Noname manuscript No.
(will be inserted by the editor)

TREANT: Training Evasion-Aware Decision Trees

Stefano Calzavara · Claudio Lucchese · Gabriele Tolomei · Seyum Assefa Abebe · Salvatore Orlando

the date of receipt and acceptance should be inserted later

Abstract Despite its success and popularity, machine learning is now recognized as vulnerable to *evasion attacks*, i.e., carefully crafted perturbations of test inputs designed to force prediction errors. In this paper we focus on evasion attacks against decision tree ensembles, which are among the most successful predictive models for dealing with non-perceptual problems. Even though they are powerful and interpretable, decision tree ensembles have received only limited attention by the security and machine learning communities so far, leading to a sub-optimal state of the art for adversarial learning techniques. We thus propose TREANT, a novel decision tree learning algorithm that, on the basis of a formal threat model, minimizes an evasion-aware loss function at each step of the tree construction. TREANT is based on two key technical ingredients: *robust splitting* and *attack invariance*, which jointly guarantee the soundness of the learning process. Experimental results on publicly available datasets show that TREANT is able to generate decision tree ensembles that are at the same time accurate and nearly insensitive to evasion attacks, outperforming state-of-the-art adversarial learning techniques.

1 Introduction

Machine Learning (ML) is increasingly used in several applications and different contexts. When ML is leveraged to ensure system security, such as in spam filtering and intrusion detection, everybody acknowledges the need of training ML models resilient to adversarial manipulations (Huang et al., 2011; Biggio and Roli, 2018). Yet the same applies to other critical application scenarios in which ML is now employed, where adversaries may cause severe system malfunctioning or faults. For example, consider an ML model which is used by a bank to grant loans to inquiring customers: a malicious customer may try to fool the model into illicitly qualifying him for a loan. Unfortunately, traditional ML algorithms proved vulnerable to a

Stefano Calzavara, Claudio Lucchese, Seyum Assefa Abebe and Salvatore Orlando
Università Ca' Foscari Venezia, Italy

Gabriele Tolomei
Sapienza Università di Roma, Italy

wide range of attacks, and in particular to *evasion attacks*, i.e., carefully crafted perturbations of test inputs designed to force prediction errors (Biggio et al., 2013; Nguyen et al., 2015; Papernot et al., 2016a; Moosavi-Dezfooli et al., 2016).

To date, research on evasion attacks has mostly focused on linear classifiers (Lowd and Meek, 2005; Biggio et al., 2011) and, more recently, on deep neural networks (Szegedy et al., 2014; Goodfellow et al., 2015). Whereas deep learning obtained remarkable and revolutionary results on many perceptual problems, such as those related to computer vision and natural language understanding, *decision trees ensembles* are nowadays one of the best methods for dealing with non-perceptual problems, and are one of the most commonly used techniques in Kaggle competitions (Chollet, 2017). Decision trees are also considered *interpretable* compared to other models (Tolomei et al., 2017), yielding predictions which are human-understandable in terms of syntactic checks over domain features, which is particularly appealing in the security setting. Unfortunately, despite their success, decision tree ensembles have received only limited attention by the security and machine learning communities so far, leading to a sub-optimal state of the art for adversarial learning techniques (see Section 2.3).

In this paper, we thus propose TREANT,[1] a novel learning algorithm designed to build decision trees which are resilient against evasion attacks at test time. Based on a formal threat model, TREANT optimizes an evasion-aware loss function at each step of the tree construction (Madry et al., 2018). This is particularly challenging to enforce correctly, considered the greedy nature of traditional decision tree learning (Hunt et al., 1966). In particular, TREANT has to ensure that the local greedy choices performed upon tree construction are not short-sighted with respect to the capabilities of the attacker, who has the advantage of choosing the best attack strategy based on the fully built tree. TREANT is based on the combination of two key technical ingredients: a *robust splitting* strategy for decision tree nodes, which reliably takes into account at training time the attacker's capability of perturbing instances at test time, and an *attack invariance* property, which preserves the correctness of the greedy construction by generating and propagating constraints along the decision tree, so as to discard splitting choices which might be vulnerable to attacks.

We finally deploy our learning algorithm within a traditional random forest framework (Breiman, 2001) and show its predictive power on real-world datasets. Notice that, although there have been various proposals that tried to improve robustness against evasion attacks by using ensemble methods (Hershkop and Stolfo, 2005; Perdisci et al., 2006; Tran et al., 2008; Biggio et al., 2010), it was shown that ensembles of weak models are not necessarily strong (He et al., 2017). We avoid this shortcoming by employing TREANT to train an ensemble of decision trees which are individually resilient to evasion attempts.

1.1 Roadmap

To show how TREANT improves over the state of the art, we proceed as follows:

1. We first review decision trees and decision tree ensembles, presenting a critique of existing adversarial learning techniques for such models (Section 2).

[1] The name comes from the role playing game "Dungeons & Dragons", where it identifies giant tree-like creatures.

2. We introduce our formal threat model, discussing an exhaustive white-box attack generation method, which allows for an accurate evaluation of the performance of decision trees under attack and proves scalable enough for our experimental analysis (Section 3).
3. We present TREANT, the first tree learning algorithm which greedily, yet soundly, minimizes an evasion-aware loss function upon tree construction (Section 4).
4. We experimentally show that TREANT outperforms existing adversarial learning techniques on four publicly available datasets (Section 5).

Our analysis shows that TREANT is able to build decision tree ensembles that are at the same time accurate and nearly insensitive to evasion attacks, providing a significant improvement over the state of the art.

2 Background and Related Work

2.1 Supervised Learning

Let $\mathcal{X} \subseteq \mathbb{R}^d$ be a d-dimensional vector space of real-valued *features*. An *instance* $\boldsymbol{x} \in \mathcal{X}$ is a d-dimensional feature vector $(x_1, x_2, \ldots, x_d)$ representing an object in the vector space.[2] Each instance $\boldsymbol{x} \in \mathcal{X}$ is assigned a label $y \in \mathcal{Y}$ by some unknown *target* function $g : \mathcal{X} \mapsto \mathcal{Y}$. Starting from a set of hypotheses $\mathcal{H}$, the goal of a *supervised learning* algorithm is to find the function $\hat{h} \in \mathcal{H}$ that best approximates the target g. This is practically achieved through empirical risk minimization (Vapnik, 1992); given a sample of correctly labeled instances $\mathcal{D} = \{(\boldsymbol{x}_1, g(\boldsymbol{x}_1)), \ldots, (\boldsymbol{x}_n, g(\boldsymbol{x}_n))\}$ known as the *training set*, the empirical risk is defined by a loss function $\mathcal{L} : \mathcal{H} \times (\mathcal{X} \times \mathcal{Y})^n \mapsto \mathbb{R}^+$ measuring the cost of erroneous predictions, i.e., the cost of predicting $\hat{h}(\boldsymbol{x}_i)$ instead of the true label $g(\boldsymbol{x}_i)$, for all $(\boldsymbol{x}_i, g(\boldsymbol{x}_i)) \in \mathcal{D}$. Supervised learning thus amounts to finding $\hat{h} = \operatorname{argmin}_{h \in \mathcal{H}} \mathcal{L}(h, \mathcal{D})$.

The loss $\mathcal{L}$ is typically obtained by aggregating an instance-level loss $\ell : \mathcal{Y} \times \mathcal{Y} \mapsto \mathbb{R}^+$. In this work, we assume $\mathcal{L}(h, \mathcal{D}) = \sum_{(\boldsymbol{x}, y) \in \mathcal{D}} \ell(h(\boldsymbol{x}), y)$.

2.2 Decision Trees and Decision Tree Ensembles

A powerful set of hypotheses $\mathcal{H}$ is the set of the *decision trees* (Breiman et al., 1984; Quinlan, 1986). We focus on binary decision trees, whose internal nodes perform thresholding over feature values. Such trees can be inductively defined as follows: a decision tree t is either a leaf $\lambda(\hat{y})$ for some label $\hat{y} \in \mathcal{Y}$ or a non-leaf node $\sigma(f, v, t_l, t_r)$, where $f \in [1, d]$ identifies a feature, $v \in \mathbb{R}$ is the threshold for the feature f and t_l, t_r are decision trees (left and right respectively). At test time, an instance $\boldsymbol{x}$ traverses the tree t until it reaches a leaf $\lambda(\hat{y})$, which returns the *prediction* $\hat{y}$, denoted by $t(\boldsymbol{x}) = \hat{y}$. Specifically, for each traversed tree node $\sigma(f, v, t_l, t_r)$, $\boldsymbol{x}$ falls into the left tree t_l if $x_f \leq v$, and into the right tree t_r otherwise. We just write λ or σ to refer to some leaf or node of the decision tree when its actual content is irrelevant. The problem of learning an optimal decision tree is

[2] For simplicity, we only consider numerical features over $\mathbb{R}$. However, our framework can be readily generalized to other use cases, e.g., categorical or ordinal features, which we support in our implementation and experiments.

Algorithm 1 BuildTree

Input: training data $\mathcal{D}$
$\hat{y} \leftarrow \text{argmin}_y \mathcal{L}(\lambda(y), \mathcal{D})$
$\sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r)), \mathcal{D}_l, \mathcal{D}_r \leftarrow \text{BestSplit}(\mathcal{D})$
if $\mathcal{L}(\sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r)), \mathcal{D}) < \mathcal{L}(\lambda(\hat{y}), \mathcal{D})$ **then**
    $t_l \leftarrow \text{BuildTree}(\mathcal{D}_l)$
    $t_r \leftarrow \text{BuildTree}(\mathcal{D}_r)$
    return $\sigma(f, v, t_l, t_r)$
else
    return $\lambda(\hat{y})$
end if

Algorithm 2 BestSplit

Input: training data $\mathcal{D}$
▷ Build a set of candidate tree nodes $\mathcal{N}$ via an exhaustive search over f and v
$\mathcal{N} \leftarrow \{\sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r)) \mid \hat{y}_l, \hat{y}_r = \text{argmin}_{y_l, y_r} \mathcal{L}(\sigma(f, v, \lambda(y_l), \lambda(y_r)), \mathcal{D})\}$
▷ Select the candidate node $\hat{t} \in \mathcal{N}$ which minimizes the loss $\mathcal{L}$ on the training data $\mathcal{D}$
$\hat{t} = \text{argmin}_{t \in \mathcal{N}} \mathcal{L}(t, \mathcal{D}) = \sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r))$
▷ Split the training data $\mathcal{D}$ based on the best candidate node $\hat{t} = \sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r))$
$\mathcal{D}_l \leftarrow \{(\boldsymbol{x}, y) \in \mathcal{D} \mid x_f \leq v\}$
$\mathcal{D}_r \leftarrow \mathcal{D} \setminus \mathcal{D}_l$
return $\hat{t}, \mathcal{D}_l, \mathcal{D}_r$

known to be NP-complete (Hyafil and Rivest, 1976; Murthy, 1998); as such, a top-down greedy approach is usually adopted (Hunt et al., 1966), as shown in Algorithm 1.

The function BuildTree takes as input a dataset $\mathcal{D}$ and initially computes the label $\hat{y}$ which minimizes the loss on $\mathcal{D}$ for a decision tree composed of just a single leaf; for instance, when the loss is the Sum of Squared Errors (SSE), such label just amounts to the mean of the labels in $\mathcal{D}$. The function then checks if it is possible to grow the tree to further reduce the loss by calling a *splitting* function BestSplit (Algorithm 2), which attempts to replace the leaf $\lambda(\hat{y})$ with a new sub-tree $\sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r))$. This sub-tree is greedily identified by choosing f and v from an exhaustive exploration of the search space consisting of all the possible features and thresholds, and with the predictions $\hat{y}_l$ and $\hat{y}_r$ chosen so as to minimize the global loss on $\mathcal{D}$. If it is possible to reduce the loss on $\mathcal{D}$ by growing the new sub-tree, the tree construction is recursively performed over the subsets $\mathcal{D}_l = \{(\boldsymbol{x}, y) \in \mathcal{D} \mid x_f \leq v\}$ and $\mathcal{D}_r = \mathcal{D} \setminus \mathcal{D}_l$, otherwise the original leaf $\lambda(\hat{y})$ is returned. Real-world implementations of the algorithm typically use multiple stopping criteria to prevent overfitting, e.g., by bounding the tree depth, or by requiring a minimum number of instances in the recursive calls.

Random Forest (RF) and Gradient Boosting Decision Trees (GBDT) are popular ensemble learning methods for decision trees (Breiman, 2001; Friedman, 2001). RFs are obtained by independently training a set of trees $\mathcal{T}$, which are combined into the *ensemble predictor* $\hat{h}$, e.g., by using majority voting to assign the class label. Each $t_i \in \mathcal{T}$ is typically built by using bagging and per-node feature sampling over the training set. In GBDTs, instead, each tree approximates a gradient descent step along the direction of loss minimization. Both methods are very effective, where RF is able to train models with low variance, while GDBTs are models of high accuracy yet possibly prone to overfitting.

2.3 Related Work

Adversarial learning, which investigates the safe adoption of ML in adversarial settings (Huang et al., 2011), is a research field that has been consistently increasing of importance in the last few years. In this paper we deal with *evasion attacks*, a research sub-field of adversarial learning, where deployed ML models are targeted by attackers who craft adversarial examples that resemble normal data instances, but force wrong predictions. Most of the work in this field regards classifiers, in particular binary ones. The attacker starts from a positive instance that is classified correctly by the deployed ML model and is interested in introducing minimal perturbations on the instance to modify the prediction from positive to negative, thus "evading" the classifier (Nelson et al., 2010; Biggio et al., 2013, 2014; Srndic and Laskov, 2014; Kantchelian et al., 2016; Carlini and Wagner, 2017; Dang et al., 2017; Goodfellow et al., 2015; Zhang et al.). Contrary to robust machine learning (Tyler, 2008) where some form of probabilistic random noise is assumed, in the adversarial setting even a single perturbation which is able to fool the classifier is assumed to be adopted by the attacker with 100% probability. To prevent evasion attacks, different techniques have been proposed for different models, including support vector machines (Biggio et al., 2011; Xiao et al., 2015), deep neural networks (Gu and Rigazio, 2015; Goodfellow et al., 2015; Papernot et al., 2016b), and decision tree ensembles (Kantchelian et al., 2016; Chen et al., 2019). Unfortunately, the state of the art for decision tree ensembles is far from satisfactory.

The first adversarial learning technique for decision tree ensembles is due to Kantchelian et al. (2016) and is called *adversarial boosting*. It is an empirical data augmentation technique, borrowing from the *adversarial training* approach (Szegedy et al., 2014), where a number of evading instances are included among the training data to make the learned model aware of the attacks and, thereby, possibly more resilient to them. Specifically, at each boosting round, the training set is extended by crafting a set of possible perturbations for each original instance and by picking the one with the smallest margin, i.e., the largest misprediction risk, for the model trained so far. Adding perturbed instances to the training set forces the learning algorithm to minimize the *average* error over both the original instances and the chosen sample of evading ones, but this does not provide clear performance guarantees under attack. This is both because evading instances exploited at training time might not be representative of test-time attacks, and because optimizing the average case might not defend against the *worst-case* attack. Indeed, the experiments in Section 5 show that the performance of ensembles trained via adversarial boosting can be severely downgraded by evasion attacks.

The second adversarial learning technique for decision tree ensembles was proposed in a very recent work by Chen *et al.*, who introduced the first tree learning algorithm embedding the attacker directly in the optimization problem solved upon tree construction (Chen et al., 2019). The key idea of their approach, called *robust trees*, is to redefine the splitting strategy of the training examples at a tree node. They first identify the so-called *unknown* instances of $\mathcal{D}$, which may fall in either in $\mathcal{D}_l$ or in $\mathcal{D}_r$, depending on adversarial perturbations. The authors thus claim that the optimal tree construction strategy would need to account for an exponential number of attack configurations over these unknown instances. To tame such algorithmic complexity, they propose a sub-optimal heuristic approach based on four "representative" attack cases. Though the key idea of this algorithm is

certainly interesting and shares some similarities with our own proposal, it also suffers from significant shortcomings. First, representative attack cases are not such anymore when the attacker is aware of the defense mechanism, and they are not anyway sufficient to subsume the spectrum of possible attacks: our algorithm takes into account all the possible attack cases, while being efficient enough for practical adoption. Moreover, the approach in (Chen et al., 2019) does not implement safeguards against the incremental greedy nature of decision tree learning: there is no guarantee that, once the best splitting has been identified, the attacker cannot adapt his strategy to achieve better results on the full tree. Indeed, the experimental evaluation in Section 5 shows that it is very easy to evade the trained models, which turn out to be even more fragile than those trained through adversarial boosting in some cases.

3 Threat Model

The possibility to craft adversarial examples was popularized by Szegedy et al. (2014) in the image classification domain: their seminal work showed that it is possible to introduce minimal perturbations into an image so as to modify the prediction of its class by a deep neural network.

3.1 Loss Under Attack and Adversarial Learning

At an abstract level, we can see the attacker A as a function mapping each instance to a set of possible perturbations, which might be able to evade the ML model. Depending on the specific application scenario, not every attack is plausible, e.g., A cannot force some perturbations or behaves surreptitiously to avoid detection. For instance, in the typical image classification scenario, A is usually assumed to introduce just slight modifications that are perceptually undetectable to humans. This simple similarity constraint between the original instance $\boldsymbol{x}$ and its perturbed variant $\boldsymbol{z}$ is well captured by a distance (Goodfellow et al., 2015), e.g., one could have $A(\boldsymbol{x}) = \{\boldsymbol{z} \mid \|\boldsymbol{z} - \boldsymbol{x}\|_\infty \leq \varepsilon\}$.

Similarly, assuming that the attacker can run independent attacks on every instance of a given dataset $\mathcal{D}$, we can define $A(\mathcal{D})$ as the set of the datasets $\mathcal{D}'$ obtained by replacing each $(\boldsymbol{x}, y) \in \mathcal{D}$ with any $(\boldsymbol{z}, y)$ such that $\boldsymbol{z} \in A(\boldsymbol{x})$.

The hardness of crafting successful evasion attacks defines the *robustness* of a given ML model at test time. The goal of learning a robust model is therefore to minimize the harm an attacker may cause via perturbations. This learning goal was formalized as a min-max problem by Madry et al. (2018):

$$\hat{h} = \operatorname*{argmin}_{h \in \mathcal{H}} \underbrace{\max_{\mathcal{D}' \in A(\mathcal{D})} \mathcal{L}(h, \mathcal{D}')}_{\mathcal{L}^A(h, \mathcal{D})} . \tag{1}$$

The inner maximization problem models the attacker A replacing all the given instances with an adversarial example aimed at maximizing the loss. We call *loss under attack*, noted $\mathcal{L}^A(h, \mathcal{D})$, the solution to the inner maximization problem. The outer minimization resorts to the empirical risk minimization principle, aiming to find the hypothesis that minimizes the loss under attack on the training set.

3.2 Attacker Model

Distance-based constraints for defining the attacker's capabilities are very flexible for perceptual problems and proved amenable for heuristic algorithms for solving the inner maximization problem of Equation 1 (Madry et al., 2018). However, they cannot be easily generalized to other realistic application scenarios, e.g., where perturbations are not symmetric, where the attacker may not be able to alter some of the features, or where categorical attributes are present. To overcome such limitations, we model the attacker A as a pair (R, K), where R is a set of *rewriting rules*, defining how instances can be corrupted, and $K \in \mathbb{R}^+$ is a *budget*, limiting the amount of alteration the attacker can apply to each instance. Each rule $r \in R$ has form:

$$[a, b] \xrightarrow{f}_k [\delta_l, \delta_u],$$

where $[a, b]$ and $[\delta_l, \delta_u]$ are intervals on $\mathbb{R} \cup \{-\infty, +\infty\}$, with the former defining the *precondition* for the application of the rule and the latter defining the *magnitude* of the perturbation enabled by the rule; $f \in [1, d]$ is the index of the feature to corrupt; and $k \in \mathbb{R}^+$ is the *cost* of the rule. The semantics of the rewriting rule can be explained as follows: if an instance $\boldsymbol{x}$ satisfies the condition $x_f \in [a, b]$, then the attacker can corrupt it by adding any $v \in [\delta_l, \delta_u]$ to x_f and spending k from the available budget. Note that v can possibly be negative, leading to a subtraction. The attacker can corrupt each instance by using as many rewriting rules as desired in whatever order, up to budget exhaustion.

According to this attacker model, we define $A(\boldsymbol{x})$, the set of the attacks against an instance $\boldsymbol{x}$, as follows.

Definition 1 (Attacks) Given an instance $\boldsymbol{x}$ and an attacker $A = (R, K)$, we let $A(\boldsymbol{x})$ be the set of the *attacks* that can be obtained from $\boldsymbol{x}$, i.e., the set of the instances $\boldsymbol{z}$ such that there exists a sequence of rewriting rules $r_1, \ldots, r_n \in R$ and a sequence of instances $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_n$ where:

1. $\boldsymbol{x}_0 = \boldsymbol{x}$ and $\boldsymbol{x}_n = \boldsymbol{z}$;
2. for all $i \in [1, n]$, the instance $\boldsymbol{x}_{i-1}$ can be corrupted into the instance $\boldsymbol{x}_i$ by using the rewriting rule r_i;
3. the sum of the costs of $r_1, \ldots, r_n$ is not greater than K.

Notice that $\boldsymbol{x} \in A(\boldsymbol{x})$ for any A by picking an empty sequence of rewriting rules.

We highlight that this rule-based attacker model includes novel attack capabilities like asymmetric perturbations, easily generalizes to categorical variables, and still covers or approximates standard distanced-based models. For instance, L_0-distance attacker models where the attacker can corrupt at will a limited number of features can be easily represented (Kantchelian et al., 2016). The use of a budget is convenient to fine-tune the power of the attacker and enables the adoption of standard evaluation techniques for ML models under attack, like *security evaluation curves* (Biggio and Roli, 2018).

Example 1 (L_0-Distance) The L_0-distance captures localized perturbations with arbitrary magnitude. Specifically, given an instance $\boldsymbol{x} \in \mathcal{X} \subseteq \mathbb{R}^d$ and a possible perturbation $\boldsymbol{z}$, we have that $\|\boldsymbol{z} - \boldsymbol{x}\|_0 = |\{f \in [1, d] \mid z_f \neq x_f\}|$, and thus the L_0-distance simply counts the dimensions of $\boldsymbol{x}$ that were actually perturbed.

In our framework, we can model this by means of an attacker $A = (R, K)$, where the budget K stands for the largest L_0-distance allowed on adversarial perturbations and R includes, for all features f, a rewriting rule of the form:

$$[-\infty, +\infty] \xrightarrow{f}_1 [-\infty, +\infty].$$

It is easy to show that for all $\boldsymbol{z}$ we have $\boldsymbol{z} \in A(\boldsymbol{x})$ if and only if $\|\boldsymbol{z} - \boldsymbol{x}\|_0 \leq K$. In particular, the largest perturbation is obtained from the original $\boldsymbol{x}$ by applying exactly K distinct rules, each perturbing a different dimension.

Example 2 (L_1-Distance) The L_1-distance also captures localized perturbations, but constrains their magnitude. Specifically, given an instance $\boldsymbol{x} \in \mathcal{X} \subseteq \mathbb{R}^d$ and a possible perturbation $\boldsymbol{z}$, we have that $\|\boldsymbol{z} - \boldsymbol{x}\|_1 = \sum_f |z_f - x_f|$.

In our framework, we can model this by means of an attacker $A = (R, K)$, where the budget K stands for the largest L_1-distance allowed on adversarial perturbations and R includes, for all features f, a rewriting rule of the form:

$$[-\infty, +\infty] \xrightarrow{f}_\varepsilon [-\varepsilon, +\varepsilon],$$

where $\varepsilon \in \mathbb{R}^+$ models a maximum *discrete step* of perturbation (and its cost).

It is easy to show that the set $A(\boldsymbol{x})$ can approximate $\{\boldsymbol{z} \mid \|\boldsymbol{z} - \boldsymbol{x}\|_1 \leq K\}$ with arbitrarily large accuracy by choosing appropriately small values of ε. Note that the largest perturbation is obtained from the original $\boldsymbol{x}$ by applying exactly $\lfloor K/\varepsilon \rfloor$ rules, always choosing the maximum or minimum magnitude $\pm\varepsilon$.

Example 3 (L_∞-Distance) The L_∞-distance encourages uniformly spread perturbations with small magnitude. Specifically, given an instance $\boldsymbol{x} \in \mathcal{X} \subseteq \mathbb{R}^d$ and a possible perturbation $\boldsymbol{z}$, we have that $\|\boldsymbol{z} - \boldsymbol{x}\|_\infty = \max_f |z_f - x_f|$.

We observe that this form of non-localized perturbations is not currently supported by our threat model, since, once a rewriting rule is defined for a given feature, it can always be (locally) applied up to budget exhaustion. However, a straightforward solution to this issue would be to transform our current global budget into a set of *per-feature* budgets $\{K_1, \ldots, K_d\}$. Then, if K is the largest L_∞-distance allowed on adversarial perturbations, one could let $K_i = K$ for all i and just reuse the rewriting rules defined for the case of the L_1-distance. We do not implement this extension of the model for the sake of simplicity.

3.3 Attack Generation

Computing the loss under attack $\mathcal{L}^A$ is useful to evaluate the resilience of ML models to evasion attacks at test time; yet this might be intractable, since it assumes the ability to identify the most effective attack for all the test instances. This issue is thus typically dealt with by using a heuristic attack generation algorithm, e.g., the fast gradient sign method (Goodfellow et al., 2015) or any of its variants, to craft adversarial examples which empirically work well. However, our focus on decision trees and the adoption of a rule-based attacker model enables an exhaustive attack generation strategy for the test set which, though computationally expensive, proves scalable enough for our experimental analysis and allows the

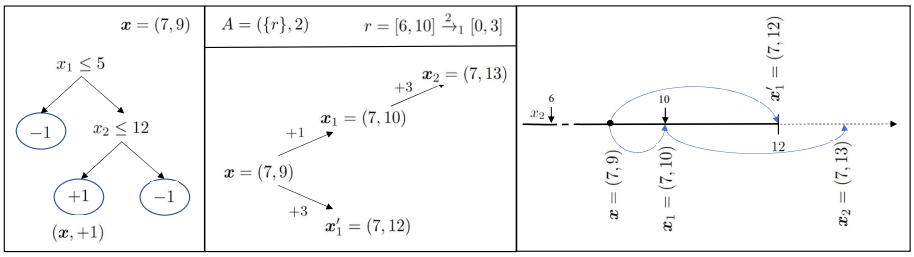

Fig. 1 A decision tree and an instance $\boldsymbol{x}$ that can be attacked by perturbing its feature 2 (by adding any value in interval $[0, 3]$). Note that since the maximum budget of A is 2, and the cost of applying the rule r is 1, the rule can be applied only twice provided that the precondition holds.

actual identification of the most effective attacks. This enables the most accurate security assessment in terms of the actual value of $\mathcal{L}^A$.

We consider a *white-box* attacker model, where the attacker has the complete knowledge of the trained decision tree ensemble. We thus assume that the attacker exploits the knowledge of the structure of the trees in the targeted ensemble and, most importantly, of the features and thresholds which are actually used in the prediction process.

Note that a decision tree ensemble induces a finite partition of the input vector space $\mathcal{X}$, defined by the features and thresholds used in the internal nodes of the trees in the ensemble, where instances falling in the same element of the partition share the same prediction. This partition of the vector space makes it possible to significantly reduce the set of attacks that are relevant to compute $\mathcal{L}^A$ by considering at most one *representative* attack for each element of the partition (Calzavara et al., 2019). Once this is done, one can feed all the attack representatives to the tree ensemble and identify the one that maximizes the loss.

For the sake of simplicity, we just sketch the algorithm that generates the attack representatives. For any given instance $\boldsymbol{x}$, we first identify the set of applicable rules: if a rule targets the feature f with magnitude $[\delta_l, \delta_r]$, the interval of possible perturbations $[x_f + \delta_l, x_f + \delta_u]$ is split into the sub-intervals induced by (i) the ensemble's thresholds relative to feature f, since this might change the prediction of the tree ensemble, and (ii) the extremes of the pre-conditions of a rewriting rule operating on feature f, since this might enable further perturbations which will eventually lead to prediction changes. We then generate a single attack for each of the identified sub-intervals by applying maximal perturbations therein and recursively apply the algorithm up to budget exhaustion. Finally, we return just the attacks which actually crossed some threshold of the tree ensemble, since only those could lead to changes in predictions.

Example 4 (Attack Generation) Consider the instance $\boldsymbol{x} = (7, 9)$ with label $+1$ and the decision tree in Figure 1, which classifies the instance correctly. Pick then the attacker $A = (\{r\}, 2)$, where r is a rewriting rule of cost 1 which allows the

corruption of the feature 2 by adding any value in $[0, 3]$, provided that the feature value is in the interval $[6, 10]$. In our formalism, this is represented as follows:

$$r = [6, 10] \xrightarrow{2}_1 [0, 3].$$

Only three values of the feature 2 are relevant in our setting to generate representative attacks: besides 12 (the threshold used by the decision tree), which actually partitions the second dimension into the intervals $(-\infty, 12]$ and $(12, +\infty)$, also 6 (the lower bound of the pre-condition of rule r) and 10 (the upper bound of the pre-condition of rule r). We include these bounds because rule r might be applied again, as long as the perturbations fall within the interval $[6, 10]$, and this might be useful to eventually cross a threshold of the decision tree.

Our algorithm thus applies r multiple times to perturb the second feature of $\boldsymbol{x} = (7, 9)$: in particular, the value 9 can initially be perturbed into any value from $[9, 12]$ after an application of the rule. Perturbations in this range can only cross one of the previously identified thresholds, i.e., 10. This induces a partitioning of $[9, 12]$ in the sub-intervals $[9, 10]$ and $(10, 12]$. The first attack $\boldsymbol{x}_1 = (7, 10)$ does not lead to a change of the prediction outcome of the decision tree, yet moved towards the decision threshold and can still be corrupted by rule r. The alternative attack $\boldsymbol{x}'_1 = (7, 12)$ also does not lead to any prediction change and cannot be corrupted any further due to the pre-condition of rule r. However, the attacker can target the second feature of $\boldsymbol{x}_1 = (7, 10)$ to corrupt it into any value from $[10, 13]$. Perturbations in this range can cross the decision threshold 12, inducing the sub-intervals $[10, 12]$ and $(12, 13]$. In particular, the attack $\boldsymbol{x}_2 = (7, 13)$ is generated by the algorithm and it is the only returned attack, since it is representative of all the attacks causing the instance $\boldsymbol{x}$ to fall into the partition $(12, +\infty)$ of the second dimension of the feature space.

4 TREANT: Key Ideas & Design

In this section, we present a novel decision tree learning algorithm that, by minimizing the loss under attack $\mathcal{L}^A$ at training time, enforces resilience to evasion attacks at test time. We call TREANT the proposed algorithm.

4.1 Overview

Compared to Algorithm 1, TREANT replaces the BESTSPLIT function by revising: (*i*) the computation of the predictions on the new leaves, (*ii*) the selection of the best split and (*iii*) the dataset partition along the recursion.

Before discussing the technical details, we build on the toy example in Figure 2 to illustrate the non-trivial issues arising when optimizing $\mathcal{L}^A$. Figure 2.(a) shows a dataset $\mathcal{D}$ for which we assume the attacker $A = (\{r\}, 1)$, where r is a rewriting rule of cost 1 which allows the corruption of the feature p by adding any value in the interval $[-1, +1]$.

Assuming SSE is used as the underlying loss function $\mathcal{L}$, the decision stump initially generated by Algorithm 1 is shown in Figure 2.(b) along with the result of the splitting. Note that while the loss $\mathcal{L} = 2$ is small,[3] the loss under attack $\mathcal{L}^A = 5$

[3] $\mathcal{L}(t, \mathcal{D}) = (-2 + 1)^2 + (-1 + 1)^2 + (-1 - 0)^2 + 4 \cdot (2 - 2)^2 = 2$.

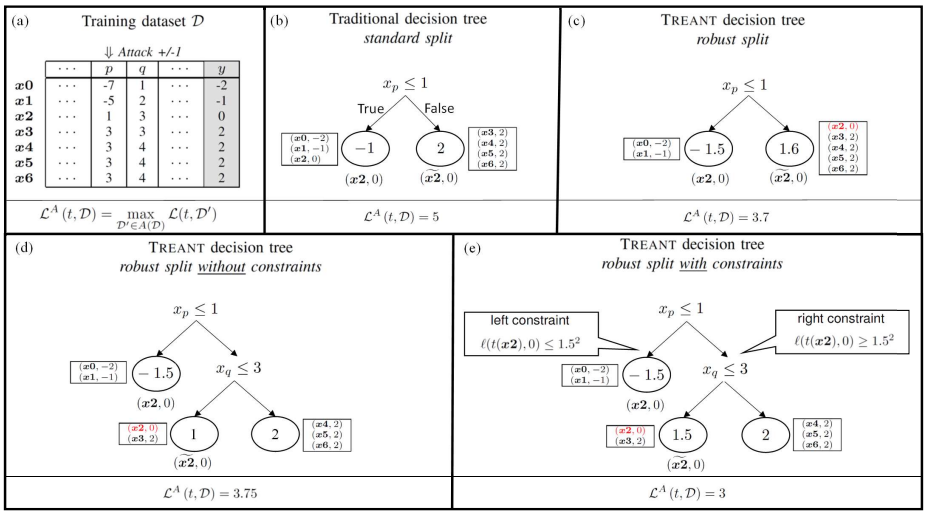

Fig. 2 Overview of the TREANT construction and its key challenges.

is much larger.[4] This is because the attacker may alter $\boldsymbol{x2}$ into a perturbed instance $\widetilde{\boldsymbol{x2}}$ so as to reverse the outcome of the test $x_p \leq 1$, i.e., the original instance $\boldsymbol{x2}$ falls into the left leaf of the stump, but the perturbed instance $\widetilde{\boldsymbol{x2}}$ falls into the right leaf. The first issue of Algorithm 1 is thus that the estimated loss $\mathcal{L}$ on the training set, computed when building the decision stump, is smaller than the loss under attack $\mathcal{L}^A$ we would like to minimize. We solve this issue by designing a novel *robust splitting* strategy to identify the best split of $\mathcal{D}$, which directly minimizes $\mathcal{L}^A$ when computing the leaves predictions and leads to the generation of a tree that is more robust to attacks. In particular, the decision stump learnt by using our robust splitting strategy is shown in Figure 2.(c), where the leaves predictions have been found by assuming that $\boldsymbol{x2}$ actually falls into the right leaf (according to the best attack strategy). For this new decision stump, the best move for the attacker is still to corrupt $\boldsymbol{x2}$, but the resulting $\mathcal{L}^A = 3.7$ is much smaller than that of the previous stump.[5] The figure also shows the outcome of the robust splitting.

However, a second significant issue arises when the decision stump is recursively grown into a full decision tree. Suppose to further split the right leaf of Figure 2.(c), therefore considering only the instances falling therein, including the instance $\boldsymbol{x2}$ put there by the robust splitting. We would find that the best split is given by $x_q \leq 3$, where the feature q cannot be modified by the attacker. The resulting tree is shown in Figure 2.(d). Note however that, by creating the new sub-tree, new attacking opportunities show up, because the attacker now finds more convenient to just leave $\boldsymbol{x2}$ unaltered and let it fall directly into the left child of the root. As a consequence, by adding the new sub-tree, we observe an increased loss under attack $\mathcal{L}^A = 3.75$.[6] This second issue can be solved by ensuring that any new sub-tree does not create new attacking opportunities that generate a larger loss. We

[4] $\mathcal{L}^A(t,\mathcal{D}) = (-2+1)^2 + (-1+1)^2 + (2-0)^2 + 4\cdot(2-2)^2 = 5$.
[5] $\mathcal{L}^A(t,\mathcal{D}) = (-2+1.5)^2 + (-1+1.5)^2 + (0-1.6)^2 + 4\cdot(2-1.6)^2 = 3.7$.
[6] $\mathcal{L}^A(t,\mathcal{D}) = (-2+1.5)^2 + (-1+1.5)^2 + (0-1.5)^2 + (2-1)^2 + 3\cdot(2-2)^2 = 3.75$.

Table 1 Notation Summary

Symbol	Meaning
$\mathcal{D}$	Training dataset
$\mathcal{D}^\lambda$	Local projection of $\mathcal{D}$ on the leaf λ
$A(\boldsymbol{x})$	Set of all the attacks A can generate from $\boldsymbol{x}$
$A(\mathcal{D})$	Set of all the attacks A can generate from $\mathcal{D}$
$\lambda(\hat{y})$	Leaf node with prediction $\hat{y}$
$\sigma(f, v, t_l, t_r)$	Node testing $x_f \leq v$ and having sub-trees t_l, t_r
$\mathcal{D}_l(f, v, A)$	Left elements of ternary partitioning on (f, v)
$\mathcal{D}_r(f, v, A)$	Right elements of ternary partitioning on (f, v)
$\mathcal{D}_u(f, v, A)$	Unknown elements of ternary partitioning on (f, v)
$\mathcal{D}_L(\hat{t}, A)$	Left elements of robust splitting on $\hat{t}$
$\mathcal{D}_R(\hat{t}, A)$	Right elements of robust splitting on $\hat{t}$
$\mathcal{C}_L(\hat{t}, A)$	Set of constraints for the left child of $\hat{t}$
$\mathcal{C}_R(\hat{t}, A)$	Set of constraints for the right child of $\hat{t}$

call this property *attack invariance*. The proposed algorithm grows the sub-tree on the right leaf by carefully adjusting its predictions as shown in Figure 2.(e), still decreasing the loss under attack to $\mathcal{L}^A = 3$ with respect to the tree in Figure 2.(c).[7] This is enforced by including constraints along the tree construction, as shown in the figure.

To sum up, the key technical ingredients of TREANT are:

1. *Robust splitting*: given a candidate feature f and threshold v, the robust splitting strategy evaluates the quality of the corresponding node split on the basis of a *ternary* partitioning of the instances falling into the node. It identifies those instances for which the outcome of the node predicate $x_f \leq v$ depends on the attacker's moves, and those that cannot be affected by the attacker, thus always traversing the left or the right branch of the new node. In particular, the $\mathcal{L}^A$ minimization problem is reformulated on the basis of left, right and unknown instances, i.e., instances which might fall either left or right depending on the attacker. Finally, the recursion on the left and right child of the node is performed by separating the instances in a binary partition based on the effects of the most harmful attack (Section 4.2).
2. *Attack invariance*: a security property requiring that the addition of a new sub-tree does not allow the attacker to find better attack strategies that increase $\mathcal{L}^A$. Attack invariance is achieved by imposing an appropriate set of constraints upon node splitting. New constraints are generated for each of the attacked instances present in the split node and are propagated to the child nodes upon recursion (Section 4.3).

The pseudo-code of the algorithm is given in Section 4.4. To assist the reader, the notation used in the present section is summarized in Table 1.

4.2 Robust Splitting

We present our novel *robust splitting* strategy that grows the current tree t by replacing a leaf λ with a new sub-tree so as to minimize the loss under attack $\mathcal{L}^A$.

[7] $\mathcal{L}^A(t, \mathcal{D}) = (-2 + 1.5)^2 + (-1 + 1.5)^2 + (0 - 1.5)^2 + (2 - 1.5)^2 + 3 \cdot (2 - 2)^2 = 3$.

For the sake of clarity, we discuss it as if the splitting was employed on the root node of a new tree, i.e., to learn the decision stump that provides the best loss reduction on the full input dataset $\mathcal{D}$. The next subsection discusses the application of the proposed strategy during the recursive steps of the tree-growing process.

Aiming at greedily optimizing the *min-max* problem in Equation 1, we have to find the best decision stump $\hat{t} = \sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r))$ such that:

$$\begin{aligned}\hat{t} &= \operatorname*{argmin}_t \mathcal{L}^A(t, \mathcal{D}) = \\ &= \operatorname*{argmin}_t \max_{\mathcal{D}' \in A(\mathcal{D})} \mathcal{L}(t, \mathcal{D}') = \\ &= \operatorname*{argmin}_t \sum_{(\boldsymbol{x}, y) \in \mathcal{D}} \max_{\boldsymbol{z} \in A(\boldsymbol{x})} \ell(t(\boldsymbol{z}), y).\end{aligned}$$

However, the equation shows that this is not trivial, because the loss incurred by an instance $(\boldsymbol{x}, y)$ may depend on the attacks it is possibly subject to. Similarly to (Chen et al., 2019), we thus define a ternary partitioning of the training dataset as follows.

Definition 2 (Ternary Partitioning) For a feature f, a threshold v and an attacker A, the *ternary partitioning* of the dataset $\mathcal{D} = \mathcal{D}_l(f, v, A) \cup \mathcal{D}_r(f, v, A) \cup \mathcal{D}_u(f, v, A)$ is defined by:

$$\begin{aligned}\mathcal{D}_l(f, v, A) &= \{(\boldsymbol{x}, y) \in \mathcal{D} \mid \forall \boldsymbol{z} \in A(\boldsymbol{x}) : z_f \leq v\} \\ \mathcal{D}_r(f, v, A) &= \{(\boldsymbol{x}, y) \in \mathcal{D} \mid \forall \boldsymbol{z} \in A(\boldsymbol{x}) : z_f > v\} \\ \mathcal{D}_u(f, v, A) &= (\mathcal{D} \setminus \mathcal{D}_l(f, v, A)) \setminus \mathcal{D}_r(f, v, A).\end{aligned}$$

In words, $\mathcal{D}_l(f, v, A)$ includes those instances $(\boldsymbol{x}, y)$ falling into the left branch regardless of the attack, hence the attacker has no gain in perturbing x_f. A symmetric reasoning applies to $\mathcal{D}_r(f, v, A)$, containing those instances which fall into the right branch for all the possible attacks. The instances that the attacker may actually want to target are those falling into $\mathcal{D}_u(f, v, A)$, thus aiming at the largest loss. By altering those instances, the attacker may force each $(\boldsymbol{x}, y) \in \mathcal{D}_u(f, v, A)$ to fall into the left branch with a loss of $\ell(\hat{y}_l, y)$, or into the right branch, with a loss of $\ell(\hat{y}_r, y)$.

Example 5 (Ternary Partitioning) The test node $x_p \leq 1$ and the attacker considered in Figure 2.(c) determine the following ternary partitioning of $\mathcal{D}$:

- $\mathcal{D}_l(p, 1, A) = \{(\boldsymbol{x0}, -2), (\boldsymbol{x1}, -1)\}$
- $\mathcal{D}_r(p, 1, A) = \{(\boldsymbol{x3}, 2), (\boldsymbol{x4}, 2), (\boldsymbol{x5}, 2), (\boldsymbol{x6}, 2)\}$
- $\mathcal{D}_u(p, 1, A) = \{(\boldsymbol{x2}, 0)\}$

In other words, the instance $\boldsymbol{x2}$ is the only instance for which the branch taken at test time is unknown, as it depends on the attacker A.

By construction, given (f, v), the loss $\mathcal{L}^A$ can be affected by the presence of the attacker A only for the instances in $\mathcal{D}_u(f, v, A)$, while for all the remaining instances it holds that $\mathcal{L}^A = \mathcal{L}$. Since the attacker may force each instance of $\mathcal{D}_u(f, v, A)$ to fall into either the left or the right branch, the authors of (Chen et al., 2019) acknowledge a combinatorial explosion in the computation of $\mathcal{L}^A$. Rather than evaluating all the possible configurations, they thus propose a heuristic approach

evaluating four "representative" attack cases: *i)* no attack, *ii)* all the unknown instances are forced in the left child, *iii)* all the unknown instances are forced in the right child, and *iv)* all the unknown instances are swapped by the attacker, i.e., they are forced in the left/right child when they would normally fall in the right/left child. Then, the loss $\mathcal{L}$ is evaluated for these four split configurations and the maximum value is used to estimate $\mathcal{L}^A$, so as to find the best stump $\hat{t}$ to grow. Note that $\mathcal{L}$ is computed as in a standard decision tree learning algorithm. Unfortunately, this heuristic strategy does not offer soundness guarantees, because the above four configurations leave potentially harmful attacks out of sight and do not induce an upper-bound of $\mathcal{L}^A$.

To avoid this soundness issue, while keeping the tree construction tractable, we pursue a *numerical optimization* as follows. For a given (f, v), we highlight that finding the best attack configuration and finding the best left/right leaves predictions $\hat{y}_l, \hat{y}_r$ are two inter-dependent problems, yet the strategy adopted in (Chen et al., 2019) is to first evaluate a few different attack configurations, and then find the leaves predictions. We instead solve these two problems simultaneously via a formulation of the min-max problem that, fixed (f, v), is expressed solely in terms of $\hat{y}_l, \hat{y}_r$:

$$(\hat{y}_l, \hat{y}_r) = \operatorname*{argmin}_{y_l, y_r} \mathcal{L}^A(\sigma(f, v, \lambda(y_l), \lambda(y_r)), \mathcal{D}), \tag{2}$$

where $\mathcal{L}^A$ is decomposed via the ternary partitioning as:

$$\begin{aligned} \mathcal{L}^A(\sigma(f, v, \lambda(y_l), \lambda(y_r)), \mathcal{D}) \quad &= \\ = \quad \mathcal{L}(\lambda(y_l), \mathcal{D}_l(f, v, A)) &+ \mathcal{L}(\lambda(y_r), \mathcal{D}_r(f, v, A)) + \\ + \sum_{(\boldsymbol{x}, y) \in \mathcal{D}_u(f, v, A)} &\max\{\ell(y_l, y), \ell(y_r, y)\}. \end{aligned}$$

Observe that if the instance-level loss ℓ is convex, then $\mathcal{L}^A$ is also convex[8] and it can be efficiently optimized numerically. Convexity is indeed a property enjoyed by many loss functions such as SSE (for regression) and Log-Loss (for classification). This allows one to overcome the exploration of the exponential number of attack configurations, still finding the optimal solution (up to numerical approximation).

Given the best predictions $\hat{y}_l, \hat{y}_r$, we can finally produce a binary split of $\mathcal{D}$ (as in Algorithm 1). To do this, we split the instances by applying the best adversarial moves, i.e., by assuming that every $(\boldsymbol{x}, y) \in \mathcal{D}_u(f, v, A)$ is pushed into the left or right child so as to generate the largest loss. If the two children induce the same loss, then we assume the instance is not attacked.

Definition 3 (Robust Splitting) For a sub-tree to be grown $\hat{t} = \sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r))$ and an attacker A, the *robust split* of $\mathcal{D} = \mathcal{D}_L(\hat{t}, A) \cup \mathcal{D}_R(\hat{t}, A)$ is defined as follows:

- $\mathcal{D}_L(\hat{t}, A)$ contains all the instances of $\mathcal{D}_l(f, v, A)$ and $\mathcal{D}_R(\hat{t}, A)$ contains all the instances of $\mathcal{D}_r(f, v, A)$;
- for each $(\boldsymbol{x}, y) \in \mathcal{D}_u(f, v, A)$, the following rules apply:
  - if $\ell(\hat{y}_l, y) > \ell(\hat{y}_r, y)$, then $(\boldsymbol{x}, y)$ goes to $\mathcal{D}_L(\hat{t}, A)$;
  - if $\ell(\hat{y}_l, y) < \ell(\hat{y}_r, y)$, then $(\boldsymbol{x}, y)$ goes to $\mathcal{D}_R(\hat{t}, A)$;

[8] The pointwise maximum and the sum of convex functions preserve convexity.

– if $\ell(\hat{y}_l, y) = \ell(\hat{y}_r, y)$, then $(\boldsymbol{x}, y)$ goes to $\mathcal{D}_L(\hat{t}, A)$ if $x_f \leq v$ and to $\mathcal{D}_R(\hat{t}, A)$ otherwise.

Example 6 (Robust Splitting) Once identified $\hat{y}_l$ and $\hat{y}_r$ for the decision stump $\hat{t} = (p, 1, \lambda(-1.5), \lambda(1.6))$ in Figure 2.(c), the datasets obtained for the leaves by robust splitting are:

– $\mathcal{D}_L(\hat{t}, A) = \{(\boldsymbol{x0}, -2), (\boldsymbol{x1}, -1)\}$
– $\mathcal{D}_R(\hat{t}, A) = \{(\boldsymbol{x2}, 0), (\boldsymbol{x3}, 2), (\boldsymbol{x4}, 2), (\boldsymbol{x5}, 2), (\boldsymbol{x6}, 2)\}$

Notice that, unlike a standard decision tree learning algorithm, the right partition contains the instance $\boldsymbol{x2}$ due to the presence of the attacker, even though such instance normally satisfies the root node test.

To summarize, the ternary partitioning allows $\mathcal{L}^A$ to be optimized for a given (f, v) and dataset $\mathcal{D}$, hence it can be used to find the best tree-growing step by an exhaustive search over f and v. Once this is done, the robust splitting allows the dataset $\mathcal{D}$ to be partitioned in order to feed the algorithm recursion on the left and right children of the newly created sub-tree. Ultimately, the goal of the proposed construction is solving the min-max problem of Equation 1 for a single tree-growing step and pushing the attacked instances into the partition induced by the most harmful attack.

4.3 Attack Invariance

The optimization strategy described in Section 4.2 needs some additional refinement to provide a sound optimization of $\mathcal{L}^A$ on the full dataset $\mathcal{D}$. When growing a new sub-tree at a leaf λ, we denote with $\mathcal{D}^\lambda$ the *local projection* of the full dataset at λ, i.e., the subset of the instances in $\mathcal{D}$ falling in λ along the tree construction by applying the robust splitting strategy. The key observation now is that the robust splitting operates by assuming that the attacker behaves *greedily*, i.e., by locally maximizing the generated loss, but as new nodes are added to the tree, new attack opportunities arise and different traversal paths towards different leaves may become more fruitful to the attacker. If this is the case, the robust splitting becomes unrepresentative of the possible attacker's moves and any learning decision made on the basis of such splitting turns out to be unsound, i.e., with no guarantee of minimizing $\mathcal{L}^A$. Notice that this is a major design problem of the algorithm proposed in (Chen et al., 2019), and experimental evidence shows how the attacker can easily craft adversarial examples in some cases (see Section 5).

In the end, the computation of the best split for a given leaf λ cannot be done just based on the local projection $\mathcal{D}^\lambda$, unless additional guarantees are provided. We thus enforce a security property called *attack invariance*, which ensures that the tree construction steps preserve the correctness of the greedy assumptions made on the attacker's behavior. Given a decision tree t and an instance $(\boldsymbol{x}, y) \in \mathcal{D}$, we let $\Lambda^A(t, (\boldsymbol{x}, y))$ stand for the set of leaves of t which are reachable by some attack $\boldsymbol{z} \in A(\boldsymbol{x})$ that generates the largest loss among $A(\boldsymbol{x})$.

Attack invariance requires that the tree construction steps preserves Λ^A, in that the attacker has no advantage in changing the attack strategy which was optimal up to the previous step, thus recovering the soundness of the greedy construction. We define attack invariance during tree construction as follows.

Definition 4 (Attack Invariance) Let t be a decision tree and let t' be the decision tree obtained by replacing a leaf λ of t with the new sub-tree $\sigma(f, v, \lambda_l, \lambda_r)$. We say that t' satisfies *attack invariance* for the dataset $\mathcal{D}$ and the attacker A iff:

$$\forall (\boldsymbol{x}, y) \in \mathcal{D}^{\lambda} : \Lambda^{A}(t', (\boldsymbol{x}, y)) \cap \{\lambda_l, \lambda_r\} \neq \emptyset.$$

The above definition states that, after growing a new sub-tree from λ, the set of the best options for the attacker against the instances in λ must include the newly created leaves, so that the path originally leading to λ still represents the most effective attack strategy against those instances.

Example 7 (Attack Invariance) Let t be the decision tree of Figure 2.(c). Figure 2.(d) shows an example where adding a new sub-tree to t leads to a decision tree t' which breaks the attack invariance property. Indeed, we have $\Lambda^{A}(t', (\boldsymbol{x2}, 0)) = \{\lambda(-1.5)\}$, which contains neither $\lambda(1)$, nor $\lambda(2)$. Notice that the best attack strategy has indeed changed with respect to t, as leaving $\boldsymbol{x2}$ unaltered now produces a larger loss (2.25) than the originally strongest attack (1.0).

We enforce attack invariance by introducing a set of *constraints* into the optimization problem of Equation 2. Suppose that the new sub-tree $\sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r))$ replaces the leaf λ and that an instance $(\boldsymbol{x}, y) \in \mathcal{D}^{\lambda}$ is placed in the right child by robust splitting, because one of its corruptions traverses the threshold v and $\ell(\hat{y}_r, y) \geq \ell(\hat{y}_l, y)$. Then, attack invariance is granted if, whenever the leaves $\lambda(\hat{y}_l)$ and $\lambda(\hat{y}_r)$ are later replaced by sub-trees t_l and t_r, there exists an attack $\boldsymbol{z} \in A(\boldsymbol{x})$ that falls into a leaf of t_r generating a loss larger than (or equal to) the loss of any other attack falling in t_l. We enforce such constraint during the recursive tree building process as follows. The requirement $\ell(\hat{y}_r, y) \geq \ell(\hat{y}_l, y)$ is transformed in the pair of constraints $\ell(t_r(\boldsymbol{x}), y) \geq \gamma$ and $\ell(t_l(\boldsymbol{x}), y) \leq \gamma$, where $\gamma = \min\{\ell(\hat{y}_r, y), \ell(\hat{y}_l, y)\}$. These two constraints are respectively propagated into the recursion on the right and left children. As long as any sub-tree t_r replacing $\lambda(\hat{y}_r)$ satisfies the constraint $\ell(t_r(\boldsymbol{x}), y) \geq \gamma$ and any sub-tree t_l replacing $\lambda(\hat{y}_l)$ satisfies the constraint $\ell(t_l(\boldsymbol{x}), y) \leq \gamma$, the attacker has no advantage in changing the original attack strategy, hence attack invariance is enforced.

To implement this mechanism, each leaf λ is extended with a set of constraints, which is initially empty for the root of the tree. When λ is then split upon tree growing, the constraints therein are included in the optimization problem of Equation 2 to determine the best predictions $\hat{y}_l, \hat{y}_r$ for the new leaves. These constraints are then propagated to the new leaves and new constraints are generated for them based on the following definition, which formalizes the previous intuition.

Definition 5 (Constraints Propagation and Generation) Let λ be a leaf to be replaced with sub-tree $\hat{t} = \sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r))$ and let $\mathcal{C}$ be its set of constraints. The sets of constraints $\mathcal{C}_L(\hat{t}, A)$ and $\mathcal{C}_R(\hat{t}, A)$ for the two new leaves are defined as follows:[9]

- if $\ell(t(\boldsymbol{x}), y) \lesseqgtr \gamma \in \mathcal{C}$ and there exists $\boldsymbol{z} \in A(\boldsymbol{x})$ such that $z_f \leq v$, then $\ell(t_l(\boldsymbol{x}), y) \lesseqgtr \gamma$ is added to $\mathcal{C}_L(\hat{t}, A)$;
- if $\ell(t(\boldsymbol{x}), y) \lesseqgtr \gamma \in \mathcal{C}$ and there exists $\boldsymbol{z} \in A(\boldsymbol{x})$ such that $z_f > v$, then $\ell(t_r(\boldsymbol{x}), y) \lesseqgtr \gamma$ is added to $\mathcal{C}_R(\hat{t}, A)$;

[9] We use the symbol $\lesseqgtr$ to stand for either $\leq$ or $\geq$ when the distinction is unimportant.

- if $(\boldsymbol{x}, y) \in \mathcal{D}_u^{\lambda}(f, v, A) \cap \mathcal{D}_L^{\lambda}(\hat{t}, A)$, then $\ell(t_l(\boldsymbol{x}), y) \geq \ell(\hat{y}_r, y)$ is added to $\mathcal{C}_L(\hat{t}, A)$ and $\ell(t_r(\boldsymbol{x}), y) \leq \ell(\hat{y}_r, y)$ is added to $\mathcal{C}_R(\hat{t}, A)$;
- if $(\boldsymbol{x}, y) \in \mathcal{D}_u^{\lambda}(f, v, A) \cap \mathcal{D}_R^{\lambda}(\hat{t}, A)$, then $\ell(t_l(\boldsymbol{x}), y) \leq \ell(\hat{y}_l, y)$ is added to $\mathcal{C}_L(\hat{t}, A)$ and $\ell(t_r(\boldsymbol{x}), y) \geq \ell(\hat{y}_l, y)$ is added to $\mathcal{C}_R(\hat{t}, A)$.

Example 8 (Enforcing Constraints) The tree in Fig. 2.(e) is generated by enforcing a constraint on the loss of $\boldsymbol{x2}$. After splitting the root, the constraint $\ell(t_r(\boldsymbol{x2}), 0) \geq \ell(\hat{y}_l, 0)$ is generated for the right leaf of the tree in Fig. 2.(c), where $\ell(\hat{y}_l, 0) = (-1.5 - 0)^2 = 2.25$. The solution of the *constrained* optimization problem on the right child of the tree in Fig. 2.(c) finally grows two new leaves, generating the tree in Fig. 2.(e). The difference from the tree in Fig. 2.(d) is that the prediction on the left leaf of the right child of the root has been enforced to satisfy the required constraint. For this tree, the attacker has no gain in changing attack strategy over the previous step of the tree construction, shown in Figure 2.(c).

More formally, after growing the tree in Fig. 2.(c) with suitable constraints we obtain the tree t' in Fig. 2.(e), where the leaf $\lambda(1.6)$ has been substituted with a decision stump with the two new leaves $\{\lambda(1.5), \lambda(2)\}$. This gives $\Lambda^A(t', (\boldsymbol{x2}, 0)) = \{\lambda(-1.5), \lambda(1.5)\}$, where $\Lambda^A(t', (\boldsymbol{x2}, 0)) \cap \{\lambda(1.5), \lambda(2)\} = \{\lambda(1.5)\} \neq \emptyset$, thus satisfying the attack invariance property.

Note that constraints grant attack invariance at the cost of reducing the space of the possible solutions for tree-growing. Nevertheless, in the experimental section we show that this property does not prevent the construction of robust decision trees that are also accurate in absence of attacks.

4.4 Tree Learning Algorithm

Our TREANT construction is summarized in Algorithm 3. The core of the logic is in the call to the TREANTSPLIT function (line 3), which takes as input a dataset $\mathcal{D}$, an attacker A and a set of constraints $\mathcal{C}$ initially empty, and implements the construction detailed along the present section. The construction terminates when it is not possible to further reduce $\mathcal{L}^A$ (line 4).

Function TREANTSPLIT is summarized in Algorithm 4. Specifically, the function returns the sub-tree minimizing the loss under attack $\mathcal{L}^A$ on $\mathcal{D}$ subject to the constraints $\mathcal{C}$, based on the ternary partitioning (lines 2-3). It then splits $\mathcal{D}$ by means of the robust splitting strategy (lines 4-5) and returns new sets of constraints (lines 6-7), which are used to recursively build the left and right sub-trees. The optimization problem (line 2) can be numerically solved via the `scipy` implementation of the SLSQP (Sequential Least SQuares Programming) method, which allows the minimization of a function subject to inequality constraints, like the constraint set $\mathcal{C}$ generated/propagated by TREANT during tree growing.

There is an important point worth discussing about the implementation of the algorithm. As careful readers may have noticed, the TREANTSPLIT function splits each leaf λ by relying on the set of attacks $A(\boldsymbol{x})$ for all instances $(\boldsymbol{x}, y) \in \mathcal{D}^{\lambda}$. Though one could theoretically pre-compute all the possible attacks against the instances in $\mathcal{D}$, this would be very inefficient both in time and space, given the potentially huge number of instances and attacks. Our implementation, instead, incrementally computes a *sufficient* subset of $A(\boldsymbol{x})$ along the tree construction. This makes the construction computationally feasible by exploiting the observation that

Algorithm 3 TREANT

Input: training data $\mathcal{D}$, attacker A, constraints $\mathcal{C}$
$\hat{y} \leftarrow \text{argmin}_y \mathcal{L}^A(\lambda(y), \mathcal{D})$ subject to $\mathcal{C}$
$\sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r)), \mathcal{D}_l, \mathcal{D}_r, \mathcal{C}_l, \mathcal{C}_r \leftarrow \textsc{TreantSplit}(\mathcal{D}, A, \mathcal{C})$
if $\mathcal{L}^A(\sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r)), \mathcal{D}) < \mathcal{L}^A(\lambda(\hat{y}), \mathcal{D})$ **then**
    $t_l \leftarrow \textsc{Treant}(\mathcal{D}_l, A, \mathcal{C}_l)$
    $t_r \leftarrow \textsc{Treant}(\mathcal{D}_r, A, \mathcal{C}_r)$
    return $\sigma(f, v, t_l, t_r)$
else
    return $\lambda(\hat{y})$
end if

Algorithm 4 TREANTSPLIT

Input: training data $\mathcal{D}$, attacker A, constraints $\mathcal{C}$
▷ Build a set of candidate tree nodes $\mathcal{N}$ using Ternary Partitioning to optimize $\mathcal{L}^A$
$\mathcal{N} \leftarrow \{\sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r)) \mid \hat{y}_l, \hat{y}_r = \text{argmin}_{y_l, y_r} \mathcal{L}^A(\sigma(f, v, \lambda(y_l), \lambda(y_r)), \mathcal{D})$ subject to $\mathcal{C}\}$
▷ Select the candidate node $\hat{t} \in \mathcal{N}$ which minimizes the loss $\mathcal{L}^A$ on the training data $\mathcal{D}$
$\hat{t} = \text{argmin}_{t \in \mathcal{N}} \mathcal{L}^A(t, \mathcal{D}) = \sigma(f, v, \lambda(\hat{y}_l), \lambda(\hat{y}_r))$
▷ Robust Splitting (see Definition 3)
$\mathcal{D}_l \leftarrow \mathcal{D}_L(\hat{t}, A)$
$\mathcal{D}_r \leftarrow \mathcal{D}_R(\hat{t}, A)$
▷ Constraint Propagation and Generation (see Definition 5)
$\mathcal{C}_l \leftarrow \mathcal{C}_L(\hat{t}, A)$
$\mathcal{C}_r \leftarrow \mathcal{C}_R(\hat{t}, A)$
return $\hat{t}, \mathcal{D}_l, \mathcal{D}_r, \mathcal{C}_l, \mathcal{C}_r$

the ternary partitioning used for node splitting only requires the identification of a single attack against the feature which is tested in the node predicate, hence the computation of the full set of attacks is not actually needed.

More specifically, each instance $(\boldsymbol{x}, y)$ is enriched with a cost annotation k, denoted by $(\boldsymbol{x}, y)^k$, initially set to 0 on the root. Such annotation keeps track of the cost of the adversarial manipulations performed to push $(\boldsymbol{x}, y)$ into λ during the tree construction. When splitting the leaf λ on (f, v), the algorithm generates only the attacks against the feature f and assumes that k was already spent from the attacker's budget to further reduce the number of possible attacks. When the instance $(\boldsymbol{x}, y)^k$ is pushed into the left or right partition of $\mathcal{D}^\lambda$ by robust splitting, the label k is updated to $k+k'$, where k' is the *minimum cost* the attacker must spend to achieve the desired node outcome. The same idea is applied when propagating constraints, which are also associated with specific instances $(\boldsymbol{x}, y)$ for which the computation of $A(\boldsymbol{x})$ is required.

Observe that this implementation assumes that only the cost of adversarial manipulations is relevant, not their magnitude, which is still sound when none of the corrupted features is tested multiple times on the same path of the tree. We enforce such restriction during the tree construction, which further regularizes the growing of the tree. Since we are eventually interested in decision tree ensembles, this does not impact on the performance of whole trained models.

4.5 Complexity Analysis

In a standard decision tree construction algorithm, the cost of splitting a node λ is $O(d \cdot |\mathcal{D}^\lambda|)$, since this requires a scan of the instances in λ to find the best feature

and threshold for tree growing (Mehta et al., 1996). Similarly, TREANT splits each decision tree node by means of an exhaustive search over all possible features and thresholds. The key difference lies in the node splitting procedure (Algorithm 4). In particular, given a fixed feature and threshold, TREANT pays an extra cost over traditional tree constructions coming from two factors: the ternary partitioning and the corresponding $\mathcal{L}^A$ optimization problem (see Eq. 2). As we anticipated, the optimization problem can be solved by the SLSQP method, which has cubic complexity in the number of variables (Kraft, 1994). Our problem only has two variables, corresponding to the predictions on the left and right leaf respectively.

Regarding the computational complexity of the ternary partitioning, we analyse below the cost of deciding whether or not an instance $\boldsymbol{x}$ belongs to $\mathcal{D}_u^\lambda(f, v, A)$. This has to be paid for every instance in the dataset and it is a multiplicative factor with respect to the standard decision tree growing algorithm.

Proposition 1 *Given a split candidate pair (f, v), an instance $(\boldsymbol{x}, y) \in \mathcal{D}$ and an attacker $A = (R, K)$, the computational complexity of deciding whether $\boldsymbol{x}$ belongs to $\mathcal{D}_u^\lambda(f, v, A)$ is $O\left(\left(\sqrt{2}|R| + 1\right)^{\frac{2K}{k^*}}\right)$, where k^* is the cost of the cheapest rule in R.*

To assess whether $\boldsymbol{x}$ can be attacked, we are interested in finding (if it exists) a sequence of perturbations with minimum cost which leads to crossing the threshold v. Being of minimum cost, we can assume that each rule r in such chain is maximally exploited so as to perturb the instance $\boldsymbol{x}$ to the extremes of the interval $[x_f + \delta_l, x_f + \delta_u]$ or to the ends of a precondition interval enabling some other rule in R. Therefore, in the worst case, each rule $r \in R$ can modify $\boldsymbol{x}$ into $2 + 2|R|$ different ways, and this process is repeated up to budget exhaustion, i.e., at most K/k^* times where k^* is the cost of the cheapest rule in R. We finally get a total cost of $O\left((|R|(2 + 2|R|))^{\frac{K}{k^*}}\right)$, or equivalently $O\left(\left(\sqrt{2}|R| + 1\right)^{\frac{2K}{k^*}}\right)$.

Note that this bound is not tight as not all rules are always applicable, k^* might be far from the cost of other rules, and it might not always be possible to generate $2 + 2|R|$ perturbed instances. In fact, we may not need to enumerate all the possible perturbations.

Below we consider a setting where the set of rules R encodes a L_1-distance attacker, which is one of the most commonly used models in literature. As discussed in Section 3.2, the above attacker can be encoded with one single rule per feature, which leads to a significantly lower computational complexity.

Proposition 2 *Given a split candidate pair (f, v), a dataset $\mathcal{D}$ and an attacker A such that there is only one rule per feature of the form $r : [-\infty, +\infty] \xrightarrow{f}_{\varepsilon} [-\varepsilon, +\varepsilon]$, the computational complexity of deciding whether $\boldsymbol{x}$ belongs to $\mathcal{D}_u^\lambda(f, v, A)$ is $O(1)$.*

Supposing $x_f \leq v$ (a similar reasoning holds for $x_f > v$), an attacked instance $\boldsymbol{z}$ with $z_f > v$ can be crafted with minimum cost by applying the rule r to $\boldsymbol{x}$ a total of $\lfloor (v - x_f)/\varepsilon \rfloor + 1$ times, where each application bears a cost equal to ε. If the attacker's budget K is sufficient to cover such cost, then the instance $\boldsymbol{x}$ belongs to $\mathcal{D}_u^\lambda(f, v, A)$, and this check can be performed in constant time $O(1)$. By repeating this for every instance in $\mathcal{D}$, we have a total cost of $O(|\mathcal{D}|)$.

The above strategy can be easily generalized to L_0-distance attackers and to every other rule set R where only one rule per feature is given. Specifically, an analogous yet slightly more involved argument proves the following result.

Proposition 3 *Given a split candidate pair (f, v), a dataset $\mathcal{D}$ and an attacker A such that there is only one rule per feature of the form $r : [a, b] \xrightarrow{f}_k [\delta_l, \delta_u]$, the computational complexity of deciding whether $\boldsymbol{x}$ belongs to $\mathcal{D}_u^{\lambda}(f, v, A)$ is $O(1)$.*

Our experimental evaluation builds on the threat model supported by the proposition above, which shows that the ternary partitioning can be efficiently performed in practical use cases.

4.6 From Decision Trees to Tree Ensembles

In this section we introduced a new tree learning algorithm, yet individual decision trees are rarely used in practice and ensemble methods are generally preferred for real-world tasks. As anticipated in Section 2, the most popular ensemble methods for decision trees are Random Forest (RF) and Gradient Boosting Decision Trees (GBDT). Extending Treant to these ensemble methods is straightforward, because both methods can be seen as *meta-algorithms* which build on top of existing tree learning algorithms. RF builds multiple independent trees $t_1, \ldots, t_n$ by using bagging and per-node feature sampling in each t_j, while in GBDT each tree t_i adds a gradient descent step to minimize the cumulative loss incurred by the previous trees. Hence, both methods eventually apply an underlying tree learning algorithm multiple times to different training data.

The only delicate point to notice is that, since the Treant algorithm is parametric over an attacker $A = (R, K)$, using the same attacker in the individual tree constructions is a *conservative* approach to ensemble learning. This comes from two factors: first, the construction of each tree t_j relies on a robust splitting procedure which only accounts for attacks against t_j, yet other trees in the ensemble might contribute to make such attacks ineffective; second, the attacker's budget K is essentially refreshed along each tree construction. This conservative approach overestimates the power of the attacker and cannot harm security, though it might unnecessarily downgrade performance in the unattacked setting. That said, our experimental evaluation in the next section shows that ensembles built using Treant are very accurate also in such setting. We leave the design of more sophisticated ensemble learning techniques to future work.

5 Experimental Evaluation

5.1 Methodology

We compare the performance of classifiers trained by different learning algorithms: two standard approaches, i.e., Random Forest (Breiman, 2001) (RF) and Gradient Boosting Decision Trees (Friedman, 2001) (GBDT) as provided by the LightGBM[10] framework; two state-of-the-art adversarial learning techniques, i.e., Adversarial Boosting (Kantchelian et al., 2016) (AB) and Robust Trees (Chen et al., 2019) (RT); and a Random Forest of trees trained using the proposed Treant algorithm (RF-Treant). Notice that the original implementation of AB exploited

[10] `https://github.com/microsoft/LightGBM`

Table 2 Main statistics of the datasets used in our experiments.

Dataset	`census`	`wine`	`credit`	`malware`
n. of instances	45,222	6,497	30,000	47,580
n. of features	13	12	24	1,000
class distribution (pos.÷neg. %)	25÷75	63÷37	22÷78	96÷4

a heuristic algorithm to quickly find effective adversarial examples, which does not guarantee to find the most damaging attack. Our own implementation of AB, which is built on top of LightGBM, exploits the white-box attack generation method described in Section 3.3 to identify the *best* adversarial examples. In this regard, our implementation of AB is more effective than the original algorithm.

We perform our experimental evaluation on four publicly available datasets, using three standard validity measures: accuracy, macro F1 and ROC AUC. We compute all measures both in absence of attacks and under attack, using our white-box attack generation method. We used a 60-20-20 train-validation-test split through stratified sampling. Hyper-parameter tuning on the validation data was conducted to set the number of trees (≤ 100), number of leaves ($\{8, 32, 256\}$) and learning rate ($\{0.01, 0.05, 0.1\}$) of the various ensembles so as to maximize ROC AUC. All the results reported below were measured on the test data. Observe that all the compared adversarial learning techniques are parametric with respect to the budget granted to the attacker, modeling his power: we consider multiple instances of such budget both for training (*train budget*) and for testing (*test budget*).

5.2 Datasets and Threat Models

We perform our experimental evaluation on four datasets: Census Income,[11] Wine Quality,[12] Default of Credit Cards,[13] and Malware Analysis.[14] We refer to such datasets as `census`, `wine`, `credit`, and `malware`, respectively. Their main statistics are shown in Table 2; each dataset is associated with a binary classification task.[15]

We therefore design different threat models by means of sets of rewriting rules indicating the attacker capabilities, with each set tailored to a given dataset. The features targeted by those rules have been selected after a preliminary data exploration stage, where we investigated the importance and the data distribution of all the features, e.g., to identify the magnitude of adversarial perturbations. Of course, in a real-world deployment the definition of the appropriate threat model would depend on the specific application scenarios of the trained classifiers: the definitions here considered are evocative of plausible attack scenarios possibly anticipated by domain experts, yet they are primarily intended as a way to test the robustness of the trained models against evasion attacks.

[11] `https://archive.ics.uci.edu/ml/datasets/census+income`

[12] `https://www.kaggle.com/c/uci-wine-quality-dataset/data`

[13] `https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients`

[14] `https://ieee-dataport.org/open-access/malware-analysis-datasets-top-1000-pe-imports`

[15] The `wine` dataset was originally conceived for a multiclass classification problem; we turned that into a binary one, where the positive class identifies good-quality wines (i.e., those whose quality is at least 6, on a 0-10 scale) and the negative class contains the remaining instances.

In the case of `census`, we define six rewriting rules: (*i*) if a citizen never worked, he can pretend that he actually works without pay; (*ii*) if a citizen is divorced or separated, he can pretend that he never got married; (*iii*) a citizen can present his occupation as a generic "other service"; (*iv*) a citizen can cheat on his education level by lowering it by 1; (*v*) a citizen can add up to $2,000 to his capital gain; (*vi*) a citizen can add up to 4 hrs per week to his working hours. We let (*i*),(*ii*), and (*iii*) cost 1, (*iv*) cost 20, (*v*) cost 50, and finally (*vi*) cost 100 budget units. We consider 30, 60, 90, and 120 as possible values for the budget.

In the case of `wine`, we specify four rewriting rules: (*i*) the alcohol level can be increased by 0.5% if its original value is less than 11%; (*ii*) the residual sugar can be decreased by 0.25 g/L if it is already greater than 2 g/L; (*iii*) the volatile acidity can be reduced by 0.1 g/L if it is already greater than 0.25 g/L; (*iv*) free sulfur dioxide can be reduced by -2 g/L if it is already greater than 25 g/L. We let (*i*) cost 20, (*ii*) and (*iii*) cost 30, and (*iv*) cost 50 budget units. We consider 20, 40, 60, 80 and 100 as possible values for the budget.

For `credit`, the attacker is represented by three rewriting rules: (*i*) the repayment status of August and September can be reduced by 1 month if the payment is delayed up to 5 months; (*ii*) the amount of bill statement in September can be decreased by 4,000 NT dollars if it is between 20,000 and 500,000; and (*iii*) the amount of given credit can be increased by 20,000 NT dollars if it is below 200,000. For each rule, a cost of 10 budget units is required. We consider 10, 30, 40, and 60 as possible budget values.

Finally, the attacker targeting the `malware` dataset is modelled by three rewriting rules, which allow one to flip three binary features from 0 to 1. These features represent invocations to the following functions: `_cexit`, `SearchPathW`, `exit`. Each rule has cost 20 and we pick three possible values of the budget: 20, 40 and 60. Note that since the `malware` dataset is imbalanced, at training time we oversampled the minority class so as to increase ten times the corresponding instances, yet the oversampling was not applied at test time. For such imbalanced datasets, accuracy values are less relevant than macro F1 and ROC measures, for which measures a trivial classifier always predicting the majority class would barely score 0.50.

5.3 Experimental Evaluation

The primary goal of our experiments is answering the following research questions:

1. What is the robustness of standard decision tree ensembles like RF and GBDT against evasion attacks?
2. Can adversarial learning techniques improve robustness against evasion attacks and which technique is the most effective?
3. What is the impact of the training budget on the effectiveness of adversarial learning techniques?
4. What are the key structural properties of the decision trees trained by TREANT, i.e., why do they provide appropriate robustness guarantees?
5. What is the performance overhead of TREANT compared to other adversarial learning techniques?

We report on the answer to each research question in a separate sub-section.

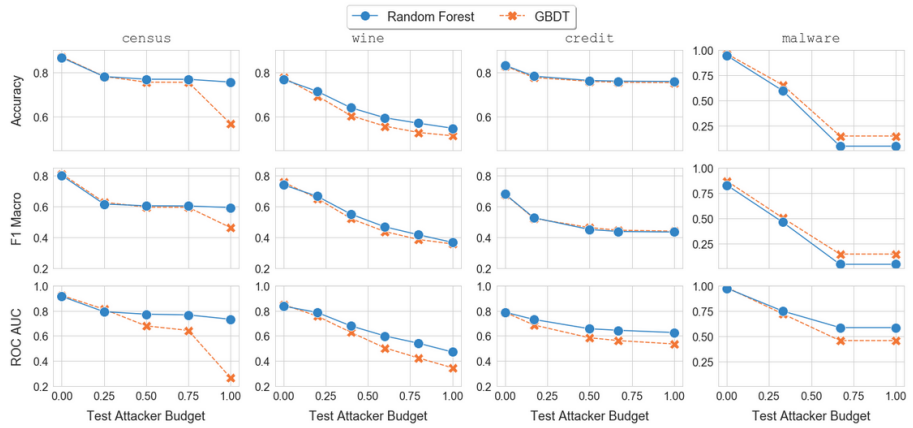

Fig. 3 The impact of the attacker on RF and GBDT.

5.3.1 Robustness of Standard Decision Tree Ensembles

In Figure 3, we show how the accuracy, F1 and ROC AUC of standard ensembles of decision trees trained by RF and GBDT change in presence of attacks. The x-axis indicates the testing budget of the attacker, normalized in the range $[0, 1]$, with a value of 0 denoting the unattacked scenario.

Two main findings appear clear from the plots. First, both GBDT and RF are severely impacted when they are attacked, and their performance deteriorates to the point of turning them into almost random classifiers already when the attacker spends just half of the maximum budget, e.g., in the case of the `wine` dataset. On that dataset, the drop of ROC AUC ranges from -25.8% to -40.6% for GBDT and from -15.5% to -28.4% for RF, when the attacker is supplied just half of the budget. Second, RF typically behaves better than GBDT on our validity measures, with a few cases where the improvement is very significant. A possible explanation of this phenomenon is that RF usually exhibits better generalization performance, while GBDT is known to be more susceptible to jiggling data, therefore more likely to overfit (Nawar and Mouazen, 2017). Since robustness to adversarial examples in a way resembles the ability of a model to generalize, RF is less affected by the attacker than GBDT. Still, the performance drop under attack is so massive even for RF that none of the traditional methods can be reliably adopted in an adversarial setting.

The higher resiliency of RF to adversarial examples motivated our choice to deploy TREANT on top of such ensemble method in our implementation. It is worth remarking though that TREANT is still general enough to be plugged into other frameworks for ensemble tree learning.

5.3.2 Robustness of Adversarial Learning Techniques

We now measure the benefit of using adversarial learning techniques to contrast the impact of evasion attacks at test time. More specifically, we validate the robustness of our method in comparison with the two state-of-the-art adversarial

Table 3 Comparison of adversarial learning techniques (training and test budget coincide). The asterisk denotes statistically significant difference against the best competitor with p value 0.01 under McNemar test.

			AB			RT			RF-TREANT		
			Acc.	F_1	ROC	Acc.	F_1	ROC	Acc.	F_1	ROC
census	Budget	30	**0.85**	**0.78**	**0.90**	0.81	0.69	0.88	**0.85**	0.77	**0.90**
		60	0.78	0.69	0.83	0.81	0.70	0.87	**0.85***	**0.77**	**0.89**
		90	0.80	0.71	0.83	0.78	0.61	0.86	**0.85***	**0.77**	**0.89**
		120	0.79	0.69	0.79	0.74	0.56	0.53	**0.84***	**0.76**	**0.89**
wine	Budget	20	**0.76**	**0.74**	**0.82**	0.73	0.70	0.80	**0.76**	**0.74**	**0.82**
		40	0.72	**0.69**	0.79	0.63	0.57	0.67	**0.73***	**0.69**	**0.80**
		60	**0.72**	**0.69**	0.79	0.59	0.49	0.55	**0.72**	0.68	**0.80**
		80	0.72	0.68	0.77	0.62	0.54	0.63	**0.73***	**0.69**	**0.80**
		100	0.70	0.67	0.76	0.65	0.57	0.71	**0.73***	**0.69**	**0.80**
credit	Budget	10	0.81	0.64	0.75	0.81	0.63	0.75	**0.82***	**0.66**	**0.77**
		30	0.79	0.54	0.66	**0.81**	0.61	0.73	**0.81**	**0.62**	**0.75**
		40	0.78	0.55	0.66	**0.81**	**0.62**	0.73	**0.81**	**0.62**	**0.74**
		60	0.78	0.53	0.62	**0.81**	**0.62**	0.72	**0.81**	**0.62**	**0.74**
malware	Budget	20	**0.94**	0.78	0.95	**0.94**	**0.83**	**0.97**	**0.94**	**0.83**	**0.97**
		40	**0.94**	0.79	0.95	**0.94**	**0.83**	**0.97**	**0.94**	**0.83**	**0.97**
		60	0.88	0.69	0.94	0.94	**0.83**	**0.97**	**0.95**	**0.83**	**0.97**

learning methods: Adversarial Boosting (AB) and Robust Trees (RT). Note that the authors of RT did not experimentally compare RT against AB in their original work (Chen et al., 2019).

We first investigate how robust a model is when it is targeted by an attacker with a test budget exactly matching the training budget. This simulates the desirable scenario where the threat model was accurately defined, i.e., each model is trained knowing the actual attacker capabilities. Table 3 shows the results obtained by the different adversarial learning techniques for the different training/test budgets. It is clear how our method improves over its competitors, basically for all measures and datasets. Most importantly, the superiority of our approach often becomes more apparent as the strength of the attacker grows. For example, the percentage improvement in ROC AUC over AB on the `credit` dataset amounts to 2.1% for budget 10, while this improvement grows to 19.6% for budget 60. It is also interesting to show that the heuristic approach implemented in RT is not always representative of all the possible attacks which might occur at test time: RT behaves similarly to our method on the `credit` and `malware` datasets, but it performs way worse on the `census` and `wine` datasets. In particular, the heuristic approach exploited by RT is most effective on the `malware` dataset, likely due to the presence of simple binary features. Indeed, this proves the effectiveness of our proposed strategy in presence of more complex datasets where the attacker behaviour cannot be approximated through simple heuristic approaches.

The second analysis we carry out considers the case of adversarial learning techniques trained with the maximum available budget. We use security evaluation curves to measure how the performance of the compared methods changes when the test budget given to the attacker increases up to the maximum avail-

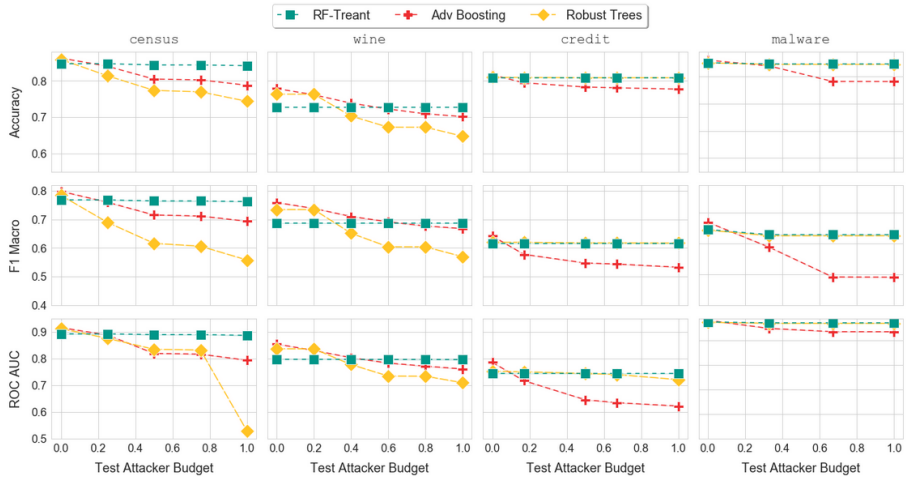

Fig. 4 Comparison of adversarial learning techniques for different test budgets and maximum train budget.

able. The results are shown in Figure 4, where we normalized the test budget in $[0, 1]$. The security evaluation curves show that our method constantly outperforms its competitors on all datasets and measures, especially when the attacker gets stronger. The price to pay for this increased protection is just a slight performance degradation in the unattacked setting, which is always compensated under attack. Indeed, the performance of our method is nearly constant and insensitive to variations in the attacker's budget, which is extremely useful when such information is hard to exactly quantify for security experts. We observe again that RT cannot always cope with strong attackers: this is particularly apparent in the case of the `census` dataset, where the model trained by RT is completely fooled when the test budget reaches its maximum.

5.3.3 Impact of the Training Budget

Another intriguing aspect to consider is how much adversarial learning techniques are affected by the assumptions made on the attacker's capabilities upon learning, i.e., the value of the training budget. Figure 5 is essentially the "dual" of Figure 4, where we consider the strongest possible attacker (with the largest test budget) and we analyze how much models learned with different training budgets are able to respond to evasion attempts.

We draw the following observations. First, our method leads to the most robust models for all measures and datasets, irrespective of the budget used for training. Moreover, our method is the one which most evidently presents a healthy, expected trend: the greater the training budget used to learn the model, the better its performance under attack. This trend eventually reaches its peak when the training budget matches the test budget. AB and RT show a more unpredictable behavior, as their performance fluctuates up and down, and sometimes suddenly drops. This is likely due to the fact that these approaches are heuristics and eventually shortsighted with respect to the set of all the attacks which might occur at test time.

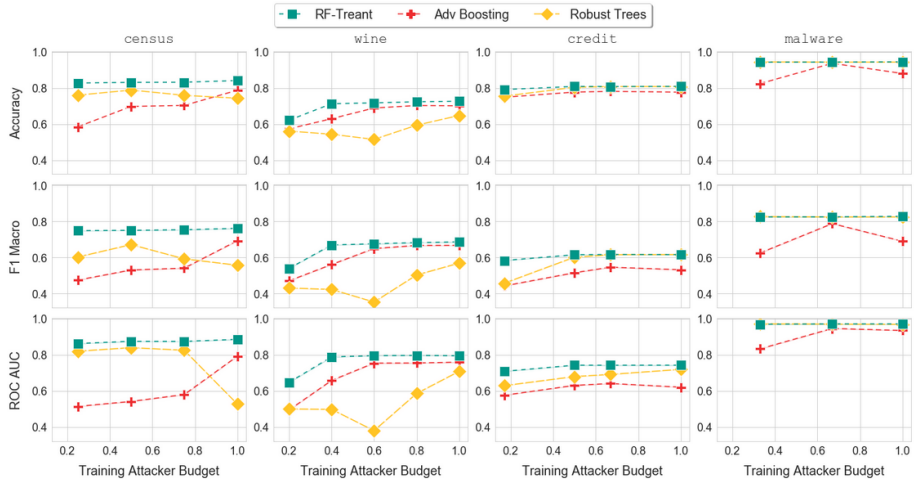

Fig. 5 Comparison of adversarial learning techniques for different train budgets and maximum test budget.

Finally, we remark a last appealing, distinctive aspect of our method: even when the training uses a significantly smaller budget than the one used by the attacker at test time, it already achieves nearly optimal performance. The same is not true for its competitors, which complicates their deployment in real-world settings, since it requires security experts to be very precise in their budget estimates.

5.3.4 Structural Properties of Decision Trees

To better understand why our approach provides improved robustness against attacks, we studied the feature importance of the trained adversarial learning models on the different datasets (considering the highest training budget). Figure 6 shows the plots built for the `wine` and `credit` datasets, where we use a grey background to denote attacked features.

In the case of the `wine` dataset, we observe that the alcohol level (though attacked) is a very useful feature for all the models. The importance of this feature seems inherent to the training data, however the figure is very different for other attacked features, like residual sugar and volatile acidity. These features are quite useful overall for AB and RT, while they are essentially not used by our model: this justifies the improved robustness of our method over its competitors. As to the case of the `credit` dataset, the feature importance of RT follows the same lines of our model: attacked features are essentially not used, while AB is fooled into giving a lot of importance to them. This motivates why AB performs quite worse than the other two methods there.

5.3.5 Efficiency Analysis

We conclude our experiments with an efficiency evaluation of our algorithm. We note that code optimization was not the main goal of our prototype implementation, i.e., we were more concerned about robustness to attacks than about effi-

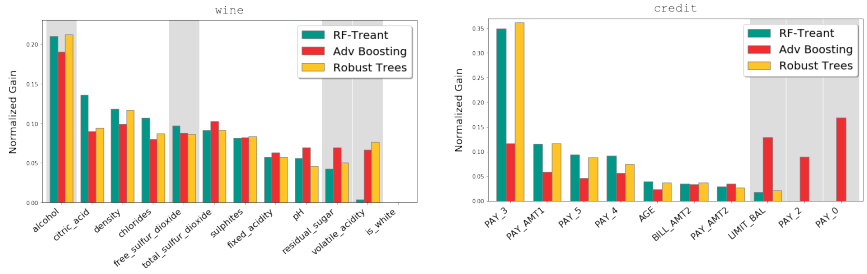

Fig. 6 Feature importance for `wine` and `credit` datasets. The grey background denotes attacked features.

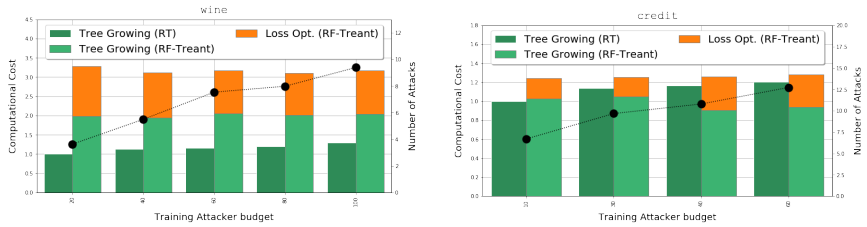

Fig. 7 Normalized training times for the `wine` and `credit` datasets. The black line shows the amount of attacks generated during training in terms of a multiplicative factor of the original dataset size.

ciency. However, it is still possible to draw interesting conclusions about efficiency as well. Figure 7 compares the training times of our models against those of the models trained by our implementation of RT on the `wine` and `credit` datasets. This analysis is insightful, because RT is essentially a simplified version of our approach, where the ternary partitioning is heuristically approximated and attack invariance is not enforced upon tree construction.

The figure reports the normalized running times with respect to the fastest training time, i.e., RT with the lowest train budget, as a function of the attacker budget. The figure also plots the amount of generated attacks in terms of the multiplicative increase of the original training dataset, due to the corrupted instances an attacker can generate for each budget.

Regarding RF-TREANT, we report a breakdown of its cost into loss optimization and tree growing. The impact of the loss optimization ranges between 20% and 30% of the total time. Fluctuations in the loss optimization cost are due to the use of a numerical solver, whose inner workings and resolution strategies may change on the basis of the number of variables, constraints, etc., of the optimization problem. The remaining cost of the tree growing phase, which includes the ternary partitioning, dominates the overall RF-TREANT training time. Overall, the cost of RF-TREANT is pretty stable when varying the attacker budget.

The figure confirms that RT is indeed faster than our approach, for all budgets. For the `wine` dataset, the overhead is essentially of a 3x factor, while for `credit` the cost of the two algorithms is actually very close. The overhead on `wine` however is largely justified by the complete coverage of all the possible attack strategies, which greatly improves the robustness guarantees provided by our approach (see

Table 3). An interesting trend is shown on the `credit` dataset, where with an increased budget the cost of solving the optimization problem increases, while the tree growing itself becomes cheaper than that of RT. This is likely due to the larger number of attacks that induces more complex optimization problems to be solved, i.e., with more constraints required to enforce attack invariance. At the same time, such constraints reduce the tree growing opportunities, thus reducing the cost of the tree growing.

It is also interesting to note that the number of generated attacks grows much faster than the training times: for example, when moving from budget 20 to budget 100, the number of generated attacks for the `wine` dataset increases from around 4x to around 9x the dataset size, yet the overall training time does not significantly increase. This means that the attack generation takes only a limited fraction of the training time, because each feature just needs to be attacked independently of all the others. Similar considerations apply to the `credit` dataset, though it is worth noticing that the overhead of our approach over RT is much more limited there for all budgets.

6 Conclusion

This paper proposes TREANT, a new adversarial learning algorithm that is able to grow decision trees that are resilient against evasion attacks. TREANT is the first algorithm which greedily, yet soundly, minimizes an evasion-aware loss function, capturing the attacker's goal of maximizing prediction errors. Our experiments, conducted on four publicly available datasets, confirm that TREANT produces accurate tree ensembles, which are extremely robust against evasion attacks. Compared to the state of the art, TREANT exhibits a significant improvement.

As future work, we plan to revise our decision tree construction to make it aware of its deployment inside an ensemble; in other words, we aim at exploiting the information that the currently grown ensemble is particularly strong or weak against some classes of attacks to guide the construction of the next member of the ensemble. At the same time, we want to explore ways to relax the restriction that each attacked feature is only tested once on each path of the decision trees, without sacrificing the soundness and scalability of the construction. It is also intriguing to explore further applications and extensions of our proposed threat model: for example, we consider to take advantage of work on inverse classification (Lash et al., 2017) to express dependencies between different features, e.g., features which cannot be manipulated, but are computed as a function of other corrupted features.

On the experimental side, we would like to evaluate our learning technique against regression datasets to get an additional quantitative evaluation of its security benefits. Finally, we plan to investigate the combined use of standard decision trees and decision trees trained using TREANT in the same ensemble, to improve the trade-off between accuracy in the unattacked setting and resilience to attacks.

References

B. Biggio and F. Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.

B. Biggio, G. Fumera, and F. Roli. Multiple classifier systems for robust classifier design in adversarial environments. *Int. J. Machine Learning & Cybernetics*, 1 (1-4):27–41, 2010.

B. Biggio, B. Nelson, and P. Laskov. Support vector machines under adversarial label noise. In *ACML Asian conference on machine learning*, pages 97–112, 2011.

B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Srndic, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD*, pages 387–402, 2013.

B. Biggio, G. Fumera, and F. Roli. Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering*, 26(4):984–996, 2014.

L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984. ISBN 0-534-98053-8.

S. Calzavara, C. Lucchese, and G. Tolomei. Adversarial training of gradient-boosted decision trees. In W. Zhu, D. Tao, X. Cheng, P. Cui, E. A. Rundensteiner, D. Carmel, Q. He, and J. X. Yu, editors, *Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, Beijing, China, November 3-7, 2019*, pages 2429–2432. ACM, 2019.

N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *IEEE symposium on security and privacy S&P*, pages 39–57, 2017.

H. Chen, H. Zhang, D. S. Boning, and C. Hsieh. Robust decision trees against adversarial examples. In *International Conference on Machine Learning ICML*, pages 1122–1131, 2019.

F. Chollet. *Deep Learning with Python*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2017. ISBN 1617294438, 9781617294433.

H. Dang, Y. Huang, and E. Chang. Evading classifiers by morphing in the dark. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 119–133, 2017.

J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations ICLR*, 2015.

S. Gu and L. Rigazio. Towards deep neural network architectures robust to adversarial examples. In *International Conference on Learning Representations ICLR, Workshop Track Proceedings*, 2015.

W. He, J. Wei, X. Chen, N. Carlini, and D. Song. Adversarial example defense: Ensembles of weak defenses are not strong. In *USENIX Workshop on Offensive Technologies WOOT*, 2017.

S. Hershkop and S. J. Stolfo. Combining email models for false positive reduction. In *ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 98–107, 2005. doi: 10.1145/1081870.1081885.

L. Huang, A. D. Joseph, B. Nelson, B. I. P. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *ACM Workshop on Security and Artificial Intelligence AISec*, pages 43–58, 2011.

E. B. Hunt, J. Marin, and P. J. Stone. Experiments in induction. 1966.

L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15–17, 1976.

A. Kantchelian, J. D. Tygar, and A. D. Joseph. Evasion and hardening of tree ensemble classifiers. In *International Conference on Machine Learning ICML*, pages 2387–2396, 2016.

D. Kraft. Algorithm 733: Tomp–fortran modules for optimal control calculations. *ACM Transactions on Mathematical Software TOMS*, 20(3):262–281, Sept. 1994. ISSN 0098-3500. doi: 10.1145/192115.192124. URL `https://doi.org/10.1145/192115.192124`.

M. T. Lash, Q. Lin, W. N. Street, and J. G. Robinson. A budget-constrained inverse classification framework for smooth classifiers. In *IEEE International Conference on Data Mining Workshops ICDMW*, pages 1184–1193, 2017.

D. Lowd and C. Meek. Adversarial learning. In *ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 641–647, 2005.

A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations ICLR*, 2018.

M. Mehta, R. Agrawal, and J. Rissanen. Sliq: A fast scalable classifier for data mining. In *International conference on extending database technology*, pages 18–32. Springer, 1996.

S. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. Deepfool: A simple and accurate method to fool deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition CVPR*, pages 2574–2582, 2016.

S. K. Murthy. Automatic construction of decision trees from data: A multi-disciplinary survey. *Data Minining and Knowledge Discovery*, 2(4):345–389, 1998.

S. Nawar and A. Mouazen. Comparison between random forests, artificial neural networks and gradient boosted machines methods of on-line vis-nir spectroscopy measurements of soil total nitrogen and total carbon. *Sensors*, 17(10):2428, 2017.

B. Nelson, B. I. P. Rubinstein, L. Huang, A. D. Joseph, S. Lau, S. J. Lee, S. Rao, A. Tran, and J. D. Tygar. Near-optimal evasion of convex-inducing classifiers. In *International Conference on Artificial Intelligence and Statistics AISTATS*, pages 549–556, 2010.

A. M. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *IEEE Conference on Computer Vision and Pattern Recognition CVPR*, pages 427–436, 2015.

N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *IEEE European symposium on security and privacy EuroS&P*, pages 372–387, 2016a.

N. Papernot, P. D. McDaniel, X. Wu, S. Jha, and A. Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *IEEE symposium on security and privacy S&P*, pages 582–597, 2016b.

R. Perdisci, G. Gu, and W. Lee. Using an ensemble of one-class SVM classifiers to harden payload-based anomaly detection systems. In *IEEE International Conference on Data Mining ICDM*, pages 488–498, 2006. doi: 10.1109/ICDM.2006.165.

J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

N. Srndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *IEEE symposium on security and privacy S&P*, pages 197–211, 2014.

C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. J. Goodfellow, and R. Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations ICLR*, 2014.

G. Tolomei, F. Silvestri, A. Haines, and M. Lalmas. Interpretable predictions of tree-based ensembles via actionable feature tweaking. In *ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 465–474, 2017.

T. P. Tran, P. Tsai, and T. Jan. An adjustable combination of linear regression and modified probabilistic neural network for anti-spam filtering. In *International Conference on Pattern Recognition ICPR*, pages 1–4, 2008. doi: 10.1109/ICPR.2008.4761358.

D. E. Tyler. Robust statistics: Theory and methods, 2008.

V. Vapnik. Principles of risk minimization for learning theory. In *Advances in neural information processing systems*, pages 831–838, 1992.

H. Xiao, B. Biggio, B. Nelson, H. Xiao, C. Eckert, and F. Roli. Support vector machines under adversarial label contamination. *Neurocomputing*, 160:53–62, 2015.

F. Zhang, Y. Wang, S. Liu, and H. Wang. Decision-based evasion attacks on tree ensemble classifiers. *World Wide Web*, pages 1–21.