, y) = (x_1, \ldots, x_n, y)$. x_j is the value of the j^{th} attribute of $\boldsymbol{x}$, and y is its corresponding label. The information gain for an attribute j is given by:*

$$IG(\boldsymbol{X}, j) = H(\boldsymbol{X}) - H(\boldsymbol{X}|j)$$

Where $H(\cdot)$ is the Shannon entropy. In other words: Information gain or is a statistical property that measures how well a given attribute separates the training examples according to their target classification.

As already indicated, decision trees offer many advantages: They are simple to understand, interpret and visualize. Little to no data preprocessing required. And generally, they are widely applicable as they make no assumptions about the shape of the data. However, there are also some weaknesses:

- Decision-tree tend to create overly complex trees that fail to generalize the data well. This is called *overfitting*.
- Decision tree learners are inclined to create biased trees if some classes dominate [49]. It is thus advised to balance such datasets prior to decision tree fitting.
- Being a greedy algorithm, it does not guarantee to return the globally optimal model.
- It is possible for decision trees to be unstable due to small variations in the data, which can result in completely different trees being produced. Fortunately this effect can be reduced by methods like bagging and boosting.

Bagging and boosting *Ensemble learning* is a concept in which multiple models are trained using the same learning algorithm. Bagging and boosting are both examples of this technique. They combine multiple weak individual learners into one that achieves greater performance than a single learner would. This has been proven to yield better results on many machine learning problems [49]. *Bagging* refers to bootstrapping + aggregation, in which weak learners are trained on a random subset of data sampled with replacement (bootstrapping). Subsequently their predictions are aggregated. Bootstrapping assures independence and diversification as every subset is sampled separately. *Boosting* differs from the aforementioned approach in that it is a *sequential* ensemble method. Let's use decision trees as an illustration. Given a total of n trees, the individual models/trees are added in a sequential way. The second tree is added to improve the performance of the first tree, etc. In the end, the individual models are weighted and combined to a final strong classifier.

2.2 Gradient boosted decision trees

Gradient Boosted Decision Trees (GBDT) algorithms make use of decision trees as the base learner and add up the predictions of several trees. Gradually, new decision trees, that are based on the residual between ground truth and current predictions, are trained and added to the current ensemble. Today, there's a multitude of sophisticated and high-performance GBDT frameworks available, such as LightGBM [41] and XGBoost [18].

In the remainder of this section, we will dive into the formalities of the GBDT algorithm. Let l be a convex loss function and D is a dataset of n instances with d features. That is $X = X_1, .., X_n$,

where $X_i = (\mathbf{x}, y) = (x_1, .., x_d, y)\ \forall i \in [1, n]$. At the t^{th} iteration, GBDT minimizes the following objective function: [64]

$$O(\mathbf{x})^{(t)} = \sum_{i}^{n} \left(g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right) + \Omega(f_t) \tag{2.1}$$

where $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$ is the first order gradient statistics of the loss function. f_t is the decision tree at the t^{th}, $\Omega(f_t) = \frac{1}{2}\lambda \|V\|^2$ is the regularisation term, V is the leaf weight, and λ is the regularisation parameter.

Trees grow from their roots to their maximum depth. Let I_L and I_R be the instances of the left and right subsets after a split. The gain of a split is given by: [46]

$$G(I_L, I_R) = \frac{(\sum_{i \in I_L} g_i)^2}{|I_L| + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{|I_R| + \lambda} \tag{2.2}$$

By traversing all combinations of features and feature values, GBDT finds the split which maximizes the gain. If the current node cannot achieve the splitting requirements (e.g. if it's larger than the maximum tree depth, or all splits have a gain < 0), it becomes a leaf node. We define $I = I_L \cup I_R$. The optimal leaf value is then given by: [46]

$$V(I) = -\eta \frac{\sum_{i \in I} g_i}{|I| + \lambda} \tag{2.3}$$

By applying a shrinkage/learning rate η to the leaf values, we can reduce the influence of each individual tree, allowing future trees to improve the model. The entire GBDT process is shown in algorithm 1:

Algorithm 1: GBDT training process

Input: $X = X_1, \ldots, X_n$: instances, $\mathbf{y} = y_1, \ldots, y_n$: labels
Input: λ: regularisation parameter, d_{max}: maximum depth, η: learning rate
Input: T: total number of trees, l: loss function
Output: An ensemble of trained decision trees.

```
for t = 1 to T do
    Update gradients of all training instances on loss l
    for depth = 1 to d_max do
        forall node in current depth do
            forall split value i do
                G_i ← (Σ_{i∈I_L} g_i)^2 / (|I_L|+λ) + (Σ_{i∈I_R} g_i)^2 / (|I_R|+λ)   ▷ Equation 2.2
            Split node on split value i = arg max_i(G_i)
    forall leaf node i do
        V_i ← −η Σ_{i∈I} g_i / (|I|+λ)                                                ▷ Equation 2.3
```

2.3 Differential privacy

Differential privacy (DP) is a mechanism for provable public sharing of information about a dataset as a whole, while keeping information about the individuals within it private. [29] The

key idea is the following: If a single substitution in the database has a small enough effect on the result of a query to that database, no information about an individual member can be gained. In other words: An attacker is assumed to have almost complete knowledge of the training data, and may only be uncertain about a single point in the training data. Using this, a defender can deny the existence of every single training data point. There are several interesting use cases for differentially private algorithms, such as publishing the results of a survey while ensuring the confidentiality of study participants.

Given a finite data universe $\mathcal{X}$, we define a dataset $D \in \mathcal{X}^n$ as an ordered tuple of n rows $(x_1, .., x_n) \in \mathcal{X}$. We say that two datasets $D, D' \in \mathcal{X}^n$ are *neighboring* if they differ only by a single row. This relationship is denoted by $D \sim D'$.

Definition 2.2 (ε-Differential Privacy [46]) *Let ε be a positive real number and f be a randomised function. Function f is said to provide ε-differential privacy if, for any two neighboring datasets $D \sim D'$ and every output O of function f:*

$$Pr[f(D) \in O] \leq e^{\varepsilon} \cdot Pr[f(D') \in O] \tag{2.4}$$

The parameter ε is also called *privacy budget*, and it controls the privacy guarantee level of function f. A smaller ε represents stronger privacy. In practice, ε is usually chosen <1, such as 0.1 or $\ln 2$.

Sensitivity is a parameter determining how much perturbation is required in the differential privacy mechanisms. We say a function $f\colon \mathcal{X}^n \to \mathbb{R}^m$ has sensitivity Δ if for all neighboring $D, D' \in \mathcal{X}^n$ it holds that $||f(D) - f(D')||_1 \leq \Delta$.

To achieve differential privacy, two popular mechanisms exist: the *Laplace mechanism* and the *exponential mechanism*. Numeric queries can be performed by the former, and non-numeric queries by the latter. In the Laplace mechanism, controlled noise is added to the query result before it is returned to the user. [30] The noise is sampled from the Laplace distribution centered at 0 and scaled by factor b.

Definition 2.3 (Laplace Distribution) *A random variable has a $Lap(\mu, b)$ distribution if its probability density function is:*

$$f(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right) \tag{2.5}$$

Theorem 2.4 (Laplace Mechanism [46]) *Let $f : \mathcal{D} \to \mathcal{R}^d$ be a function. The Laplace mechanism F is defined as:*

$$F(D) = f(D) + Lap(0, \Delta f/\varepsilon) \tag{2.6}$$

Where the noise $Lap(0, \Delta f/\varepsilon)$ is drawn from a Laplace distribution with mean $\mu = 0$ and scale $b = \Delta f/\varepsilon$. Then F provides ε-differential privacy.

Theorem 2.5 (Exponential Mechanism [46]) *Let $u : (\mathcal{D} \times \mathcal{R}) \to \mathbb{R}$ be a utility function, with sensitivity Δu. The exponential mechanism F is defined as:*

$$F(D, u) = \textit{choose } r \in \mathcal{R} \textit{ with probability} \propto \exp\left(\frac{\varepsilon u(D, r)}{2\Delta u}\right) \tag{2.7}$$

Then F provides ε-differential privacy.

As both of the previous mechanisms only offer privacy guarantees for single functions, two composition theorems exist. Using *sequential composition*, a sequence of differentially private computations can be shown to be private. The privacy budgets can be added up for each randomized mechanism.

Theorem 2.6 (Sequential Composition) *Let $f = \{f_1, .., f_m\}$ be a series of functions performed sequentially on a dataset. If f_i provides ε_i-differential privacy, then f provides $\sum_i^m \varepsilon_i$-differential privacy.*

the *parallel composition* theorem can be applied in a case where f is applied to disjoint subsets of a dataset.

The parallel composition corresponds to a case where each f is applied on disjointed subsets of the dataset. The largest privacy budget is then decisive for the final result.

Theorem 2.7 (Parallel Composition) *Let $f = \{f_1, .., f_m\}$ be a series of functions performed separately on disjoint subsets of a dataset. If f_i provides ε_i-differential privacy, then f provides $\{\max(\varepsilon_1, .., \varepsilon_m)\}$-differential privacy.*

2.4 Differentially private gradient boosted decision trees

This section finally combines the different notions and formulas from the previous background sections and puts them to use. Literature ([72, 46]) proposes two adjustments to algorithm 1 for creating differentially private decision trees. (i) Noise has to be added to each leaf node's value in the decision tree. (ii) To choose the attribute and its value upon which a decision node is split, the exponential mechanism must be used. By ranking the different different splits according to their information gain, the the exponential mechanism can choose a split with a proportional probability.

2.4.1 DPBoost

The remainder of this section summarizes DPBoost [46], Li et al.'s work on DP-GBDT, which set the theoretical foundation for this thesis. Li et al.'s goal was to improve model accuracy of previous DP-GBDT work [1, 48, 68]. In their work, several optimizations have been made, such as tighter sensitivity bounds for the leaf nodes and the splitting function, and more effective privacy budget allocation. This results in better accuracy of the algorithm. The following three paragraphs elaborate on the paper's new techniques, before the final algorithm is presented in Algorithm 2.

Gradient-based data filtering Gradient-based data filtering (GDF) is a technique that DPBoost uses in order to obtain a tighter sensitivity bounds. Let $g_l^* = \max_{y_p \in [-1,1]} ||\frac{\partial l(y_p, y)}{\partial y}|_{y=0}||$. At the beginning of each iteration, the instances that have 1-norm gradient larger than g_l^* are filtered out. Only the remaining instances are used to build a new differentially private decision tree in this iteration. Using this, we can bound the sensitivities of G (equation 2.2) and V (equation 2.3): By applying GDF in the training of GBDTs, we get $\Delta G \leq \frac{3\lambda+2}{(\lambda+1)(\lambda+2)} g_l^{*2}$ and $\Delta V \leq \frac{g_l^*}{1+\lambda}$. The derivations can be found in DPBoost [46], chapter 3.

Geometric leaf clipping All trees have the same sensitivities with GDF. During the training process, the gradients tend to decrease with each iteration. Therefore, as the number of iterations increases, one could derive an even tighter sensitivity bound. Based on this, a

geometric leaf clipping mechanism was developed, that can be conducted before the Laplace mechanism is applied to the leaf values. In combination with GDF the following bounds hold: $\Delta V \leq \min(\frac{g_l^*}{1+\lambda}, 2g_l^*(1-\eta)^{t-1})$ and $\Delta G \leq 3g_l^{*2}$, where η is the learning rate, λ is a regularisation parameter and t is the index current tree that is built.

Adaptive privacy budget allocation Specifically, they suggest allocating half of the privacy budget for the leaf nodes (denoted as ε_{leaf}). The remaining budget is equally divided to each depth of the internal nodes (ε_{node} per level). By Theorem 2.7, because the inputs of one depth level are disjoint, the privacy budget consumption of only one depth has to be accounted for. Therefore $\varepsilon_{leaf} + \varepsilon_{node} * d_{max} = \varepsilon_t$ holds.

Algorithm 2: Differentially private GBDT training process [46]

Input: instances $X = X_1, .., X_n$, labels $y = y_1, .., y_n$
Input: λ: regularisation parameter, d_{max}: maximum depth, η: learning rate
Input: T: total number of trees, l: loss function, ε: privacy budget
Output: An ensemble of trained differentially private decision trees.

$\varepsilon_t = \varepsilon$ ▷ Disjoint training subsets → apply Theorem 2.7
for $t = 1$ **to** T **do**
  Update gradients of all training instances on loss l
  $\varepsilon_{leaf} = \frac{\varepsilon_t}{2}$, $\varepsilon_{node} = \frac{\varepsilon_t}{2d_{max}}$
  for depth $= 1$ **to** d_{max} **do**
    forall node **in** current depth **do**
      forall split value i **do**
        $G_i \leftarrow \frac{(\sum_{i \in I_L} g_i)^2}{|I_L|+\lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{|I_R|+\lambda}$ ▷ Equation 2.2
        $P_i \leftarrow \exp(\frac{\varepsilon_{node} \cdot G_i}{2\Delta G})$ ▷ Theorem 2.5
      Split node on split value i, where i is chosen with probability $P_i / \sum_j P_j$
  forall leaf node i **do**
    $V_i \leftarrow \eta \left(-\frac{\sum_{i \in I} g_i}{|I|+\lambda} + Lap(0, \Delta V / \varepsilon_{leaf}) \right)$ ▷ Equation 2.3 and Theorem 2.4

2.5 Intel SGX

TEEs are a set of architectural extensions that provide software with strong security guarantees even when it is exposed to powerful adversaries. Intel Software Guard Extensions (SGX) [37, 20] provides secure execution capabilities to Intel x86 mainframe CPUs. Through SGX, secure and verifiable execution environments known as *enclaves* can be used by applications to execute securely on the processor. SGX is contained inside the CPU package. Hence, its security properties are strongly tied to the processors hardware implementation. All other components of the system (including OS, firmware, and memory hardware) are removed from the Trusted Computing Base (TCB). This is achieved by ensuring that both processor and memory states of an enclave are only exposed to the enclave internal code. Protection from software access and physical attacks on enclave memory is provided by a hardware unit inside the CPU, called the memory encryption engine (MEE). Through transparent decryption and encryption, the MEE ensures that secret data is only stored in plaintext inside the processor. Furthermore, SGX provides remote attestation mechanisms, through which enclaves can prove to remote parties that they have been correctly initialized on a genuine (and therefore presumed secure) Intel processor.

The rest of this section elaborates on several SGX mechanics that are of particular relevance for this thesis. We cover (remote) attestation, data sealing and the use of secure monotonic counters.

Remote attestation Enclaves are no secure software modules by themselves. Rather they are software modules that can execute in a TEE. Due to the fact that enclaves can't be debugged or monitored at runtime, their code has to be verified by users before execution. However, as the untrusted system controls enclave initialization, an adequate mechanism is required. Through *attestation*, a specific software can demonstrate its trustworthiness to an external party in terms of authenticity and integrity. [20] There are two types of enclave attestation supported by SGX: local and remote. During local attestation, a second special enclave must be present on the same platform as the attestant. Remote attestation in turn can be performed with any software running on any external platform. During the remote attestation procedure, the CPU creates a measurement for the attested enclave that uniquely identifies it. A special (Quoting Enclave[1]) then signs this information, providing an attestation signature. In order to achieve secure communication between these two enclaves, local attestation is performed. Once the attestation signature has been received by the remote party, it will forward it to the Intel Attestation Service (IAS), which verifies its validity. This way, the remote party can determine that (i) no tampering occurred (ii) the attested software is running within an genuine hardware-assisted SGX enclave.

Sealing Programs in practice likely contain some secrets that need to be preserved during an event where an enclave is destroyed, a computer crashes etc. SGX therefore enables enclaves to have encrypted and authenticated persistent storage [20]. Each SGX-enabled CPU contains a randomly generated root seal key, that was fused into the hardware during production and is not stored by Intel. From this key, an enclave can derive a sealing key that can be used to encrypt enclave data. The resulting blobs can be passed to the operating system for long-term storage. There are two options when sealing data: (i) Seal to current enclave, which means only an enclave with the same exact same measurements will be able to generate the key to unseal the data. (ii) Seal to the enclave author. This means that the sealed data can be accessed by different versions of the same enclave and also by other enclaves belonging to the same vendor.

Secure monotonic counters TEEs needs protection against rollback attacks. Imagine a scenario where an adversary manipulates persistently stored data by replaying old data or by interleaving output from multiple enclave instances. To mitigate these kind of attacks, Secure Monotonic Counters (SMCs) has been introduced with Intel SGX SDK 1.8 [36]. SMCs leverage non-volatile memory storage locations that survive events like a power loss. There are essentially two operations that can be performed on such a counter after it was initialized, `Read()` and `Incr()`. The counters state and its communication with an enclave are cryptographically secured. These counters can be leveraged for versioning of sealed data: Whenever data is sealed, the corresponding SMC value is included. After e.g. a reboot, the data is restored. By extracting the stored counter value and comparing it to the obtained from rading the SMC. Since both the sealed data and the SMC are cryptographically protected, tampering in form of e.g. rollback attacks can be detected. There are a drawbacks to the current SMC implementations though [51]: (i) Incrementing/writing to the counter values is comparably slow (80-250 ms), which can limit its use in high-throughput applications. (ii) The non-volatile memory used by the counters wear out after approximately one million write operations.

[1] The Quoting Enclave is an Intel-developed secure enclave, which has access to the hardware attestation key required for signing the measurement.

Side-channel susceptibility The small TCB of SGX has its advantages. Scaling performance with the processor's capability, for example. This comes at a cost, however. Due to the simplicity of the design, enclaves must rely on shared, untrusted resources like memory, I/O, etc. As a result, enclaves are susceptible to side-channel attacks. This includes for example timing attacks, cache-attacks, speculative execution attacks, branch prediction attacks, microarchitectural data sampling and software-based fault injection attacks. [55] Intel does acknowledges that "SGX does not defend against this adversary"arguing that " and further states that "preventing side-channel attacks is a matter for the enclave developer" [35]. There are two general types of SGX side-channel mitigation approaches. (i) One approach is to harden the programs by rewriting the code at compile time and randomizing at execution time. Consequently, monitoring side-channels is much more complicated for the adversary. These measures however are often susceptible to more versatile future attack variants [67, 16, 52]. (ii) An alternative approach is to manually eliminate secret-dependent data paths entirely. eliminate secret-dependent code and data paths altogether. This method can be very effective at times, but there are also many pitfalls for a developer to fall into. [52, 63, 16]. Either way, SGX developers are left with the major task of evaluating the security implications and figuring out practical methods to mitigate the side-channels hazards.

2.6 Side channels

This section offers a very brief introduction to side-channels. We start with an illustrative example. Subsequently we define the notion of digital side channels, which will be important for later chapters.

Consider Algorithm 3 as an example. It depicts the binary version of the classical square-and-multiply algorithm, which can be used to perform exponentiation. Variations of it can be found in old implementations of RSA encryption and decryption. Actually, in efficient RSA implementations, all of the modular multiplications and squaring operations are performed using an algorithm of it's own, called Montgomery Multiplication [53]. Which was itself target to several side-channel attacks [61]. But to keep things simple, we only focus on the actual square-and-multiply function. The problem lies in the branch on line 4. If the branch is taken, an additional modular multiplication is performed. Through this side-channel, the value of the secret d can be inferred one bit at a time [14].

Algorithm 3: Binary Square-and-Multiply

Input: M, d, N
Output: $M^d \mod N$, where d is an n-bit number $d = (d_0, .., d_{n-1})$

```
1 S = M
2 for i from 1 to n − 1 do
3     S = S ∗ S mod N
4     if d_i = 1 then
5         S = S ∗ M mod N
6 return S
```

The code in Algorithm 3 has in fact been exploited with use of a variety of other side-channels, such as power [42], cache [71], branch prediction [2] and even sound [32]. Accordingly many defense techniques have been proposed.

Digital side-channels For the remainder of this thesis we a notion to classify the variety of possible side-channels. We therefore adopt the term of *digital side channels*, as proposed in the work of Rane et al. [58]. Digital side-channels are side-channels that carry information over discrete bits. An adversarial observer can detect them at the level of either program state or the ISA (instruction set architecture). Therefore, address traces, cache usage, and data size are all examples of digital side-channels. In contrast to elements like power draw, electromagnetic radiation and heat, which are not part of this group.