TÉCNICO LISBOA

A Fraud Detection System Robust to Adversarial Perturbations

Tiago Manuel Borges Leon Gomes de Melo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Paolo Romano
Dr. Marco Sampaio

Examination Committee

Chairperson: Prof. Pedro Tiago Gonçalves Monteiro
Supervisor: Prof. Paolo Romano
Member of the Committee: Prof. Francisco António Chaves Saraiva de Melo

November 2022

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

First, I would like to give my thanks to João Bravo for the invaluable, continuous support provided throughout this project. I cannot put to words how grateful I am to have had the privilege of working with him. I would also like to thank Marco Sampaio for supervising this work, his sharp attention to detail and making sure this thesis is the best version it can be. I also want to give my thanks to my thesis supervisor Prof. Paolo Romano, for always having refreshing ideas and bringing new perspectives that brought so much value to this work.

I would like to thank Feedzai, namely João Ascensão and Pedro Bizarro, for granting me the opportunity to work in such an ambitious and efficient environment. I can say without a doubt that seeing the work being done here inspired me to push myself to be better - and for that I could not be more grateful.

To my parents, Ana Margarida Leon and António Melo, my deep gratitude for raising me to be who I am today and supporting me every step of the way. To my grandmother Maria Eduarda, my thanks for getting me through the last push of this work.

I would also like to thank my friends Miguel and João. This journey was all the more enjoyable with you by my side.

Lastly, I would like to give a special thanks to my brother, Pedro, for all the words of encouragement and our moments of fun that I so much treasure.

Abstract

Adversarial attacks have quickly become a concern regarding many security-centered applications. Typically, the goal with these attacks is to mislead the classifier into producing the incorrect output while keeping the attack fairly similar to a clean example. Fraud detection systems have a naturally adversarial setting, in the sense that there is often an adversary (that is, a fraudster) trying to bypass a model, and a classifier trying to outguess this adversary. Fraudsters have been known to continuously adapt their strategies in order to maximize their chances of fooling the model, which leads the concept of fraud to change over time. To keep up with this concept drift, fraud detection systems often rely on frequently retraining the model on fresher data least they grow stale and their performance degrades. This is an expensive operation. In this work, we propose preempting this adversarial adaptation rather than reacting to it. We hypothesize that by training a model robust to adversarial attacks it will perform better against future attack strategies and thus require less model refreshes. We find that by generating a wide range of attacks and exposing the model to these adversarial samples during training, we obtain a significantly more robust model without considerable loss of performance in historical data, avoiding drops in performance that can be larger than 50% if attacks of the same type are used against a non-adversarially trained baseline.

Keywords

Adversarial robustness; Tabular data; Fraud detection; Tree-based models; Discrete optimization

Resumo

A descoberta de ataques adversariais rapidamente se propagou como uma ameaça a sistemas centrados em segurança. Frequentemente, o objetivo destes ataques é enganar um modelo e levá-lo a fazer uma classificação incorreta; idealmente mantendo o ataque o mais semelhante possível à vítima. Neste sentido, a natureza de sistemas de deteção de fraude pode facilmente ser considerada adversarial: há um adversário (isto é, um fraudista) que pretende enganar um modelo, e um modelo que idealmente não se deixa enganar pelo fraudista. Esta interação entre as duas partes é o principal motivo por detrás da mudança no conceito de fraude ao longo do tempo. Cunhada como *concept drift* ("derrapagem de conceito"), esta mudança constante leva a que a *performance* dos modelos responsáveis pela deteção de fraude seja lentamente deteriorada, culminando na sua obsolescência. Consequentemente, uma estratégia comum para prevenir esta mesma obsolescência consiste em retreinar os modelos com dados mais recentes de tempo a tempo. Na prática, esta é uma operação dispendiosa. Nesta dissertação hipotetizamos que um modelo que seja robusto a ataques adversariais possa generalizar melhor e seja melhor a detetar fraude futura. Neste sentido, este modelo robusto poderia ser retreinado menos vezes, poupando recursos a longo prazo. Nesta dissertação propomos uma *framework* que visa melhorar a robustez de modelos a ataques adversariais. Os resultados obtidos levam-nos a concluir que o nosso método permite uma melhoria significativa contra ataques adversariais sem grande sacrifício de *performance* em dados históricos, evitando perdas em *performance* superiores a 50% caso estes ataques sejam usados contra um modelo que não tenha sido treinado adversarialmente.

Palavras Chave

Robustez adversarial; Dados tabulares; Deteção de fraude; Modelos em árvore; Optimização discreta

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AUC	Area Under the Curve
CP	Card Present
CNP	Card Not Present
CVV	Card Verification Value
DL	Deep Learning
FGSM	Fast Gradient Sign Method
FPR	False Positive Rate
IOF	Inverse Occurrence Frequency
IP	Internet Protocol
pAUC	partial Area Under the ROC Curve
ML	Machine Learning
PGD	Projected Gradient Descent
RMSE	Root Mean Squared Error
ROC	Receiver Operating Characteristic
SCD	Stochastic Coordinate Descent
TOR	The Onion Router
TPR	True Positive Rate
VPN	Virtual Private Network

1

Introduction

Contents

This chapter briefly introduces the topics we aim to cover in this thesis. We start by discussing the underlying motivation for improving fraud detection models. We then discuss how there might be potential improvements by framing this problem of fraud detection in an adversarial setting. Lastly, we enumerate the goals we aim to achieve in this work.

1.1 Motivation

Machine Learning models are a vital part of many security-sensitive applications [2–5], in which system designers often assume classifiers to obey certain security requirements. However, recent work [6] has uncovered properties of neural networks that might compromise these assumptions. Indeed, many applications with Deep Learning at heart might be vulnerable to *adversarial attacks*. More so, it has also been shown that these attacks are not limited to neural network-based models, but decision trees and random forests as well [7, 8]. In such attacks, adversaries maliciously tamper with the input examples in order to lead models to misclassify these examples. In the computer vision domain, these new samples are designed to be kept as similar to the original ones as possible, and might not even occur naturally in the original training set. The original notion of an adversarial attack is to generate inputs that are virtually indistinguishable from real data but are wrongly classified by the target model [9, 10].

The detection of credit card fraud is one of the largest industries with machine learning at its core. There is a high value in quickly classifying a transaction as legitimate or fraudulent and high capital investments have been placed in this industry [9]. Naturally, the driving force behind the concept of "fraud" are fraudsters, who have been known to continuously adapt their techniques to try to bypass fraud detection systems. Such is a naturally adversarial setting in which we have an opponent that can directly profit from fooling a classifier.

This premise drives the need to develop a robust model that fraudsters should struggle with bypassing. Typically, fraud detection systems rely on periodically "refreshing" the model by retraining it with more recent data to prevent their performance from degrading in the face of changing fraud patterns. This is an expensive operation that requires significant resources. Moreover, model refreshes are even less straightforward if we consider that we cannot use very recent data due to the inherent delay in labels in this domain. If instead of "reacting" to changes in fraud we preemptively train more robust classifiers, model refreshing could be done more sparingly. Ultimately, we aim to develop a model that is able to catch current and potentially prevent future fraud strategies.

1.2 Problem Statement

We now provide an informal formulation of the setting and problem this thesis aims to solve. Fraud detection systems aim to quickly and accurately detect fraudulent transactions: ultimately, the goal of a fraud detection system is to distinguish between fraudulent and legitimate financial transactions. In this setting, most often the data used to achieve this goal is tabular, with each row representing a unique transaction. Additionally, in order to aid the performance of the classifiers, it is often essential to enrich the feature space through feature engineering. It is using this enriched dataset that a machine learning model is trained to then start classifying transactions.

We can think of a fraudster as an adversary who is trying to capitalize on eventual breaches the fraud detection system might have. Unlike in many settings, credit card fraudsters do not have complete access to the input that is submitted to the model. In other words, they cannot freely adapt every feature to generate an adversarial sample, since some features are automatically generated without the fraudster's knowledge. Another key characteristic of credit card fraud detection is that the data is tabular and imbalanced. Since their origin, adversarial attacks are most common within an image classification context: numerical pixel data with most classes following a balanced distribution. This is seldom the case for fraud detection systems. Adversarially manipulating tabular data requires a substantially different approach and, as a result, so does developing models able to withstand these manipulations. Lastly, the inherent imbalance between fraudulent and legitimate transactions often results in models naturally skewed to the majority class. This handicap also plays a significant role in training high-performing models that can efficiently classify samples.

1.3 Thesis Goals

Bearing in mind these concerns, we highlight the need to develop a model that is able to withstand the attacks of an adversary without much loss of performance in clean samples. Our main goal is to propose an adversarial model training method for such a fraud detection system. Our main contributions are:

- We develop a method to generate strong attacks against tabular transaction data. Devising an efficient way of generating attacks is not only necessary to train robust models, but also to provide an accurate measure of the robustness of a model.
- We introduce training frameworks that increase the robustness of a classifier against a broad range of attacks. Particularly, we develop a model that is well protected against multiple attack strategies, regardless of whether or not it was trained against them
- We achieve the former contribution without sacrificing significant performance on historical data, despite there being a trade-off between robustness and accuracy [1, 11].

2

Related Work

Contents

Adversarial robustness has recently gained a lot of interest in classification literature, with many studies addressing new strategies to design attacks. However, most literature leans towards the image classification context, with relatively little research tackling this concern in the tabular domain.

We start by providing an overview on the underlying foundations of adversarial robustness. This encompasses the first approaches towards defining and achieving it as well as some key properties that adversarial examples entail. We then formally define the problem, discuss the primordial frameworks for generating attacks and defenses, and finally present the fundamental classification of the different types of adversarial attack strategies.

As we will see, despite stemming from an image classification Deep Learning (DL) context, adversarial attacks are applicable on both image and tabular domains. From this premise comes the need to approach this problem from both standpoints, in each considering the different possible ways to generate attacks and defenses. This section is thus structured as follows. We start by analyzing the concept of adversarial robustness in neural networks and formally defining the problem within this scope. We also discuss the properties of adversarial attacks as well as some brief overview of the literature on the offensive and defensive vectors. We then move on to more relevant concepts within this domain. Particularly, we address how generating adversarial examples differs in and image data, and review attacks against tree-based models.

2.1 Adversarial Robustness on Neural Networks

In 2004, Dalvi et al. [12] originally formulated the concept of *Adversarial Classification* as a game between a classifier, whose goal is to correctly identify the label of a given instance, and an adversary, whose goal is to mislead the classifier into labelling positive instances as negative. This research presents the problem as a min-max optimization problem, solvable using Linear Programming. Although the authors mention fundamental concepts in the context of Adversarial Robustness, most of the research targeted Bayesian Classifiers.

When originally published, state of the art Deep Neural Networks were reported to have extremely high performance on both speech and image classification tasks [6, 13, 14]. However, in 2013 Szegedy et al. [6] present two counter-intuitive results that raise important awareness towards the robustness of neural networks and subsequently, machine learning models in general. The authors demonstrate how this high accuracy does not necessarily translate to robustness, contrary to prior belief and intuition. Firstly, it is shown that by introducing an imperceptible, seemingly random perturbation, we can influence neural networks to produce wrong outputs (coining the term as *adversarial examples*) and even lead such models to classify instances as we may desire (defined as *targeted attacks*). Moreover, Szegedy et al. also prove these so-called adversarial examples are not necessarily restricted to one single model.

Instead, an example mislabelled by a specific neural network can often be *transferred* to others, fooling them all-the-same.

These findings paved the way to research on both ends: how adversarial examples can effectively be generated (adversarial attacks) and how can we protect our models against adversarial examples, i.e., how can we develop adversarially robust machine learning models. The latter is particularly relevant in contexts where security is a primary concern, and it is mainly driven by the former. Such attacks pose a serious threat in many DL and non-DL classification systems where an adversary can benefit from model mistakes. As pointed out by Papernot et al. [2], such attacks are frequent in many use cases [15].

2.1.1 Problem Statement

A common formulation of adversarial attacks found in the literature [6, 16], is the following. Let us consider an input x of dimension k and a corresponding label y. Moreover, let us also consider a mapping $h_\theta : X^k \rightarrow \mathbb{Y}$ from the input space to the output space, that is $h_\theta(x) = y$, with θ being the model parameters. As originally proposed by Szegedy et al. [6], we are interested in finding the smallest perturbation δ that causes a change in the output:

$$\min_{\delta \in \Delta} \|\delta\| \quad \text{s.t.} \quad h_\theta(x + \delta) \neq y, \tag{2.1}$$

This is a formulation that prioritizes the generation of imperceptible attacks. However, in practical terms we can consider that an attacker will often want to maximize their chances of developing a successful attack. Thus, literature often also defines the goal of generating an adversarial example as finding a sample that maximizes the loss of the classifier it is targeting. So, let us also consider the loss function as $J(\theta, x, y)$, which corresponds to the cost of mislabelling an example x of label y using model parameters θ. This cost function is typically what we want to minimize when optimizing for θ. In neural networks, this optimization is typically done using (stochastic) gradient descent. We compute the gradient of the loss function with respect to θ, and update it in the opposite direction. In contrast, an adversary will often want to adjust the input (for example, an image) so that they maximize the target model's loss function. In this formulation, the constraint is placed on the values of $\tilde{x} = x + \delta$ that are available to the attacker since it would be meaningless if any point in the input space was available. As a result, at every point we optimize over a constrained perturbation space, looking for the perturbation which will result in the largest classifier loss. Let us consider this allowable set of perturbations to be denoted by Δ. We thus formulate that the adversarial perturbations we can choose from are constrained by $\delta \in \Delta$, with the problem we are trying to solve being:

$$\max_{\delta \in \Delta} J(\theta, x + \delta, y) \tag{2.2}$$

For example, a frequently used set of perturbations in the context of image classification is the l_∞ ball

centered at x defined as [16]:

$$\Delta = \{\delta : \|\delta\|_\infty \leq \epsilon\} \tag{2.3}$$

where $\|\delta\|_\infty = \max_i |\delta_i|$. We allow each feature of the input to be perturbed by, at most, ϵ. In tabular domains and other contexts, this set does not necessarily make sense since the data may not be homogeneous. As a result, we may need another way of defining a good set of perturbations. This is discussed in further detail in the next sections.

From these formulations, we can infer there is yet another way we can capitalize on perturbations that maximize the loss of a model: *targeted attacks*. In short, the main goal of a targeted attack is to change the original input x in such a way that the input will be classified as a target class. Targeted attacks can be generated by optimizing the input so that the loss is smallest for a particular label. This is done very similarly to the aforementioned method, in the sense that we look for the perturbation that will result in the smallest loss (for a specific class) while simultaneously maximizing the loss for any other class [16]. However, since our case focuses on binary classification, these formulations do not apply, as there is only one other label to change to.

This covers the optimization problem an adversary must solve when trying to fool a classification model. From a classifier standpoint, our goal is still similar to that of Empirical Risk Minimization [17]: finding the weights that minimize the loss in the training set. However, we now must account for an adversary that is trying to maximize the same loss w.r.t. the model input. Put together, when the adversarial goal is Eq. (2.2), developing an adversarially robust classifier becomes a min-max optimization problem, the inner maximization loop being the adversary and the outer being our own classifier:

$$\min_\theta \left(\max_{\delta \in \Delta} J(\theta, x + \delta, y) \right) \tag{2.4}$$

This formulation defines the underlying intuition for one of the most popular defense strategies against adversarial attacks: *adversarial training*. In broad terms, it consists of injecting adversarial examples in the training set, exposing the model to attacks generated using a specific strategy. We will look into adversarial training in further detail later on.

2.1.2 Taxonomy of Adversarial Attacks

Adversarial examples threaten the reliability of security-centered machine learning services [6]. This is of particular relevance if we consider that an example misclassified by model A is frequently also misclassified by another model B. This *transferability* property opens the door for many attacks that do not even need access to the real model being targeted, but can instead attack it using a proxy model [18]. However, we will later see that transferability is not without drawbacks, as there appears to be a trade-off [19] between how well an attack performs against a specific model and how well this success rate

holds against a range of models.

That being said, it is important to distinguish between two classes of attacks: white and black box.

- As the name suggests, a **white box** attack assumes an attacker not only has access to the weights and architecture of the model being attacked, but can also perform infinite queries and try out as many attacks as desired. Naturally, this approach poses a more serious threat to a target model, since attackers can iteratively perfect their own attacks and devise strategies with high success rates. [20].

- On the other hand, a **black box** attack [8] is argued to resemble real world scenarios more accurately, since it assumes no visibility over the model being attacked. In practice, attackers often only have access to a hard-label produced by the model and have a limited number of queries they can do before being denied further access.

It is worth noting that if we consider how adversarial examples are inherently transferable, much of the overhead of attacking a black-box system is reduced. Attackers can design and deploy a proxy model that somewhat resembles the victim and from there optimize their adversarial examples. In other words, this property enables an attacker in the black-box setting to optimize their attacks for their own proxy model and then deploy these attacks to fool its real-world counterpart [2, 6, 18]. Narodytska et al. [21] evaluated the use of proxy models to attack a black-box image classification network. The proxy model was developed by changing a few hyper-parameters of the original network: the learning rate, the size of filters, and the number of layers in the network. It was shown that 25 to 43% of the generated attacks when querying the original network were also adversarial against its variations.

2.1.3 Attack Strategies on Neural Networks

We now review the literature on attack generation against neural networks (NNs). Given its prevalence in image classification tasks, there is a large body of work addressing the development of adversarial attacks against this class of models. In general, the generation of adversarial attacks against NNs exploits the inherent optimization techniques of these models, such as the gradients used in back-propagation.

2.1.3.A Fast Gradient Sign Method

Goodfellow et al. [18] shows that a linear model is susceptible to adversarial examples as long as its input has sufficient dimensionality. Prior work relied on hypothesized properties of neural networks to explain this phenomena, such as their non-linearity. The authors argue this explanation is not only simpler but also explains why softmax regression is also vulnerable to adversarial examples [18].

In many domains, the data collected depends greatly on the precision of how it is represented. For instance in greyscale images, since each pixel is represented using only a single byte, all information below the 1/255 threshold is discarded [18]. From this premise, one expects a machine learning model not to respond differently when classifying an input x and an input $\tilde{x} = x + \delta$ if every feature in the perturbation δ is smaller than the precision they support. As put by the authors, *"we expect the classifier to assign the same class to x and $\tilde{x}$ so long as $\|\delta\|_\infty < \epsilon$, where ϵ is small enough to be discarded by the sensor or data storage apparatus associated with our problem."* [18].

However, if we consider the output of a linear model to be calculated using the dot product between a weight vector and an input, the output produced when using an adversarial sample $\tilde{x}$ is such that:

$$w^T \cdot \tilde{x} = w^T \cdot x + w^T \cdot \delta. \tag{2.5}$$

The adversarial perturbation δ causes the activation to grow by $w^T \cdot \delta$. We can increase the impact of the perturbation δ further by assigning $\delta = \epsilon \cdot sign(w)$. *"If w has n dimensions and the average magnitude of an element of the weight vector is m, then the activation will grow by ϵmn"* [18].

It is worth noting that $\|\delta\|_\infty$ does not grow with the dimensionality of the problem. Rather, since the change in activation caused by perturbation by δ can grow linearly with n, then for high dimensional problems, we can make many very small changes to the input that add up to one large change to the output without having much impact on the l_∞ norm of δ. Goodfellow et al. describe this phenomenon as "accidental steganography", where a linear model is forced to consider only the signal that aligns best with its weights, even if multiple signals are present and other signals have much greater amplitude [18].

As a result, one of the first approaches towards the generation of imperceptible attacks was presented by Goodfellow et al.: the Fast Gradient Sign Method (FGSM) [18]. FGSM capitalizes on the differentiable properties of a given network to find the smallest perturbation that results in a change of the output. Goodfellow et al. propose a method that allows us to compute the "best" perturbation efficiently (that is, the one that maximizes loss). The authors note that locally the perturbation that increases the loss the most is the one that aligns with the gradient w.r.t. the input, meaning the optimal perturbation δ^* can be defined as:

$$\delta^* = \epsilon \cdot \operatorname{sign}(\nabla_x J(\theta, x, y)) \,, \tag{2.6}$$

In the case of neural networks, computing the gradient to find this optimal perturbation is efficient thanks to back-propagation [18]. Note that this perturbation will be $\mp\epsilon$ depending on the sign of the gradient, so that it has an l_∞ norm of ϵ. This vector is multiplied by a constant ϵ depending on how large we can allow each feature perturbation to be.

As mentioned before, one of the most widely accepted approaches towards developing the robust-

ness of models is by adversarially training them, that is including adversarial examples in the training set. This regularization method seems to work particularly well to protect against FGSM [18]. Once again, when performing adversarial training, we are exposing the model to attacks that are generated as the model learns. Thus, Goodfellow et al. propose a method to regularize the network. Although this method lent itself to (seemingly) great robustness improvements, it was later the subject of criticism. Particularly, Madry et al. [10] argue that these improvements were restricted to FGSM attacks - the model overfit to the narrow range of adversarial examples that can be generated using FGSM. This behavior has also been investigated by [19]. Essentially, adversarial examples generated using FGSM ended up being easier to classify than clean examples. This results from the fact that the adversary produces a restricted set of perturbations that the model can overfit to.

2.1.3.B Projected Gradient Descent

One can interpret FGSM as being a "simple one-step scheme for maximizing the inner part of the saddle point formulation" 2.4 [10]. We can develop a stronger attack if we iteratively perfect the adversarial perturbation; that is, perform projected gradient descent upon the negative loss function. Considering an adversarial example generated using FGSM $\tilde{x} = x + \delta^*$ where δ^* is the optimal perturbation seen in Equation (2.6). If we consider FGSM corresponds to a single gradient step, we can consider Projected Gradient Descent (PGD) to try to solve equation 2.2 with multiple projected gradient steps. We can consider this multi-step "variant" to be formulated as:

$$\tilde{x}^{t+1} = \Pi_{x+\Delta}\left(\tilde{x}^t + \epsilon \cdot sign(\nabla_x J(\theta, \tilde{x}^t, y))\right) \tag{2.7}$$

with $\Pi_{x+\Delta}$ being some projection operation which acts on its argument by trying to find its closest point (w.r.t. some norm) within an allowed set (denoted by $x + \Delta$). Here we are essentially performing FGSM and projecting the result onto the set of valid adversarial examples. We do this in order to iteratively update our initial sample. PGD has been considered the ultimate first-order adversary, in the sense that it is the strongest attack that capitalizes on the local first order information of the network [10].

2.1.3.C DeepFool

Shortly after [6] and [18], Moosavi-Dezfooli [22] propose DeepFool. DeepFool is an untargeted attack method that relies on projecting a clean input onto the closest decision hyperplane using an l_p norm. The general idea of DeepFool is to iteratively linearize the classifier to generate small perturbations that are enough to cause a change in the classification label. The authors argue that an affine model's

robustness at a given point can be inferred by calculating its distance to the decision hyperplane. From this premise, now expanding to a general model, DeepFool iteratively linearizes the model around the current iteration point and the minimal perturbation is calculated using

$$\arg\min_{\delta_i} \|\delta_i\|_2 \quad \text{where} \quad h_\theta(x_i) + \nabla h_\theta(x_i)^T \cdot \delta_i = 0 \,. \tag{2.8}$$

Although this formulation is focused on binary classification, the authors later build on it to generalize to multi-class classification problems. DeepFool is tested using natural images against (at-the-time) state of the art methods, namely FGSM and L-BFGS [6, 18]. Coupled with competitive adversarial generation times (i. e., how long it takes to generate an adversarial example), DeepFool generates much smaller average perturbations. This, in turn, results in test errors that outperform FGSM when a model is trained using adversarial examples generated using DeepFool. Indeed, fine-tuning with DeepFool can improve the accuracy of the networks, whereas fine-tuning using FGSM led to a decrease of the test accuracy [22] . The authors also criticize FGSM for generating overly perturbed images that do not occur in the test data, hence decreasing the overall performance of the method [22].

2.1.3.D C&W attacks

C&W attacks [20] aimed to overthrow the defenses proposed by Papernot et al. when using distillation as a defense [2], using it as baseline to evaluate the effectiveness of their methods. The authors in reference [20] formally define the problem of finding an adversarial example as it was originally formulated [6] (see equation 2.1). This formulation essentially corresponds to finding the closest sample $x + \delta$ to x that effectively causes a model to change its output. The authors use l_0, l_2 and l_∞ to generate their attacks. It is shown that all three result in 100% success rates against distilled networks, with C&W attacks achieving it with smaller perturbations than those generated by DeepFool [22]. It is worth highlighting that since the C&W attack tries to find the smallest perturbation that causes a change in the output, it essentially "guarantees" a 100% success rate. FGSM [18], on the other hand, constrains the perturbation size and thus might not produce a successful attack. These cannot be fairly compared as they address different formulations.

2.1.3.E Zeroth Order Optimization

The previously mentioned approaches consider a white-box setting, in the sense that they assume an attacker has full access to the target model to be fooled. Although this enables an attacker to perform much stronger attacks, it is often argued that its black-box counterpart resembles real-world scenarios more accurately [8, 23].

Cheng et al. [8] propose a more general method to develop adversarial examples against black-box

models with hard-label outputs (instead of soft outputs with probabilities) [7]. The problem of crafting adversarial attacks in such a setting is only tractable if we can query the target model [24]. In [8], the authors propose an effective black-box attack using zeroth order optimization, querying the real target model.

In broad terms, zeroth order optimization is essentially an optimization method that does not rely on gradients. Such is of particular relevance in settings where gradients are impossible to calculate (for instance, when all the attacker has access to are the hard labels produced by the model). The Gradient-Free method used was originally proposed by [25, 26]. Consider a function $g_x(\theta)$ that returns the distance from an instance x and its nearest adversarial example along the direction θ. The authors of reference [8] use a gradient approximation to iteratively update the search direction θ along which we are searching for a change in an output. Cheng et al. develop efficient adversarial attacks whilst keeping distortion comparable to the white-box C&W attack [20]. Such results can be attainable using a relatively small number of queries to the target model. This method was later on perfected and evolved into *SignOPT* [27]. Departing from the Zeroth Order Optimization attacks, SignOPT improves the query complexity over reference [8]. It stands on two core ideas: very accurate values of the directional derivative are not required to lead the algorithm to converge, and there exists an imperfect but informative estimation of directional derivative of g that can be computed by a single query. SignOPT obtains much higher success rates for the same number of queries when compared with its parent.

2.1.4 Developing Robust Neural Networks

We now review some of the existing approaches to boost adversarial robustness in this class of models. Currently, there are two main frameworks for developing robust neural networks: *Distillation as a Defense* [2] and Adversarial Training [6, 18, 18, 24]. Before diving into the foundations of these methods, let us define (informally) what a robust model is and go over some broad notions on robust neural networks and robustness requirements.

The concept of robustness can be informally defined as a model's capacity to resist perturbations [2], that is, its ability to withstand small input noise without affecting its output and thus, without significantly hindering its performance. A robust model should yield good accuracy for its training set (possibly containing adversarial examples) but also for examples outside it. Since adversarial examples capitalize on areas of the domain the model does not cover, a robust model must generalize well to examples outside its training set. Intuitively, a robust model should provide a somewhat smooth function that classifies inputs consistently with their neighborhoods, even if it has not been exposed to them during training [2]. This neighborhood can be defined as the samples within a norm's distance, where this norm is properly suited for the input domain. In the case of FGSM for instance, this bounding box would consist of a perturbation space within an l_∞ ball. The intuition behind this definition of robustness lies in

how we should expect a model's output to remain constant in a neighborhood around any sample that can be drawn from the input space. As a result, *"the larger this neighborhood is for all inputs within the natural distribution of samples, the more robust is the DNN"* [2]. Naturally, we cannot consider all inputs here, as that would mean that the ideal robust classifier would be a constant function, which, albeit very robust, is not very relevant.

Papernot et al. [2] provide some broad requirements on designing robust defenses against adversarial perturbations. Although the original work focuses more on Deep Neural Networks, these guidelines will serve as important foundations to evaluate and assess the costs of implementing a robust model. Moreover, we can eventually derive valuable evaluation metrics from these rules, as we will discuss further on.

- *Low impact on the architecture*: we should aim to develop a solution that keeps the architecture as close to the original as possible, since introducing and developing new architectures not present in literature demands a careful study and benchmarking of how they behave.
- *Maintain accuracy:* although we may eventually sacrifice some accuracy on clean samples (inherent trade-off discussed in [1, 11]), the final model performance should remain comparable to a naive model in a non-adversarial setting. This discards solutions that rely on heavy regularization, as they are most prominent to cause underfitting.
- *Maintain speed of the model:* the developed solution should not worsen the performance of the model when it comes to classifying a sample. On the other hand, a higher overhead on training time is comparably more acceptable as it can be seen as a fixed cost.
- *Defenses should work for similar samples in the training dataset:* the ideal model should be able to classify samples within a neighborhood consistently. Samples that deviate greatly from the input training set are less interesting since they can probably be easily spotted by a human in the loop.

In the next subsections we explore some of the proposed defense methods in the literature, keeping in mind these criteria.

2.1.4.A Distillation as Defense

In the context of neural networks, attacks were mostly based on gradients that allowed an attacker to generate an adversarial sample that would cause the largest loss. These gradients allow one to infer the sensitivity of the network to each of its input dimensions. If these gradients are extremely high, that is, if the network is too sensitive to minor input changes, generating adversarial attacks becomes easier, as a small input perturbation can cause a drastic change in the output. So, according to Papernot et al. [2], in order to protect a (differentiable) model against adversarial samples, a promising step would be to

reduce the amplitude of these gradients. By smoothing the model, we aim to help it generalize better to samples not present in the training set, and thus make it less susceptible to adversarial attacks that capitalize on these "uncharted" input spaces [2].

From these premises and previously exposed attacks, Papernot et al. propose Distillation as Defense. Distillation as Defense is one of the strongest defenses against adversarial attacks in neural networks and can lower the success rate of constrained perturbation space methods (such as FGSM) from 95% to 0.5%. This technique can be applied to any feed-forward network with just the additional overhead of a single re-training step [2, 20].

In order to distill a neural network we start by performing standard training of a model with similar architecture. However, instead of just computing the softmax as an activation function while training the network, we replace it with a smoother version by dividing its logits by some constant T (temperature) ideally larger than 1, since otherwise it results in a less smooth network. We then use this network to produce soft training labels with which we will be training a second network. The main points we aim to achieve with this procedure are that by using the soft labels we 1) convey hidden knowledge learned by the first network and 2) avoid over-fitting to any of the training data. Since neural networks are highly non-linear models, when we prevent over-fitting we are trying to remove the blind spots that might occur from training. The authors of [2] argue that it is in these blind spots that adversarial examples lie. However, it was later shown that defensive distillation does not actually remove adversarial examples. Adversarial examples exist not because of blind spots in a highly non-linear network, but rather because of the locally-linear nature of neural networks (adapted from [20]).

2.1.4.B Adversarial Training

Lastly, Szegedy et al. [6] propose *adversarial training* to increase the robustness of models against adversarial examples. Essentially, adversarial training consists of a slightly more sophisticated method for data augmentation. It consists of generating new, adversarial samples from a set of victim examples while training, which in practice results in expanding the dataset used to train the model. The premise lies directly on the problem we are trying to solve presented in equation 2.2. Adversarial training emulates this "adversarial loss" by mimicking an adversary in the loop that is trying to maximize classifier loss (inner maximization loop), so that we are now solving for the θ that minimizes the inner maximization. A core difference between adversarial training and standard data augmentation techniques is that the latter expands the training set using transformations that are likely to occur naturally within the training set (or in nature). The goal with adversarial training on the other hand is to expose flaws in how the classifier models its decision function; thus the generated example are less likely to occur naturally but rather target these specific sub-spaces of the input spaces [18]. So, when training a model, we can iteratively generate more adversarial examples as it updates. The primary reason for this to be done in

train time and not prior is that the concept of an adversarial example depends on the current parameters of the model, as we will see in a moment. An adversarial example generated prior to training is, most likely, very different to one generated post training. The authors show that by training the model using a mixture between clean and adversarial samples the model can thus be regularized [6, 18].

This approach becomes increasingly expensive as the volume of data grows. In many modern-day machine learning applications, such as credit card fraud detection, classifiers require massive amounts of data, and individually generating an adversarial example for each transaction could significantly hinder training times. As a result, Kurakin et al. [19] propose adversarial training to be done using batches. Records are grouped into batches containing both clean and adversarial examples in each training step [19]. Moreover, in each batch the authors can fine-tune the number of adversarial examples and their relative weight using the following objective function:

$$\tilde{J}(\theta, \mathcal{B}_{clean}, \mathcal{B}_{adv}) = \frac{1}{(m-k)+\lambda k}\Big(\sum_{(x,y)\in\mathcal{B}_{clean}} J(\theta, x, y) + \lambda \sum_{(x,y)\in\mathcal{B}_{adv}} J(\theta, x, y)\Big) \tag{2.9}$$

where m corresponds to the number of examples in the mini-batch, k is the number of adversarial examples therein and λ the weight of adversarial examples. Essentially, these are fixed hyper-parameters that can eventually be fine-tuned, though the authors used $m = 32$, $k = 16$ and $\lambda = 0.3$. We are trying to optimize θ in order to minimize $\tilde{J}$, where $J(\theta, x, y)$ is the loss of classifying some input x of label y.

2.2 Adversarial Robustness on Tree-Based models

In the discussion presented above, the generation of adversarial attacks could leverage the inherently efficient optimization mechanisms behind neural networks. Using the network's gradient to find the optimal perturbation to the input was a core step in many of the proposed algorithms [10, 18, 22]. Naturally, this puts a lot of weight on whether the target model is differentiable. For this reason, the research on crafting adversarial examples for tree-based models is comparatively poorly developed. Indeed, generating strong adversarial attacks that target non-differentiable models such as decision trees or their ensembles thus demands different approaches.

One could argue that the decision trees tend to be constant within a certain neighborhood could lead tree-based models to be less vulnerable to adversarial attacks. Although such models are in fact less vulnerable to gradient-based attacks, current literature shows they are nonetheless vulnerable to adversarial samples. It seems that tree-based models like Gradient Boosted Decision Trees are just as vulnerable as neural networks [7, 8].

There seem to be fewer papers covering adversarial robustness in tree-based models with tabular data compared to image classification attacks against neural networks or other differentiable models.

Nonetheless, we start this section with a brief description of the additional concerns raised when perturbing tabular data. We then structure the last few sections as a short summary of attack and defense strategies in the literature within this context.

2.2.1 Perturbations to Feature Engineered Tabular Data

We start by formalizing the additional considerations raised when designing adversarial attacks in the context of tabular data. Particularly, we pay special attention to the **imperceptibility**, **feature importance**, and **domain constraints** of each feature that this setting imposes. Moreover, we note the importance of not only the input space of each of these features, but whether or not they can be manipulated by an attacker: that is, if they are **editable** or not.

Unlike images, the inputs in tabular data not only span heterogeneous domains, but also frequently contain derived features. These are obstacles towards defining a perturbation space. We can no longer make use of already existing l_p norms since the input space is no longer homogeneous, with each feature possibly having different data types and meanings. To define a perturbation ball we now need to customize a norm that accommodates non-numerical features. Moreover, the fact that some features are generated implies they cannot be directly manipulated. This "non-editability" of the derived fields might seem to offer us some safety from malicious perturbations in those fields. However, although they cannot be directly edited by attackers, there are still possible vectors of attack from indirectly manipulating said fields.

According to [23], there are two important aspects regarding perceptibility that differ between tabular and image domains that we must consider. Unlike pixel data, tabular features are not interchangeable, in the sense that different features carry much different meanings. Moreover, detecting perturbations in images is very different from detecting perturbations in tabular domains. Interpreting and reading tabular data often requires expert intervention in some contexts [9]. Should it come to an expert individual to make a decision on classifying a sample, it is to be expected that they base their decision in a smaller sub-set of features. Ergo, when measuring the imperceptibility of an attack, it is important to assign heavier weights to frequently checked features and vice-versa.

Another important aspect to consider is whether the final perturbed sample "makes sense". In image classification, as long as each pixel absolute value was within the encoding range (typically from 0 to 255, or between 0 and 1), there were not many hints that could be drawn to spot any malicious tampering. Unlike image data, tabular data has more domain constraints. For example, in some cases an age of 100 might already be considered suspicious. Each individual feature must obey its respective domain, in the sense that they must be bound by human intuition and perception.

Besides the intuitive constraints imposed by the meaning of common features such as age and income, there are particular fields that are either expensive to manipulate or simply cannot be altered.

Moving averages, volume in the last months and number of recent transactions are all good examples that can, in fact, be manipulated but require more effort than simply changing features such as transaction amount for example. An approach followed by [9] is to enforce the editability constraints by defining a vector of editable features and passing it to the adversarial algorithm.

Let us couple these additional concerns with the previously formalized notions of adversarial attacks. Considering an input x of dimension k with label y and a mapping $h_\theta : X^k \to \mathbb{Y}$ from the input space to the output space, that is $h_\theta(x) = y$. We are yet again interested in finding the smallest perturbation δ that causes a change in the output. However, the meaning of "smallest" must now be conveyed using a custom norm that considers the importance, as well as the domain and editability of each feature. Let us then also consider a mapping $m : \Delta \to [0, 1]$ which maps a perturbation to a perceptibility score between 0 and 1. Moreover, we are only interested in producing "valid" (i. e., that pertain to all the relevant domains) adversarial samples. We thus define $A \subseteq X^k$ as the set of valid samples. Thus, our goal now becomes:

$$\underset{\delta \in \Delta}{\operatorname{argmin}}\, m(\delta) \quad \text{s.t.} \quad h_\theta(\tilde{x}) \neq y \quad \text{and} \quad \tilde{x} \in A \tag{2.10}$$

This formulation was originally proposed by Ballet et al. [23]. It is important to also note how the mapping m must consider an explicit feature importance vector. We denote by v the vector $v = [v_0, \ldots, v_k]$ with the weights corresponding to each feature where $v_i \in [0, 1]$. The previously stated mapping can thus be described as the l_p norm of the feature importance weighted perturbation vector, such that:

$$m_v(\delta) = \|\delta \odot v\|_p^2 \tag{2.11}$$

where $\odot$ is the element-wise product. We highlight why using standard l_p norms between the crafted adversarial sample and the original input does not suffice as well in the context of tabular data. The l_2 norm considers the Euclidean distance between the clean and adversarial samples. This not only might not be suitable for certain features in the dataset, but also ends up disregarding larger perturbations on certain fields. On the other hand, when we try to minimize the l_∞ norm, we are expressing a preference in many small perturbations over one big perturbation. Cartella et al. [9] argue that neither of these norms fully satisfy the needs of designing imperceptible adversarial attacks in tabular data.

From the ideas presented, [9] go one step further and design a specialized custom norm fit to the context of fraud detection. Besides considering "just" the importance of each feature, there are also additional boolean parameters discriminating whether or not a feature is checked by an expert. This adds another dimension to how we can think of the perturbation space, as the adversarial generator should be interested in perturbing highly important features but that are not checked by an expert. On the other hand, Cartella et al. [9] also consider penalizing the generator for perturbing unimportant

features that are not checked. From these premises, the previously presented m norm becomes:

$$m_v(\delta) = \|\delta \cdot (\alpha \cdot h + \beta \cdot [(1-h) \cdot (1-v) + h \cdot v])\|_p \tag{2.12}$$

with h being whether the feature is checked, α, β the weights of the check and importance vectors respectively, and $\|\cdot\|_p$ any p-norm.

We also highlight the need to address how to compute the similarity between categorical features. From the comparative evaluation presented by Boriah et al. [28], there are measures specifically targeted at computing similarity between matches (depending on the rarity of values for instance) or between mismatches. Moreover, the authors also discuss how there are pairs of measures that complement each other, in the sense that where one hinders, the other excels. In broad terms, most of the evaluated metrics rely on the frequency of the value of each feature to calculate the similarity between instances.

In the conducted study in [28], one of the mentioned metrics to measure distance between categorical features corresponds to the Inverse Occurrence Frequency (IOF). IOF is based on inverse document frequency in the context of information retrieval [28, 29]. IOF is given by the following formula:

$$\frac{1}{1 + \log frequency_a \cdot \log frequency_b} \tag{2.13}$$

From this we infer the range captured by IOF is $\left[\frac{1}{1+(\log \frac{N}{2})^2}, 1\right]$. The minimum is assumed when the values assumed by feature a and b each occur $\frac{N}{2}$ times, where N is the number of records. The maximum is achieved when both value a and value b each occur only once.

2.2.2 Attacking Tree-based models

We end this literature review with two sections covering the state of the art on attacks and defenses against tree-based models, respectively.

Zeroth Order Optimization - A promising approach that has already been mentioned which does not depend on the gradient of the model being attacked was presented by Cheng et al. [8]: the Zeroth Order Optimization attack. As mentioned, the setting in which this problem is tackled is extremely challenging: a black-box model in which the attacker only has access to a hard-label. Zeroth Order Optimization [30] is a gradientless optimization approach that assumes only oracle access to given queries on the model.

Essentially, the algorithm proposed consists of a binary search in a direction θ in which we iteratively adapt our input until there is a change in the oracle output. The search direction is firstly computed using the difference between our source sample (the sample to be perturbed) and a sample from another class. This search direction is iteratively updated using a zeroth order optimization algorithm: the Randomized Gradient-Free method [25, 26]. Upon convergence, the authors then use this direction to query the

black-box model until the hard-label output changes. Once this change is found, Cheng et al. perform a binary search to narrow down the point at which it occurs. Remarkably, this does not rely on any model gradients and is thus particularly fitting to non-differentiable models. Ultimately, the authors show that by using around 70,000 queries it can be concluded that GBDT are also vulnerable to adversarial attacks whilst maintaining perturbations comparable to those in a white-box setting.

LT-Attack - Zhang et al. [31] propose an iterative approach in which we start with an initial adversarial example x' s.t. $h(x') \neq y$ and greedily moves closer to x, a clean sample of label y. To find this initial sample, the authors start with a "victim" sample, add a random vector drawn from a normal distribution and perform binary search using the previously mentioned method proposed by Cheng et al. [8]. Then, both the optimal initial adversarial example and the victim sample are mapped to a discrete input space and their distance calculated. This procedure is repeated an arbitrary number of times and the leaf tuple with smallest distance is chosen. Once we have an initial sample, a new adversarial example x'_{new} is iteratively chosen within a small neighborhood around x' that has the minimum p distance to x. This can be formally defined using the following update rule

$$x'_{new} = \underset{\hat{x}}{\operatorname{argmin}} \|\hat{x} - x\|_p \quad \text{s. t.} \quad \hat{x} \in Neighbor(x'), \quad h_\theta(x) \neq y \tag{2.14}$$

where the stop condition is if x'_{new} is not less perturbed than x'. However, the major difficulty in this method is to define a neighborhood so that Eq. 2.14 can be efficiently solved.

In most of the existing attacks, Eq. (2.14) is solved by a continuous optimization algorithm where $Neighbor(x')$ is a small l_p ball around the current solution x'. Besides non-differentiablity, one of the major difficulties of attacking tree ensembles is that the model prediction will remain unchanged within a possibly large region containing x', so traditional continuous distance measurements are not suitable here. To handle these difficulties Zhang et al. [31] discretize the continuous input space into a "leaf tuple" space. In that sense, the algorithm iterates through this discrete input space; the proposed attack "*iteratively optimizes the adversarial leaf tuple by moving it to the best adversarial tuple within a small neighborhood*".

2.2.3 Adversarial Robustness for Tree-based models

Robust Splitting - Chen et al. [7] present an approach based on augmenting the standard tree training procedure in order to increase the robustness of tree-based models. A visualization of the proposed method can be seen in Figure 2.1. The key observation is that in a case like this, a horizontal split might result in a higher accuracy but, in turn, it can very easily be compromised as it is vulnerable to adversarial attacks. On the other hand, a vertical split suffers a slightly lower accuracy but is much more robust if we consider perturbations of limited l_∞ norm. From a high level, the proposed method

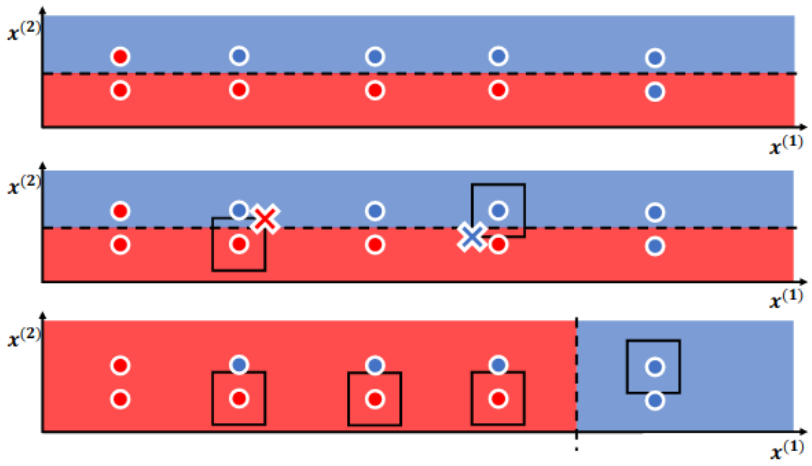


Figure 2.1: Visual example of robust splitting (taken from Robust Decision Trees Against Adversarial Examples [1])

considers the possible perturbations in the training data points and minimizes the losses over the worst case perturbations, that is, considering every adversarial example given a clean input. To achieve this, Ilyas et al. [32] and Tsipras et al. [1] classify features into two categories: robust and non-robust. The figure provided by the authors shows this division is straight forward in tree-based models, x(1) being a robust feature and x(2) a non-robust feature.

Adversarial Training - The previously mentioned approach on adversarial training proposes the minimization of training error while simultaneously expecting an adversary to maximize it. This formulation can be represented as:

$$\min_{\theta} \left(\max_{\delta \in \Delta} J(\theta, x + \delta, y) \right) \tag{2.15}$$

In the previous sections, this problem lent itself to standard optimization techniques, since the model being targeted was differentiable. This, in turn, made the inner maximization problem much more tractable, as generating adversarial examples greatly benefits from such approaches. As a result, this formulation no longer holds when it comes to developing adversarially robust tree-based models, as we can no longer rely on these techniques to generate examples [33].

In order to solve this problem, an option would be to make the inner loop subject to standard optimization techniques. Calzavara et al. [33] thus propose to achieve this by defining a finite set of perturbations and switch the maximum function to LogSumExp (LSE): a differentibable approximation of the maximum. For the former, Calzavara et al. highlight that in the context of tree-based models we can pre-compute the thresholds in which we will split each feature to perform classification. The authors then capitalize on this to define a finite set of perturbations without loss of generality (here represented by $\mathbb{A}(\Delta)$). So, the formal definition of adversarial training with these considerations now becomes:

$$\min_{\theta} \left(\log \sum_{\delta \in \mathbb{A}(\Delta)} e^{J(\theta, x+\delta, y)} \right) \tag{2.16}$$

Such a formulation allows for the use of optimization strategies. After testing against the Census Income dataset, the authors verified that this approach resulted in a better trade-off between robustness and accuracy compared with just standard training.

This concludes the literature review within the scope of adversarial robustness. We highlight a few ideas that can be drawn from this research, mainly:

- Developing an adversarially robust model is often linked with developing strong attacks.
- Common approaches towards defining adversarial samples rely on defining a perturbation space around clean samples.
- In image domains we might define these spaces using a standard l_p norm, whereas in tabular domains there is a need for a custom one.
- The differentiability of the model w.r.t its inputs helps finding its vulnerabilities.

3

Methods

Contents

In this chapter we detail the methods developed in this thesis. In particular, we aim to provide a clear definition of the perturbation space we are going to use to generate attacks, the strategies that search through this perturbation space, and finally how we leverage these strategies to adversarially train our models. We hypothesize that this methodology will produce a model robust to a realistic set of perturbations, within the context of fraud detection. With this in mind, it is crucial to develop a perturbation space that accurately captures the perturbations that can be done by fraudsters.

Designing efficient attacks demands a careful definition of the perturbation space and search algorithm. Naturally, we are interested in devising attacks with the highest possible success rate under the lowest possible budget (that is, smallest possible perturbations). With this in mind, the attack strategies to be developed in this thesis always search through a constrained perturbation space. In other words, we must try to find the perturbation that results in the largest possible model score loss without going over an enforced norm constraint. Moreover, we consider we should generate attacks representative of the real world adversarial adaptations of fraudsters. This implicitly denies us access to the model we are attacking if we consider the average fraudster does not know the model they are trying to bypass.

In Chapter 2 we discussed three possible settings for attack generation: **black box**, **white box** and **white box proxies**. In this work, we attack a black box model in a partial observability setting, in the sense that we assume a fraudster has access to the scores produced by the model. We find this setting provides non-trivial results worth exploring on their own. However, we consider it would also be interesting to attack models in other observability settings, and thus leave white box and proxy attacks as future work.

Besides the setting in which the attack will take place, it is important to define a set of perturbations. Like previously seen approaches, the adversarial samples that can be generated are usually constrained using an unstructured perturbation space. Handling tabular data on the other hand demands that we use a specialized perturbation space tailored to each feature and its respective meaning. Moreover, we can include certain domain knowledge in the definition of this space. As mentioned, our attack strategy explores a subset of this perturbation space, where the size of this subset is conveyed using a norm constraint. We detail the design of such a norm in Section 3.4. We experiment various sizes of perturbation sets. In other words, we explore different scenarios where an attacker has different budgets to generate their adversarial samples. With these experiments we are interested in studying how the freedom of the attack influences its success rate and, later on, the robustness of the adversarially trained model.

This chapter is structured as follows: we start by discussing which perturbations can be generated (i.e. what actions can a fraudster take to commit fraud and what impact do they have on other features). We conduct interviews with risk analysts to aid this definition and present our proposal for a perturbation space. We also pay special attention to one particular perturbation consisting of temporally

shifting transactions. We then discuss our designs for perturbation norms that we use to quantify the magnitude of the attacks we generate. Afterwards, we explore the different strategies implemented to search through the perturbation space. We have developed a total of 4 strategies: random search, two variations of stochastic coordinate descent and greedy search. Lastly, we conclude this chapter by going over the planned steps to perform adversarial training.

3.1 Overview

We have defined two main goals in this project: to understand how robust our models are and to boost their robustness. In other words, we are interested in discovering how well an adversarially robust model deals with attacks, and whether it withstands new fraud strategies.

Despite the lack of literature in comparison to neural networks, this dissertation will focus on developing robust tree-based models. Tree-based models are a more common choice in fraud detection tasks since they are better suited to a mix of numerical and categorical data and empirically have been found to perform well in comparison to deep learning approaches [34]. Even though deep neural networks have been making headways in tabular data, ensembles of tree based models are still the more common choice in many use-cases with tabular data [35, 36]. If we are interested in developing a model resistant to real fraud, we should focus on algorithms that are suited to this domain.

The high-level overview of the methodology for this project is as follows.

1. We start by collecting additional information that will help us define a more concrete and realistic set of perturbations. This information was obtained by interviewing risk analysts, to get a better idea on how fraudsters typically bypass fraud detection systems.

2. Leveraging the domain knowledge collected from these interviews, we then define a perturbation space suitable for the context of fraud detection. We highlight that although the types of perturbations that can be done in this context are dataset independent, their actual implementation varies from dataset to dataset.

3. We then develop baselines according to common practice in the context of fraud detection using the selected datasets. To do this, a baseline model is trained using historical data and then it is evaluated on a dataset with and without adversarial samples. This classifier is later compared against a model trained to be adversarially robust.

4. We explore the perturbation space to generate adversarial attacks. This step entails three main stages: designing a norm to express the size of a given perturbation, defining and implementing an efficient method of searching through the space (that is, *an attack strategy*) and finally benchmarking, in which we compare the effectiveness of each attack strategy. We detail the procedure

followed for each of these stages in this chapter. We craft a range of attacks that can bypass some of the difficulties posed by tree-based classifiers by treating the setting as a black-box under partial observability, although we still aim to craft attacks specifically targeted at such a class of models in future work.

5. Finally, using the attacks from the last step, we train the models to be adversarially robust. We experiment adversarial training in different conditions and assess the results. We are interested in inferring which parameters yield the highest robustness as well as the best adversarial to clean performance trade-offs.

We now briefly discuss some considerations over the models to use for this project. The predictive power of ensembles of decision trees comes from the principle that many weak learners can be used together to create a strong model. In boosted trees, we can consider we start off with a single weak learner. In each boosting round, we fit another weak learner which should improve the performance of the joint model in its previous round. We create this new estimator by using the gradient of the loss function w.r.t. to the output of the model in the previous boosting round. A popular implementation for gradient boosted trees is proposed by XGBoost [37]. LightGBM [38] further improves this implementation by reducing the number of data points that are scanned when boosting trees, effectively reducing computational cost. For this thesis, we use LightGBM models.

3.2 Primer on feature engineering concepts

We highlight some key concepts when discussing tabular data that has undergone feature engineering pipelines. We distinguish between raw and engineered features, and discuss how the latter are obtained. Feature engineering essentially takes in a set of (usually raw) features and adds features that are computed from the existing ones in order to extract some signal that may be relevant to the model. We break down the most relevant concepts in play when other features are generated. There are two main types of derived features: profiles and mappings. Profile features, consist of time aggregations over a time window for a given group-by value. A very practical example of a profile would be the count of transactions sent out by a card in the last 24 hours. Here the set of columns used to group data would be the card, and the time window would have a duration of 24 hours with no delay, computed in real time. Another example where profiles are calculated with aggregated features would be the total amount (sum) of money sent out by a card in the last hour. Profiles have essentially four main properties: the aggregation operation itself, a set of grouping columns that defines the group-by value, a time window and, in some cases, the feature to be aggregated.

- The aggregation operation defines how data is grouped. Common aggregation operators consist

of count, sum, average, among others.

- The set of grouping columns establishes how we will "group" data. There can be more than one column (e.g, we could be grouping by each combination of card and merchant). Common grouping entities in the fraud detection domain are cards or customers, depending on the use-case of the dataset.
- The time window defines which transactions are included in the group to compute these profiles. Time windows can have a number of properties, the most relevant being the duration, the delay and whether they are computed in real time or batched (batched profiles are only computed at the end of each specified interval, for example, at the end of each day).
- The aggregation operation in a profile may or may not involve other features of the events that fall into the the window. For example, a count profile will not involve any additional features (it will only count the number of events in the window), whereas a profile containing an average of a feature will use the values of that feature (e.g., average amount sent by a card within a given timeframe).

Occasionally, profiles can also have filters. Profiles with filters are only computed if the transaction obeys certain requirements. For instance, a sum of the euro amounts sent by a card can only be computed if the euro flag in a given transaction is set to true. Profile features posed additional challenges in the generation of adversarial samples in more than a few aspects as we will discuss in further detail in this chapter.

Besides profiles, through feature engineering we can also generate additional features through mappings, which, as the name suggests, consist of applying a specific mapping function to one or more features. A simple example would be the generation of a feature that is the logarithm of the amount sent in a transaction.

3.3 Perturbations

The first step in attacking a model trained to classify a specific dataset is to define the perturbations that an adversary would be allowed to make against a data instance. The definition of this perturbation space lays the foundation to develop attacks, employ adversarial optimization techniques and will allow us to design a norm that expresses the magnitude of a perturbation. Much like what was presented in Chapter 2, this space depends heavily on the dataset under attack. To reiterate, in a tabular dataset with engineered features, developing a set of valid perturbations is a process that must be done individually for each dataset.

Besides knowing the features that compose the dataset under attack, it is also important to consider the allowed means of attack within the domain. In other words, in the context of fraud detection only

a handful of features can actually be perturbed by a fraudster - and it is precisely this subset of features we should consider when developing the perturbation space. However, the dataset that is used to train a model is not exclusively composed of the raw features that entail a single transaction. Indeed, feature engineering pipelines and procedures are commonplace, enriching each individual record with derived information that might be useful to the model. This demands that our perturbation space definition, through dataset analysis, encompasses not only a careful selection of *which* raw features are "perturbable" by a fraudster, but also *how* these perturbations propagate to other engineered features.

It is also important to note that the inverse could also be stated - that is, instead of following a top-down approach for perturbing features, we could also model these perturbations through a "bottom-up strategy". Instead of perturbing the raw features to generate a successful adversarial sample, we could also consider perturbing directly the generated features and reverse engineer what combination of raw features such a perturbation would demand. We explore the top-down approach in this project, though bottom-up is also a promising direction to explore in future work. The former has the advantage of being easier to interpret and implement, and also allows perturbations to be done in a reduced subset of the whole feature space. Moreover, a top-down approach more closely mimics the actions that can be taken by real fraudsters, making it more realistic. The main drawback of such a strategy however is that it is not obvious what will be the impact of perturbing a certain feature when using the full feature space. On the other hand, a bottom-up approach would allow us to have more control over the "realized" perturbation at the expense of a harder to grasp interpretation of the perturbation space.

3.3.1 Perturbation space design

In this section, we start by defining the set of perturbations we wish to allow for the types of dataset we will study. Then we discuss how a perturbation to a raw feature reflects itself in a derived feature. This will later allow us to map perturbations from the raw to the full feature space. In order to achieve this, we analyze **fraudulent behavior patterns and related perturbations**. In this step, we are looking to grasp how each individual raw feature can be perturbed depending on the concept it represents based on what are common approaches followed by fraudsters, as well as which features can be manipulated.

Given a dataset to study, it is important to collect domain knowledge to determine the ways a fraduster can act in to commit illicit activity. In order to do this, we conducted interviews to gather such domain knowledge from risk analysts. These interviews indicated several important aspects of the fraudster's workflow:

- When fraudsters try to adapt previous transactions that were flagged as fraud (i.e. perturb samples) there is not necessarily a single "direction" in feature space in which they perturb them, instead they proceed mostly through trial and error random searches.

- Fraudsters have also been known to automate many attacks in order to effortlessly uncover any blind-spots that may appear in the meantime.
- Features such as names and addresses are randomly generated whenever possible.
- Interviewees have also reported fraudsters collaborate between them through forums and messaging apps, which lets them easily exchange information such as new fraud types or stolen cards.
- The black market also allows them to buy and sell information about stolen credit cards.

Furthermore, from the interviews we also hypothesize that fraud changes because fraudsters adapt. This might be because there are new and better ways of committing fraud (cheaper or easier), or because some change in the fraud detection system implies that previous fraud strategies eventually stop working, or the available technology to commit fraud evolves. Analysts reported that Machine Learning (ML) systems reportedly result in slower changes in fraud patterns whereas rule-based systems are faster. This is because typically rule systems can be adapted much faster and are changed more often, which in turn lead to more fraudster adaptation. In summary, given the information collected in the interviews, the main highlights on which features can usually be perturbed are:

- **Changing the product being bought** - though it is only applicable to the merchants and payment processors, fraudsters can target less risky products when committing fraud against merchants and try different combinations of baskets. Fraud detection systems typically have features identifying products commonly bought by fraudsters (i.e. that can easily be resold), and avoiding these products might better their chances.
- **Changing card** - if we assume fraudsters have an array of stolen cards at their disposal, we can hypothesize that, at some point, fraudsters give up on trying to commit fraud with a given card and move on to the next one. Therefore, changing card is a possible perturbation.
- **Changing the amount of money** - although it makes more sense in a banking use-case (since clients cannot change the amount of money they pay for a given item in merchants), it is nonetheless possible to buy more or less amount of a specific item (or changing items altogether), effectively changing the amount involved in a particular transaction.
- **Deliberate typos in free form fields** - from the interviews, we gather this is a particularly frequent way of altering a transaction. Fraudsters have greater leeway when filling out strings in forms since often no specific format is enforced. Some examples of free-text inputs are billing and shipping addresses or recipient names.
- **Changing Internet Protocol (IP) address** - interviewees reported the profile of a common fraudster to be reasonably tech-savvy. In that regard, it is sensible to consider fraudsters should have no

difficulty in manipulating and masking their own IP addresses (i. e. through spoofing) or purposely manipulating IP-related features, such as belonging to the The Onion Router (TOR) network or being a known Virtual Private Network (VPN) address.

- **Delaying or anticipating the transaction** - lastly, it is also easy for fraudsters to perform their transactions at any arbitrary time of the day or week. We reason they might want to wait for less suspicious traffic hours, try to submit a transaction with their new card as soon as possible or space out attempts more. We will simulate this vector of attack by injecting some (positive or negative) delay in the timestamp.

As for behavioral changes, the interviewees mentioned that, intuitively, a common adaptation would be to lower the amount involved in a given transaction. Regarding the perturbation of free-text fields, fraudsters have the means to randomize attempts by randomly generating fake addresses and names.

3.3.2 Perturbation pipeline

Based on the information collected on the previous section, these are the perturbations we consider. To sum up, we consider there are, in total, 6 possible perturbations we can apply to the dataset with different repercussions:

- **Amount perturbations** - we consider it is possible for a fraudster to change the amount involved in a transaction. In our setup, in order to perturb this feature we alter the amount (and all derived features accordingly) to some multiple between 0.02 and 5, meaning that amount perturbations are relative to the original amount and not absolute. In other words, the possible perturbations to the amount span from reducing it to 2% of its original to multiplying five-fold. In other use cases (besides fraud detection), similar numerical perturbations could be defined.

- **Temporal perturbations** - temporal perturbations are particularly unique, since we are only changing the transactions that are used to compute profiles. We allow a perturbation to span anywhere between one minute to one week - meaning a fraudster can wait or anticipate their transaction for a duration of time between those two.

- **Card resets** - from the interviews we gather it is frequent for a fraudster to have access to an array of cards. That being said, we consider that it is possible for them to start using a new card, in which case we assume it has never been seen before in the dataset. Hence, the natural implication is that any profile with “card” in its grouping entities is reset under such a perturbation.

- **Card switches** - besides switching card, there is also the possibility to switch to a card with different features, for instance, switching to a card issued by a different entity. We distinguish the

two perturbations: to simply change card but keep all the card-related features (for example, the identifier of the issuer that manages a given card) and to get a new card altogether with different card features.

- **Geolocation perturbations** - as previously mentioned, we expect fraudsters to be fairly tech savvy. For this reason, we consider it should be possible for them to manipulate their perceived geolocation.
- **Network perturbations** - similarly to the previous point, a tech savvy fraudster should be able to manipulate the perceived characteristics of their network, or simply changing them altogether. This, along with card switches, make up all the possible categorical perturbations we consider.

Essentially, we can group these into 4 steps: i) changing categorical and numerical features directly, ii) by changing when the transaction takes place (temporally), iii) by changing the amount involved in the transaction (assuming fraudsters can change amounts arbitrarily) and iv) resetting profiles by changing grouping entities (e.g., card),. These are the steps a victim sample undergoes in the perturbation pipeline. We include a representation of the pipeline in Figure 3.1.

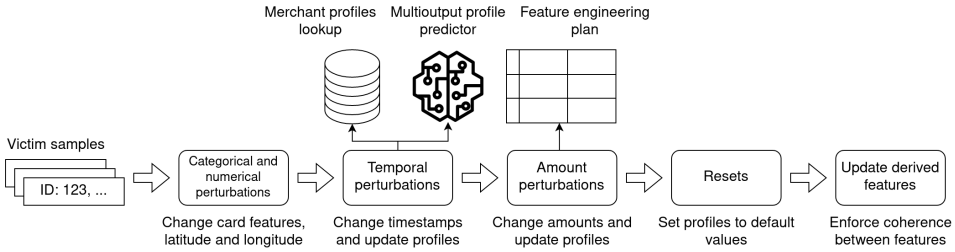


Figure 3.1: Diagram for the full perturbation pipeline

Each of these steps corresponds to its own, independent module. In that sense, different perturbations can be applied to different samples seamlessly - with one exception: card features. Card features are intrinsic to each card, meaning that in order to perturb them we would need to change the card. This is arguably possible for a fraudster, and is implemented in the step of "resetting" profiles. This brings us to the constraint that card-wise features can only be perturbed when there is a reset of the card feature.

3.3.2.A Categorical and Numerical Perturbations

In datasets in the fraud detection domain, categorical features are very common and can often be perturbed by a fraudster. For the sake of simplicity, we decided the only perturbable categorical features were those that are card and network related. Generally, when training ML models, categorical features are typically encoded in some way. We choose to use frequency encoding - each value is mapped to a number, the lower the number the more frequent the value is. As a result, we can directly perturb these

indexes rather than the real feature values. An important detail when perturbing categorical features however, is that they are often dependent on each other in ways that are constrained by the domain. For example, the distribution of Card Verification Value (CVV) flags might change dramatically for different card issuers. As a result, we cannot simply pick random numbers for each feature. Instead, in order to enforce this coherence between distributions is followed, we compute the frequency of each combination of categorical features and keep a table associating each combination with its relative frequency. Thus, in order to perturb card features, we only need to sample a random row of this table and replace the features in the original transaction accordingly. In general, it is possible to identify feature groups that are dependent among themselves, and for these groups we must sample consistent combinations. We hypothesize that categorical features related to the card (i.e. card-related features) are dependent among each other, whereas other features groups (for example, network features) are independent from them. As a result, these two groups can be perturbed independently.

We also consider it is possible for fraudsters to perturb latitude and longitude. The hypothesis is that fraudsters can spoof their own IP, which we believe to be the information used to track down the longitude and latitude of a transaction. Manipulating IP addresses thus effectively means manipulating their perceived location. In order to perturb latitude and longitude, we draw values according to a uniform distribution. We sample the geolocation from regions with enough density. The two are not perturbed independently, in the sense that it would not make sense to consider a fraudster to only manipulate their latitude and leave longitude unchanged. For that reason, the two features are treated as a single "geolocation" feature when perturbed.

3.3.2.B Timestamp Perturbations

One of the directions along which it is possible to perturb a sample is by delaying (or anticipating) a transaction. Even in a situation where only the timestamp is perturbed, such a perturbation can impact how much change a transaction undergoes in many ways due to the existence of profiles. Recalling that profiles are essentially aggregation operations done over a rolling window, changing the timestamp of a transaction could very well lead to it being grouped with other transactions by shifting the window and thus have a dramatic impact on its profiles. In order for us to generate adversarial examples through timestamp perturbations, we must ask ourselves: what would have happened if this transaction had occurred at a different time? An exact solution would imply recomputing the profiles over an copy of the dataset, with the only difference being that each victim sample would have a new timestamp occurring at the new, perturbed time. This would always lead to correct results and a valid, realistic adversarial sample. This solution is nonetheless computationally very heavy, hence not feasible, since it would be fairly slow to run in the inner adversarial optimization loop for many transactions. We therefore consider the following approaches:

- **Data-driven** - high-volume profiles (i.e. merchant profiles) are estimated using a look-up table that stores their mean value over time bins.
- **Model-based** - low-volume profiles are estimated by multiple models (conceptually, a multi-output classifer models the changes in profiles after observing a given delay).

The idea behind the **data-driven approach** is to leverage our own advantage of hindsight to estimate the new values for the profiles: we know how each profile changes over time for any given values of the grouping entities. So, if we want to estimate what is the total amount sent to a specific merchant in a specific datetime, we need only to perform a look-up operation of a transaction that occurred around that time. One might argue that this look-up operation can be extremely expensive, as sorting through 180 million records of data in the hopes of finding a "good enough" transaction is hard, and might not even be fruitful. However, this method can be further improved upon if we preemptively bin the profiles over time (say, calculate the mean for each profile using bins of one hour). This way we only need to figure out in which bin a perturbed timestamp lies and fetch the corresponding mean as suggested in Figure 3.2. This can make the look-up operation cheaper, as we just need to keep various combinations of profile values and their means over time (although these may still be many).

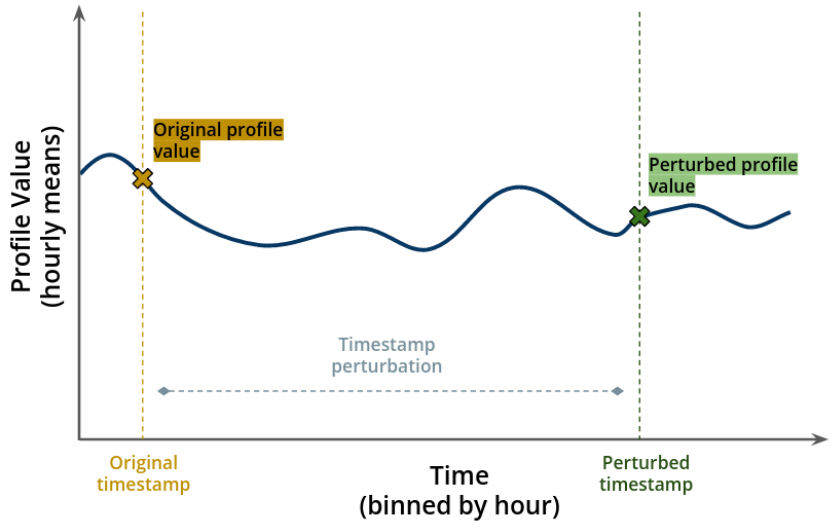


Figure 3.2: Diagram illustrating the intuition behind the estimation of high-volume profiles

The major assumption of this method is that the mean is a representative value of the profile, making it a good approximation. We hypothesize that this assumption holds for profiles calculated over a grouping entity with a large volume (as simply removing one transaction would be a minimal change to the total volume) but cannot hold for more specific grouping entities with comparably little volume (for example, cards). This reduces the applicability of this method to predict changes to a subset of features.

On the other hand, the idea behind the **model-based** approximation is to estimate how profiles

change over time by fitting a regression model specifically for this task. We thus propose training a multi-output regression model to predict the changes to each of the profiles. As inputs to this model, we use a dataset where each row corresponds to the original features, the original values for the profiles, the amount of delay that was injected in the timestamp and, as regression targets, the resulting profile values if the transaction had taken place in the perturbed timestamp. In order to build this dataset, we run the feature engineering on a dataset of raw fields where, for each chosen card to perturb, a single transaction was perturbed in its history. This is because perturbing multiple instances would leave us with a too-different history compared to the original and the input space for the regression task would have to be dependent on the number of perturbations for each card, and on their features (so that the prediction of the targets would be sensitive to those multiple perturbations). We adopt this subset of perturbations in this study and leave an analysis of the effect of more complex timestamp perturbations to future work. Once the dataset is built, we proceed to train an off-the-shelf multi-output regressor. It is important to note that we cannot use this model to predict merchant profiles, since we build this dataset through sampling by card. As a result, profiles that are grouped by merchant are broken in the sample used to train the model. Building a dataset through sampling by merchant leaves us with a dataset almost as large as the original, unpractical to train a regression model with.

Therefore, we use the second method to predict merchant profiles and the third method for the remaining profiles. The second method is trivial, given that it consists of a look-up table we can query to draw estimations from. We thus pay special attention to the third method. For the rest of this subsection we describe how we build a dataset which allows a model to predict them, and briefly explore the dataset that will be generated as well as its engineered counterpart. In the following chapter, we discuss in more detail how we train the models as well as the results obtained, and whether they are satisfactory: that is, whether the models trained can produce sensible estimates for the profiles, after the timestamp of a transaction is perturbed under an adversarial attack.

3.3.2.C Amount Perturbations

For these perturbations, we parameterize the perturbations with a scaling factor relative to the original amount. For instance, if the original amount is 100 and the perturbation is 1.3, the resulting amount is 130. This has the advantage of allowing us to adjust derived features, such as original currency amount, without having to deal with “exchange rates”. In this step we also update amount-related mappings, for example the log amount.

After perturbing the raw and other (non-profile) derived features, profiles must be updated to match the new amount values. Since this logic depends on each profile (update frequency, grouping operation, etc.), each type of profile requires a separate implementation for such an update. We highlight a few of the nuances considered when updating some types of profiles:

- Sum profiles can be updated by considering the difference in the entity they aggregate. For example, if the the sum amount of a victim transaction used to be 1200 when the amount was 100, the new sum amount must be 1150 when the new amount becomes 50.

- Likewise, mean profiles can be updated simply by considering the new value of the sum profile they use, divided by the respect count profile. In order to do this, we compute both count and sum profiles for every mean.

- Profiles that track maximum amounts pose a more interesting challenge. Consider we have a victim sample with some "old amount" and "old profile value", assumed before the perturbation is applied. Naturally, if the new amount is larger than the old value of the profile, the new value of the profile will correspond to the new value of the amount. However, when we perturb amounts to be below their original value, one of two things may happen: if the old value of the profile does not match the value of the old amount, we can safely assume the profile is "referring" to another transaction and leave it as is. However, if the old profile value and the old amount match and we reduce the amount, we might be perturbing the value that was being "tracked" by the profile. In situations like this, we resort to the respective count profile. If its value is 1 we can be sure that whatever new value the amount will assume, it will be the maximum amount. If it is larger than 1, we assume that the profile was tracking the other transactions, and leave it as it was.

- On top of different aggregation operations, it is also important to consider filters. To reiterate, profiles might only consider a subset of transactions. For example, an average of amount sent by a card to a subset of merchants will only consider transactions sent out to this set of merchants. For this reason, whenever we are applying a perturbation to profiles of a given victim transaction, we must ensure that this transaction obeys the corresponding profiles' criteria. In other words, we check if the filter imposed by each profile is met by the transaction. If it is, we update the profile; if it is not, we do not. This means that the same perturbation might lead to different profiles being updated depending on other fields of the transaction.

3.3.2.D Profile Resets

The meaning of resetting profiles is, essentially, restoring them to their initial value. This carries different meanings depending on the nature of each profile. Certain profiles may always be reset to 0, since they are not computed in real time and do not take into account the transactions as they occur, but only at the end of the day (in general). Real-time profiles, when reset, assume a set of default values. Sum amount, max amount and average amount profiles are set to the amount in the transaction, counts are set to 1, and standard deviation of the amount profiles are set to 0, should there be no filters on these

profiles. For example euro profiles should follow the above rules only if the current transaction handles euro as a currency. Otherwise, those same profiles should be set to 0.

Resetting profiles is a "binary" perturbation: we can either reset or not reset a grouping entity. We identified two resettable grouping entities in this dataset: card and amount. Starting with the simpler case, profiles grouped by original currency amount will naturally be reset whenever there is a perturbation to the amount in the previous step of the perturbations pipeline. Resets of the card feature allow us to also perturb card-specific categorical features, as hinted previously. Given this "binary" nature, reset perturbations are essentially a one-hot encoded dataframe containing these two entities as columns.

3.3.2.E Updates

Other feature updates are straight-forward, since most consist of ratios between profiles or derived features via mapping operations on a single row. After perturbing profiles and mapped features, the old values of profile ratios (or other secondary mapped features using such profiles) are no longer correct. This last step ensures that each resulting adversarial sample is coherent across all features.

3.4 Perturbation Norm

To be able to quantify the magnitude of a perturbation, we now define suitable perturbation norms. We discuss two norms in order to capture two distinct notions: how much features changed (in regards to model sensitivity) and how "expensive" it is for a fraudster to perform a given perturbation.

The first norm, referred to as **model importance**, consists of directly multiplying the normalized feature importance of the target model with the absolute difference between the victim sample and resulting adversarial sample, normalized by each feature's standard deviation. In a feature space of dimensionality m, where we denote the importance of the feature at index i by f_i, we have that:

$$||x - \tilde{x}||_{MI} = \sum_{i}^{m} \left[\frac{f_i}{\sum_j^m f_j} \cdot \frac{\text{abs}(x_i - \tilde{x}_i)}{\sigma_{x_i}} \right] . \tag{3.1}$$

Naturally, this norm expresses that the more lax the norm constraint, the higher the chance that the resulting adversarial sample is successful, and expresses how "sensitive" our model will be to this perturbation. It is important to note that this norm can only be computed *after* updating the engineered features of the victim sample, since it uses the feature space after feature engineering. To compute it, for numerical features we consider the absolute difference normalized by each feature's standard deviation. In categorical features we just consider a boolean, depending on whether or not the feature assumes a new value after the perturbation. This "difference vector" is then multiplied by the feature importance weights in equation (3.1).

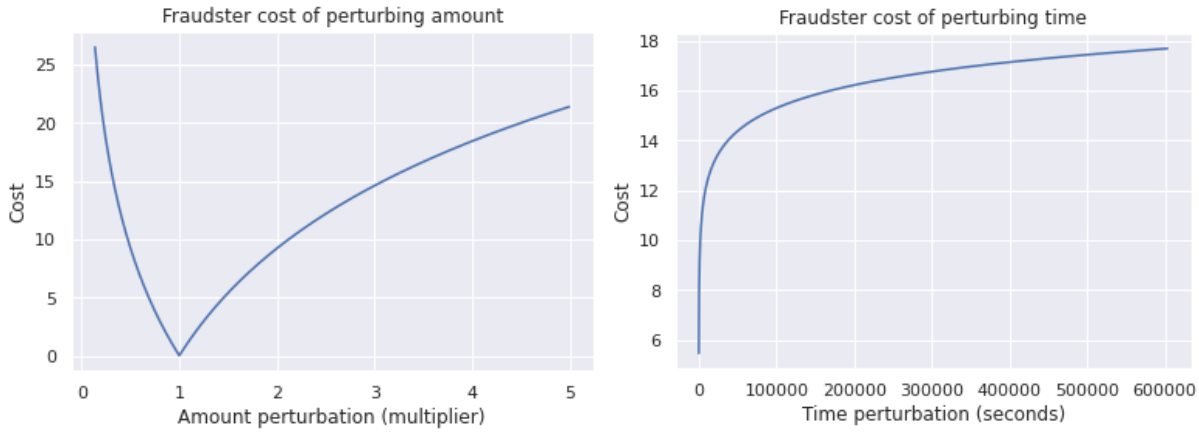


Figure 3.3: Cost functions for amount (left) and temporal (right) perturbations. The red dashed line in the right-hand plot corresponds to a delay of one day.

The second norm (denoted **fraudster cost**), on the other hand, aims to capture how expensive it is for a fraudster to apply a given perturbation. In order to achieve this, we instead consider the raw perturbation (that is, before feature engineering) multiplied with a custom set of weights for each vector of attack. This norm ends up being much more practical, since it can effectively be computed before the perturbation is *applied*. Moreover, we also consider this norm to be more relevant for the problem we are solving, because it relates more directly to the real costs that drive the fraudster's decisions when attacking the model.

The fraudster cost norm follows a similar design to its counterpart, yet considers way less features. We depart from the possible directions of attack (enumerated in list 3.3.2) and assign a cost to each raw perturbation (see table 3.1). Since most perturbations are discrete, this cost is fixed in most cases. For example, we assign a cost of 33 to resetting a card. The only two exceptions to this rule are temporal and amount perturbations. Here we found it best to design and employ a custom cost function depending on how large the alteration is. Naturally, waiting one week should weigh considerably more than waiting a couple of hours. Along the same line of thought, changing the amount to be five-fold should be more expensive than doubling it. We include a visualization of the designed cost functions in Figure 3.3. It is worth highlighting how the cost should vanish when the amount multiplier nears one (unperturbed amount). Moreover, lowering the amount involved in a transaction should also be costly. Conceptually, we can think this maps to a fraudster selecting a much less desirable basket of products to better their chances of bypassing the fraud detection system.

We opt to measure the size of the perturbations using almost exclusively the fraudster cost norm. We find that it lends itself to a more intuitive use, allowing us to compare attacks with higher or lower "budgets". Moreover, when employing optimization algorithms, we find it extremely useful to simply

Perturbation	Network	Geolocation	Temporal	Amount	Card reset	Card switch
Cost	3	4	18	26	33	49 (= 33 + 16)

Table 3.1: Cost of each perturbation. The cost of switching cards is presented as the sum of performing a card reset and a card switch.

compute the magnitude *a priori* of applying the perturbation. Additionally, in a completely black box setting, we can also assume that the fraudster has no access to the feature importance weights, as they are derived from the model - making the feature importance norm less realistic. As for the range of values that the fraudster cost norm can assume, we normalize it so that it goes up to 100. We consider that reducing the original amount to 1% of its original value is as "expensive" for the fraudster as it is to get a new card, since both of these operations represent a major change in how a fraudster must actually commit fraud. These represent the largest two perturbations, and from then we adjust (i.e. waiting one day is similar to tripling the amount involved in the transaction). We summarize our defined fraudster costs in Table 3.1.

3.5 Adversarial Attack Strategies

3.5.1 Random Search

In this section, we describe the procedure followed to generate attacks using the previously described perturbation space. We use random search as a baseline to assess the strategies we develop later on, all considering a black-box setting under partial observability. We can consider the random search to also be a black-box approach, in the sense that perturbations are generated randomly, and through the *a posteriori* model decision we can select which random perturbation is best. The data we attack consist of positives sampled from the test set. We attack 10 thousand positive samples.

As for the generation of attacks, we define one pass through the aforementioned pipeline to correspond as one iteration. Our random search will generate many adversarial samples, each independently from the other. Our goal is to have as many adversarial samples as possible, each with various perturbation sizes. At each step, we perform a Bernoulli trial for each perturbation direction to decide whether we perturb it. We kept each chance low, so that it is not very likely to perturb a transaction using all raw perturbations, which would skew the distribution of norms to large values. For instance, to perturb the timestamps of a victim sample of size M, we perform M Bernoulli trials, where it is more likely that the timestamp will go unperturbed. A similar strategy is followed for every other perturbation direction, with no dependence between each other with the exception of card features, for the aforementioned reasons. In this latter case, we use a 40% conditional probability to switch cards given that we are resetting a card, and a low probability (15%) of actually resetting it. We picked these values as a first step

and later verified that random search had a good coverage of the perturbation space. In other words, the randomly generated perturbations yielded a distribution of norms that was not skewed to either side. Hence, we did not tune these values further. We include the pseudo-code for the random search in the algorithm box 1.

Algorithm 1 Random search pseudocode

$\tilde{x} \leftarrow x$ ▷ Initialize adversarial sample to be the same as victim
for $d : directions$ **do**
  $u \leftarrow rand()$
  if $u < p_d$ **then** ▷ p_d being the probability of perturbing direction d
    $\delta \leftarrow \text{sample}(\Delta_d)$ ▷ Sample a perturbation according to direction d
    $\tilde{x} \leftarrow \text{apply}(\delta, \tilde{x})$ ▷ Update adversarial attack with the perturbation we just sampled
  end if
end for

In order to evaluate an adversarial strategy, it is important to consider both its success rate and the size of the perturbations it generates. The success rate can be computed by dividing the number of successful adversarial transactions over the total number of attacked transactions:

$$Success\ Rate = \frac{\#\ successful\ attacks}{\#\ generated\ attacks}\ . \tag{3.2}$$

An adversarial transaction is successful if any of the randomly generated attacks is successful. In other words, a transaction that was formerly labelled as positive, is now labelled as negative. We evaluate the success rate of random search under various norm constraints.

3.5.2 Black Box attacks under partial observability

This section covers the approaches followed to attack a model in a partial observability black-box setting: that is, having only oracle access to the scores produced by the model under attack. All in all, we find that the attacks generated with the developed strategies can beat the random search baseline for most of the norm constraints. In other words, for the same norm constraint, we find that some strategies can reliably find perturbations that yield a much higher success rate than a random search done with a budget of 500 total iterations.

3.5.2.A Stochastic Coordinate Descent

Given the high prevalence of discrete variables in the search space of the problem we are trying to solve, we opted to start by employing standard algorithms from discrete optimization literature, namely Stochastic Coordinate Descent (SCD) [39]. In essence, SCD consists of iteratively picking some random direction to attack, exploring its values and updating the current perturbation to the best value along the

direction if it is better than our current perturbation (i.e. results in a lower score). We repeat this for every direction until we conclude a directions sweep. We consider to have converged whenever we perform a full direction sweep which has yielded no improvements. Every time we pick a direction we generate a grid of possible values to explore within the bounds of our norm constraint. For categorical features we explore every possible value, for numerical features we generate a discrete grid of values.

This overview of the method indicates some possible variations that can be explored, namely regarding which perturbation to select from the generated grid (i.e., what is the criterion that defines the *best* perturbation). In this thesis we explore a greedy method of selecting a perturbation and a cost-efficient method.

Greedy approach The first variation of SCD (i.e. the greedy method) consists of selecting the perturbation that yields the lowest score from the generated grid. If we observe that this score is lower than our current best perturbation, we replace the latter by the new perturbation. We expect this implementation to converge fairly fast, while being prone to get easily stuck in local minima. Regarding how well we expect this strategy to explore the space, due to the tendency to evolve quickly to local minima, we foresee that the distribution of norms may not be very close to the maximum norm constraint and, as a result, the success rate may be sub-optimal.

Cost-efficient approach The key idea of this approach is to try to avoid getting stuck in local minima. We hypothesize that without any restraint, being greedy will ultimately just result in picking the most expensive perturbation, effectively leaving us no room to keep exploring. For this reason, we propose a method that instead optimizes for cost-efficiency: in each iteration, we select the perturbation that offers the best norm change to score change ratio. However, it is important to keep in mind both of these changes can be positive or negative. We observe that the score increases if our perturbation actually makes the victim sample more identifiable as fraud. On the other hand, although we expect most norm changes to be positive, it is still possible to observe a negative norm change. For example, consider we have already performed a full direction sweep in which we perturbed the amount. Now, during the next sweep, we can find some amount perturbation that effectively results in a smaller perturbation while still lowering the score. We thus identify four possible cases:

1. Negative score change and negative norm change - although probably rare, this is the optimal case: lowering the score (nearing us to a successful attack) and lowering the norm, leaving us more room to explore.

2. Negative score change and positive norm change - we expect most of our perturbations to fall along this category, where we increase the norm to lower the score.

3. Positive score change and negative norm change - this corner-case lets us “pay” in score change to have more “room” to explore by lowering the norm

4. Positive score change and positive norm change - this is the most undesirable scenario, where we are “paying” for our perturbation to be unsuccessful

Our order of preference corresponds to the one used in the above enumeration, 1 being the most preferable scenario and 4 the least. The main concern when adopting this approach is attributing a score that expresses not only this order of preference, but also establishing a sensible hierarchy within each case.

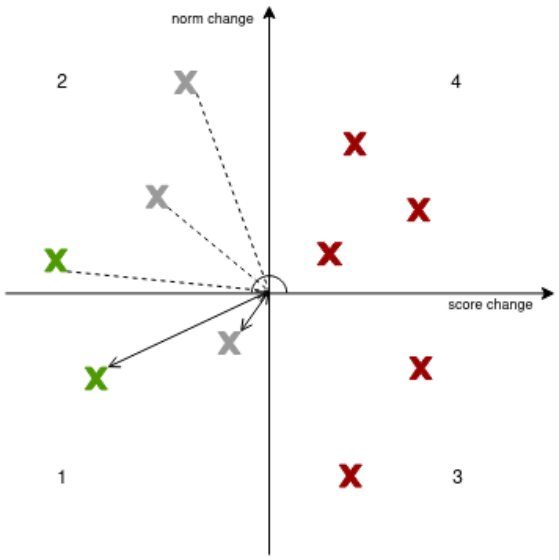


Figure 3.4: Quadrant demonstration

In the Figure 3.4 we plot the four possible cases in a two-dimensional space, with the x-axis being the observed score change, and the y-axis being the observed norm change. Consider we have just explored a grid of size 10 for a given victim, and each point corresponds to the observed changes for each entry in that grid. Note that in a purely greedy approach, we will always select the left-most point for the grid. Points in quadrants labelled as 3 and 4 will never be used for updates to the current perturbation as they increase the score, so we can immediately discard them. This leaves us with the two left-most quadrants. In quadrant 2, we will want the point with the **lowest** score change to norm change ratio. Note that we mention lowest since every ratio calculated in this quadrant will have a negative sign. Visually, this corresponds to the point with the largest “angle”. If there were no points on quadrant 1, the selected perturbation for this grid would have been the one marked in green. However, let us now consider quadrant 1. Cost-efficiency might not exactly be what we are looking for: note how the sample with the largest angle can very well barely affect score and norm. Now our objective should be to lower both as much as possible. We find a convenient way to convey this objective to be through the euclidean distance between the origin and the perturbation, effectively performing a linear

combination of the two changes. We also offer the possibility to "steer" this distance to be more or less greedy through some parameter α, where a higher α will give more weight to the score change and less to a norm change, and vice-versa. We have used an α of 0.6, giving the strategy a slight tilt towards score-lowering perturbations in this corner-case. The intuition is that this strategy is already considering cost efficiency in most cases, so should it come to a tie between model score and cost efficiency, we favor the choice of the perturbation that results in a lower model score. We remark that this parameter should be properly tuned by analyzing how well different values of α allow the strategy to cover the perturbation space and which yield the highest success rates. We leave this analysis for future work. The distance is calculated according to 3.5.2.A

$$distance(x, \tilde{x}) = \sqrt{\alpha \cdot \Delta(x, \tilde{x})^2_{score} + (1 - \alpha) \cdot \Delta(x, \tilde{x})^2_{norm}}\,, \tag{3.3}$$

where Δ_{score} and Δ_{norm} correspond to functions that compute the differences in the model score and norm of their arguments, respectively. We expect this version of SCD to offer higher success rates at the expense of more iterations and a longer time to converge.

3.5.2.B Non-Stochastic Greedy Search

Besides the presented strategies, we can also consider taking the SCD greedy strategy one step further: at each iteration, we generate a grid for all directions **individually**, effectively considering 1) what direction would be best and 2) what would the best step along that direction be. In other words, we explore every direction simultaneously and pick the best one. This arguably circumvents the local minima problem, allowing us to more safely be greedy. We expect the success rate results for this method to be the best, at the expense of more iterations and longer times to converge.

3.6 Adversarial training

The last stage to achieve an adversarially robust model consists of adversarial training. We use the best the attack strategy developed in the previous section, which we hypothesise to be the greedy search, generate attacks and inject them in the training set, for adversarial training. As discussed in the literature review in Chapter 2, we expect this method to increase the robustness of the model against adversarial attacks with a potential price to be paid in the predictive power on clean samples.

3.6.1 Evaluation metrics

We review some important concepts regarding the Receiver Operating Characteristic (ROC) curve and detail the reasoning behind our choice for the partial Area Under the ROC Curve (pAUC) to evaluate our

model's performance. The ROC curve is mostly used to display a binary classifier's performance over the possible choices for thresholds. Associated with each threshold there is a True Positive Rate (TPR) and a False Positive Rate (FPR) which the model will incur. Plotting this TPR and FPR for all possible thresholds [40] yields a ROC curve as can be seen in Figure 3.5. In the Figure, we represent an example of a ROC curve in orange as well as what would be the ROC curve of a random classifier in blue. We also shade two areas distinctively: the orange area corresponds to the total area under the curve (Area Under the Curve (AUC)), whereas the hatched area corresponds to the partial area under the curve (pAUC).

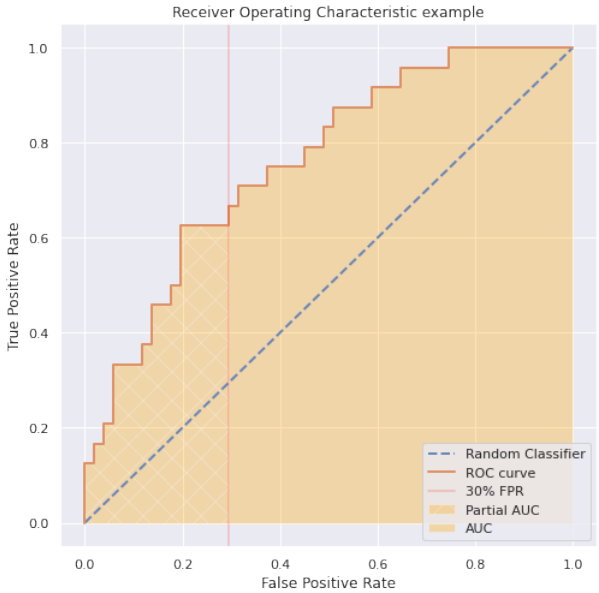


Figure 3.5: ROC curve example

The ROC curve allows the use of a single metric to summarize the performance of a model: the AUC. The idea is that the larger the area, the better TPR-FPR trade-off we can incur in. However, as can be seen in Figure 3.5, all portions of the curve are considered when computing this metric - even overly large FPRs we might not be interested in (in a fraud detection scenario we would typically target a very low FPR). A way to circumvent this issue is to only compute the AUC up to a given maximum FPR. Additionally, we might normalize the pAUC by dividing it by the maximum FPR we are considering (see Equation 3.4). This leaves us with a more digestible metric, conveying the average recall in the region of interest between 0 and our established maximum FPR.

$$pAUC(\mathrm{FPR}_{high}) = \frac{1}{\mathrm{FPR}_{high}} \int_0^{\mathrm{FPR}_{high}} \mathrm{TPR}\; d(\mathrm{FPR}) \tag{3.4}$$

We assess the robustness of a model using two metrics. Firstly, we can evaluate how robust our model is by measuring how strong of an attack the model resists to quantified by the norm constraint. In

other words, we generate attacks against the new, adversarially trained model. Then, we can compare the success rates achieved for each norm constraint when attacking the robust model to the success rates obtained when attacking the baselines. The second metric is the adversarial partial AUC. After replacing all the positives with adversarial attacks, we re-compute the AUC up to a fixed FPR. A robust model should suffer minimal difference between clean and adversarial partial AUCs, whereas we expect to observe a much more significant drop when attacking a non-robust model.

Ultimately, the best way to validate how well protected a model really is would be to deploy both robust and non-robust model in production and monitor their performance over a long enough period of time. As this is a complex and expensive operation, the next best thing is to test it against data where we know there was a significant adversarial adaptation, and compare how each model deals with it. This is also difficult to identify in real production data so we leave it for future work.

3.6.2 Adversarial training algorithm

In this section we provide an overview of our setup to run adversarial training.

Our implementation of adversarial training can be split into three main parts: attacking, validating and training. We start by generating attacks against a baseline model. These attacks take as victims a percentage of all positives in the training set and all positives in validation. The attack strategy to be used can naturally be adapted and has its own set of parameters - the most important being the norm constraint. This parameter establishes the "most expensive" attack the strategy is able to generate, which we expect to play a vital role in determining the attack's success rate.

The next step is to validate the attacks generated as well as the model's performance against them. We compute three metrics to assess the model. All metrics are computed using the validation set. In particular, we compute the **Clean pAUC at a 1% FPR** - the area under the curve, thresholding at some fixed FPR (in our case, 1%), using exclusively non-adversarial data. This lets us quantify the performance of the model against a dataset without any adversarial attacks - as a result, we have an unbiased estimate of how well we expect our model to fare in the clean test set used in the last step. Similarly, we compute the **Adversarial pAUC at a 1% FPR** to measure the performance of our model in an adversarial setting. This metric is computed exactly as the former, except that it is calculated over the validation set that has had its fraud replaced with their adversarial counterparts. We highlight that this score can only be as high as our clean AUC, since the attack strategy will never generate perturbations that are easier to classify by the model. Lastly, we also assess the **Success Rate**, as it is the most intuitive way of quantifying the effectiveness of the attacks generated. Put simply, it expresses how likely it is for a given attack to succeed. To reiterate, we compute it by summing up all the successful attacks divided by the number of attacks generated on true positives. An attack is said to be successful if it causes a formerly correct prediction to flip. Since we are attacking fraudulent examples exclusively, a

set of successful attacks is a set of fraudulent samples that is now classified as legitimate and was previously classified as fraud.

Finally, we include these newly generated samples in the training set. As mentioned, we generate attacks from a fraction of the training set and all the validation set positives - meaning that for the training set a fraction of fraud is replaced with adversarial samples, and for evaluating the adversarial validation metrics, all fraud is replaced with adversarial attacks. Conceptually, we can think of this as a model being attacked every few iterations. While training, the model uses as validation the adversarial validation set generated in the "attacking" step. This process repeats for a number of times, with the model being iteratively attacked. It is possible to stop early, should there be no significant improvement in Adversarial AUC for a given number of iterations in a row. We illustrate the pseudo-code for adversarial training in the algorithm box 2.

Algorithm 2 Adversarial training pseudocode

Input h is a fraud detection model, Δ is the allowed perturbation space
$train\_set,\ validation\_set \leftarrow \mathrm{split}(dataset)$ ▷ Split data into train and validation
$\Omega \leftarrow \mathrm{init\_strategy}(\Delta)$ ▷ Initialize search algorithm across some perturbation space Δ
for $1\ ..\ N$ **do** ▷ Where N is the maximum number of adversarial training rounds
▷ Attacking
$\tilde{x}_{train} \leftarrow \mathrm{generate\_attack}(\Omega, train\_set, h, adv\_fraction)$ ▷ Generate attacks from an $adv\_fraction$ of the $train\_set$ positives against model h using strategy Ω
$\tilde{x}_{val} \leftarrow \mathrm{generate\_attack}(\Omega, validation\_set, h, 1)$
▷ Validating
$\mathrm{AUC}_{adv} \leftarrow \mathrm{compute\_metrics}(\tilde{x}_{val}, h)$
if early stop condition is met **then** ▷ If AUC_{adv} has not improved substantially
break
end if
▷ Training
$h \leftarrow \mathrm{train\_model}(\tilde{\mathrm{x}}_{\mathrm{train}}, \mathrm{h})$
end for

4

Data & Baselines preparation

Contents

In this chapter we provide some insights on the data to be used for this thesis and address how we can establish a relevant baseline to attack and compare our robust models to. We then briefly discuss the results obtained therein. From what was presented in Chapter 3, we will develop adversarially robust models by employing adversarial training: that is, generating and including adversarial attacks during training. We thus split this project in three stages: developing baselines, generating attacks and protecting models by using these attacks in model training.

4.1 Data exploration

We start by referring some of the highlights of our discussions with the risk analysts which led to choice of the dataset being used, as well as some high-level information about it. Then, we discuss some insights gathered when exploring the raw contents of this dataset, as well as its feature-engineered counterparts. In this section, we also iterate over the pipeline followed to go from raw to train-ready data: considerations followed when building a usable sample; time-based splits and how each training, validation and test folds were built.

4.1.1 Dataset overview

From the conducted risk analyst interviews, we opted to use a payment processor dataset. This is mainly due to 1) the fact that fraud tends to change faster within this use case and 2) the intersection with Feedzai's core business. As for the dataset chosen, it contains the historical transactions between a consumer and a merchant in the exchange for goods. These transactions are labelled to be either negative (i.e. legitimate) or positive (i.e. fraudulent). We remark that in fraud detection settings, data is often extremely imbalanced, with fraud rates typically lying below 1%. We summarize some high-level information regarding the dataset in Table 4.1 and then discuss some dataset-specific details.

Use Case	Total Number of Records	Fraud Rate	Duration (weeks)
Payment Processor	$\sim$5B	0.017%	43

Table 4.1: Dataset overview

The dataset offers a wide temporal range spanning almost a year. It also has an acceptable fraud rate, considering the high volume of transactions. A brief exploration of this volume over time is depicted in Figure 4.1. Looking at the figure, we notice a week of missing data around March. There is also some seasonality which appears to repeat itself over weekly periods, which is to be expected.

Regarding features of interest, in the dataset a transaction is flagged to be either Card Present (CP) or Card Not Present (CNP). In this project we focus on CNP transactions since adversarial adaptation should be more of a concern in online transactions. This is important to keep in mind when sampling the

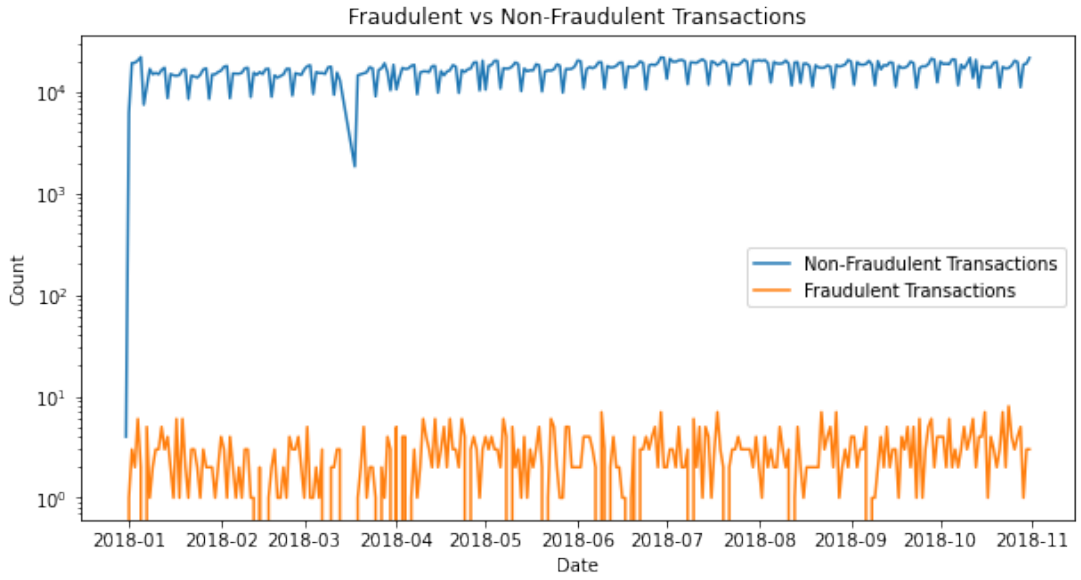


Figure 4.1: Volume of transactions

dataset and developing adversarial samples, as it would be desirable for our baseline to consider CNP samples *exclusively*. The dataset chosen contains a CNP rate of 17.33%.

Lastly, we analyze the feature space of this dataset through the lens of an adversary. Having in mind the previously presented fraudster behavior patterns listed in section 3.3.1, when looking through the raw feature space of this dataset we search for features in one of 4 semantic meanings: IP-related, free-text fields (such as billing and shipping addresses), features related with amounts and card-related features. Furthermore, we consider we can also perturb the timestamp of a transaction since fraudsters can deliberately delay or anticipate when they attempt fraud. In the dataset we found several perturbable features within these semantic meanings. IP-related features were present, as well as amount and card features. On the other hand, we observed no free-text form fields. Moreover, we verify a total of 5 distinct combinations of grouping entities that are used to compute profiles, 4 of which we can perturb. Recall that a profile consists of an aggregation feature computed over a time window that groups over a certain entity (for example, total amount sent by this card in the last hour). Perturbing a grouping entity essentially amounts to resetting the profiles grouped by that entity - this is because if a fraudster changes, for example, to a new card, this new card will have never appeared before in the dataset. For this reason, it is as if every profile grouped by card has just been reset. This assumption extends to profiles grouped by different entities that can be perturbed. Moreover, we assume that profiles that are grouped over multiple entities will be reset if one of them changes. For this reason, a large part (90%) of the profiles can be reset. Lastly, the dataset also derives mappings from perturbable raw features - such as geolocation (which we consider “spoofable”) - as well as amount mappings (such as the logarithm of the amount).

4.1.2 Sampling the data

To build a sample that is going to be feature engineered, it is important to consider that a very frequent grouping entity for building profiles is the card used in a transaction. To reiterate, in fraud detection settings it is common to undersample negative instances and keep all fraud given the highly imbalanced nature of this domain. That being said, if we were to perform random sampling on the dataset, we would inevitably be left with "broken profiles", as the histories we calculated would not match the real card histories. For this reason, a common approach is to instead select a percentage of the cards and fetch their whole histories - that is, sampling by card. Hence, the sampling strategy followed consists of including 100% of cards with at least one fraudulent transaction and 5% of cards with exclusively legitimate transactions. Additionally, we only keep a subset of cards that are meant to be scored in production by the fraud detection system. After employing this sampling strategy, the resulting dataset contained $\sim$180 million transactions, a fraud rate of 0.2873% and a CNP rate of 19.51%.

The derived features were engineered by Feedzai's internal teams for a production system, which ensures we will have a strong baseline model. After computing the engineered features, we proceeded to split the resulting dataset between CP and CNP transactions. Since most cards perform both CP and CNP transactions, there was no way to sample exclusively CNP transactions before computing the engineered features, as we need to consider all transactions when generating profiles. As a result, sampling was done after the features were engineered. After keeping only CNP transactions the sample contained $\sim$34 million transactions with a fraud rate of 1.22%.

4.1.3 Time-based splits

To design the training, validation and test folds we included some padding before the training and after the test splits. Although the dataset is seemingly long, we can observe there is a significant gap in the data around March (see Figure 4.1). The only way to not have broken profiles is to only include data 1 month (the longest window in profiles) after the gap in March. The chosen splits thus comprise of 10 weeks of training, 4 weeks of validation and 6 weeks of test, with a 4-week gap between validation and testing. A visual representation of the dataset and folds can be seen in Figure 4.2. Moreover, although these 6 weeks of test set will be used for metrics to evaluate the model, the models will be scored on the remaining dataset for a better idea on performance over time.

4.2 Baselines

In this section, we detail how our baselines were prepared as well as some of the choices we made along their development given the adversarial context of the project. Our main goal with the baseline models is

Figure 4.2: Time Cross Validation splits

to develop a benchmark of how models typically behave on classifying historical data. As such, we adopt a standard training procedure. We use the best practices to train fraud detection models, as follows :

1. Given the highly imbalanced nature of fraud detection, most of the negative class portion of the data is undersampled and all instances of fraud are kept (discussed in 4.1.2).

2. Since transaction histories are time-dependant and prone to concept drift, time-based splits (where train, validation and test folds are split over time) is common practice.

3. The hyperparameter space of the algorithm used to train the model is explored to tune the model before scoring it against test data.

4.2.1 Hyperparameter tuning

In order to tune the hyperparameters of the LightGBM algorithm used to train the model, we search through a region of a subset of its hyperparameters. This is done through random search; that is, randomly sampling values from the allowed domain for each hyperparameter (see Table 4.2) before each model training trial. In each trial we use the training set and validate it against the validation set using our chosen metric: pAUC at a fixed FPR of 1%. Once we have concluded all the random trials, we select the hyperparameters that yield the highest pAUC up to a fixed FPR of 1% on the validation set.

Hyperparameter	Range	Scale
Number of leaves	[5, 500]	Linear
Minimum data in leaves	[5, 200]	Linear
Learning rate	[1e-3, 1]	Log
Boosting type	[gbdt, goss, dart]	-
Lambda L1	[1e-3, 1]	Log
Lambda L2	[1e-3, 1]	Log
Positive weights	[1e-2, 10]	Log

Table 4.2: Hyperparameter space

We include our exploration of the hyperparameter space along with their verified performances in appendix A.

4.2.2 Results

We now present the baseline performance we obtained. In order to assess the performance of the model we compute two metrics: the normalized partial AUC up to an FPR of 1% and the recall at a fixed FPR of 1%.

In Figure 4.3 we can see the obtained ROC curve, with a dashed red line representing the FPR rate we are using to compute the aforementioned metrics. Additionally, we also show the performance over time in Figure 4.4. As previously stated, we score not only the test set we use to compute and compare metrics between different experiments, but also the periods between validation and test (referred to as "gap" in the figure) and the period after the test. It is interesting to note that this figure shows how certain periods may be harder to classify than others, and the performance may fluctuate over time - so the period chosen to compute metrics in may affect the perceived performance.

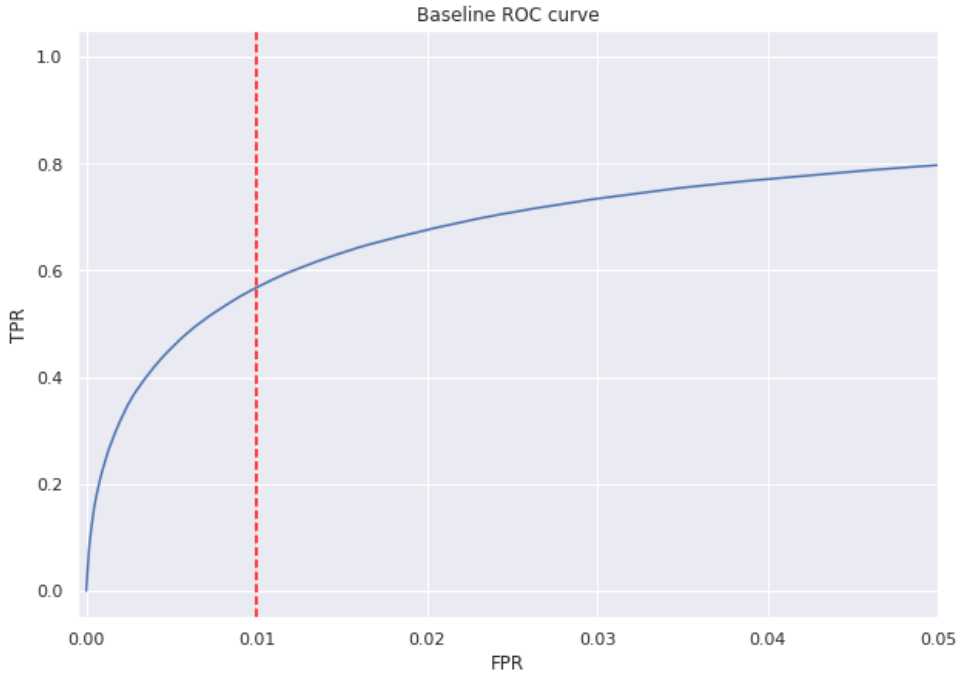


Figure 4.3: Baselines ROC curve

We also compare the scores distribution for each class in Figure 4.5. The histogram represents the frequency of the logit scores for both classes. We obtained a pAUC of 0.3715 and a recall at 1% FPR of 0.5155.

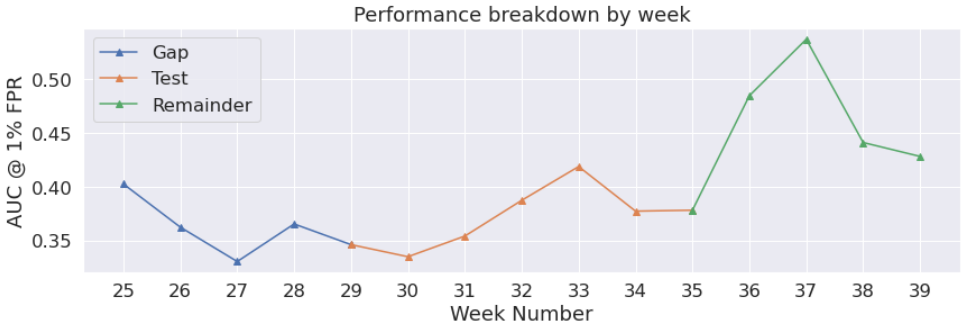


Figure 4.4: Performance broken down by week

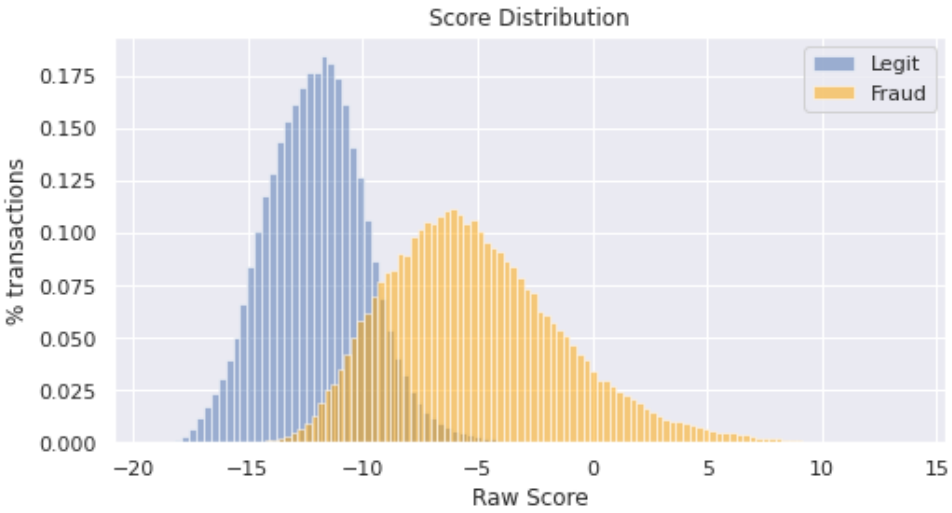


Figure 4.5: Score distribution

4.3 Timestamp perturbation estimators

In this section, we now discuss results for the trained regression models responsible for estimating the profile changes due to timestamp perturbations. This is an important ingredient to be able to conduct the adversarial experiments in the next chapter. After training the regressors following the procedure outlined in Section. 3.3.2.B, we evaluate them using two metrics: the Root Mean Squared Error (RMSE) and the R^2 metric. The RMSE consists of averaging the residuals of the predictions our model is making. R^2 corresponds to the coefficient of determination and it yields a score of 1 if the model outputs perfect predictions and a score of 0 if corresponds to the performance of a model that predicts the mean value. In that sense, if this score is below 0 an estimator that predicts the mean performs better.

4.3.1 Building a dataset

In order to produce a model that can estimate changes successfully, we must produce a dataset that contains multiple perturbed instances corresponding to profile changes over time, spanning multiple perturbation time intervals. We thus start by defining a range and distribution of acceptable delays to inject on the samples that are victims of the adversarial perturbations. To do so, we observe that the windows used to build profiles in the feature plan span from one hour to one month. Thus, we consider that a delay (either positive or negative) should lie between one minute and one week to be consistent with the time sensitivity of the typical timescales of the window used for profiling, which should relate to the fraudster's "willingness" to delay or anticipate their attack. As a side-note, henceforth we define the "delay" as a positive or negative change in the timestamp despite often the word delay being associated with a positive amount of elapsed time. We choose to draw delays from a logarithmic distribution. We want to build a sample with a frequency of input delays that is similar in each order of magnitude - i.e., we want the frequency of 1-minute delays to be comparable with the frequency of 1-week delays. Our main reasoning is that it is desirable for the model to yield comparable performances across different timescales, so uniform sampling would not give enough weight to short magnitude delays, which would likely skew performances as well. Thus, we draw delays according to the following equation:

$$delay = \text{randchoose}(\{-1, 1\}) \cdot e^{u} \,, \tag{4.1}$$

where $\text{randchoose}(\{-1, 1\})$ chooses either sign with equal probability and u is drawn from the uniform distribution $U([\log 1 \text{ minute}, \log 1 \text{ week}])$.

Once we establish a way of drawing delays, we select which samples are to be perturbed. Here it is important to keep in mind the end-goal of this process: our adversarial attacks will target transactions labelled as fraud and will try to mislead the model into producing a legitimate label. Therefore the dataset to be feature engineered will only contain fraudulent cards (that is, cards associated with at least one fraudulent transaction), in which a single fraudulent transaction in the history of these cards has been perturbed. We created copies of each card histories with a unique copy identifier, perturbed one transaction in each card history, merged all copies and computed the derived features. Lastly, it is important to consider whether or not the perturbed timestamp does not change the chronological order of the transactions. In other words if, after perturbing a victim transaction, the previous and following transactions remain as previous and following. Remaining between these transactions could be an invaluable feature for the regression model since, when true, it ensures that profiles only decrease or increase depending on the sign of the delay. We illustrate this point with an example. A count of transactions from a card in the last hour should only increase if we anticipate the timestamp and remain between the aforementioned transactions, since we would possibly be capturing more transactions without ever having the risk

of capturing less. On the other hand, if we delay a transaction but still remain within this window, the number of transactions that are going to be counted can only be smaller. We demonstrate this occurrence in Figure 4.6, in which we select a count profile and filter to include only perturbed samples that still temporally lie between the same previous and next transaction. Observe how in this case negative delays have always led the perturbed profile (y-axis) to be larger, and vice-versa. We thus hypothesize this to be an informative feature for the model. As a result, when we inject some delay, we also compute whether the new perturbed timestamp still lies between the old previous and following transactions. This boolean value is included as an additional feature to aid the regression model estimate the new profile values.

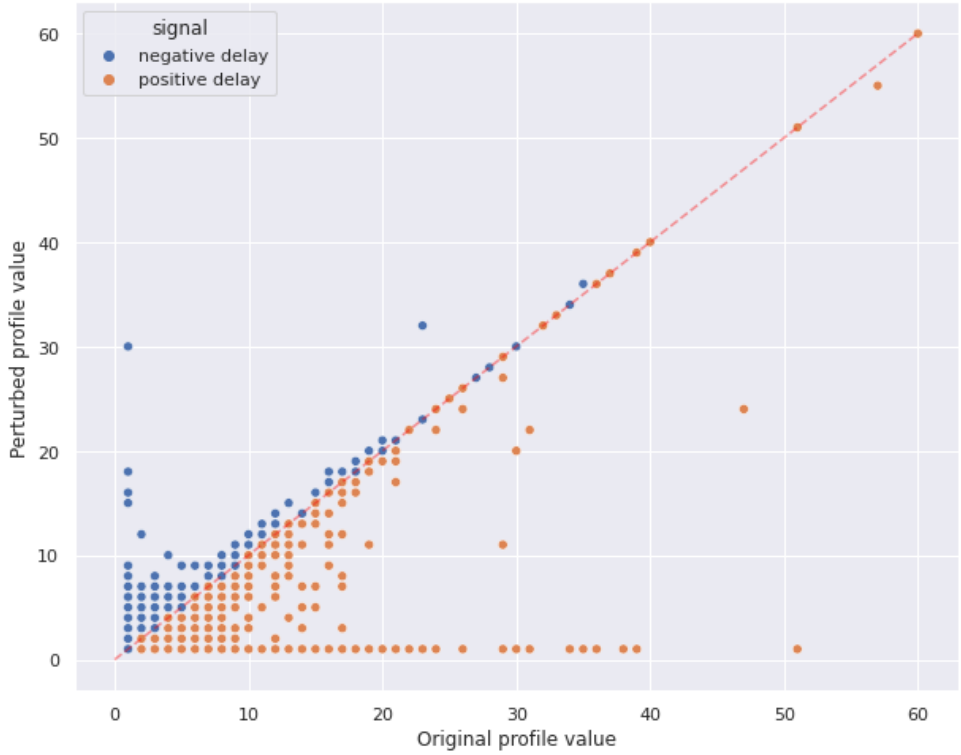


Figure 4.6: Values for an arbitrary profile before and after delay. Dashed red line corresponds to no change. Note that the apparent lack of points is due to many overlapping values.

Finally, we draw some insights on the distinct outcomes that a perturbation in the timestamp may lead to, and how the “delay” feature along with “being between previous and next transaction” might be enough to aid the model in making good predictions. In Figure 4.7 we have plotted the original values for a transaction count profile before injecting an arbitrary delay (x-axis) against the new profile values, this time unfiltered. In this scatter plot, the hue of each point corresponds to the logarithm of the absolute delay injected. Observe how many times the profile remains nearly unchanged in light perturbations, as expected. This hints that there are some cases in which we could use a baseline estimate (i.e. predicting

no change) with correct predictions. However, also note that there are cases in which the profile value drops to one (the possible minimum of a count profile) when our perturbations grows to large values. This suggests that our large perturbations many times end up isolating a transaction in an otherwise empty time window. On the other hand, the opposite also holds for instances that were once isolated in a time window (original profile value of 1) and were "pulled" to higher volume periods. This reassures us that the original profile values, along with delay and its sign will be useful features to predict profiles.

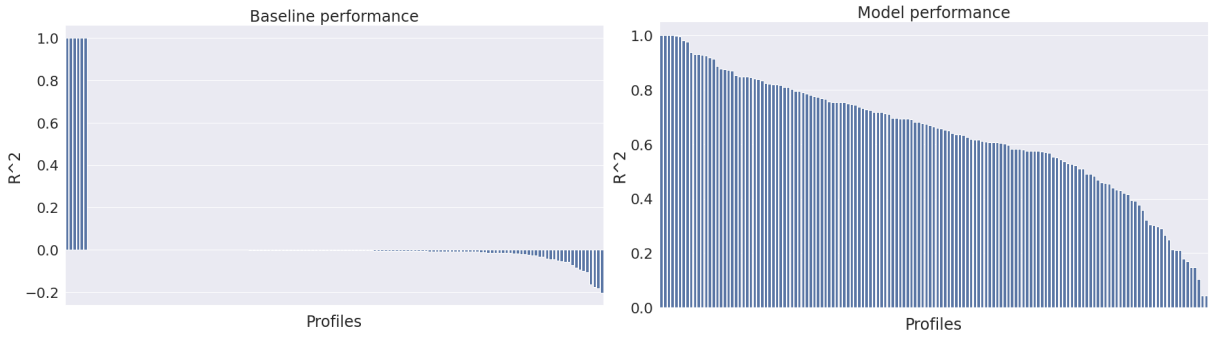
Figure 4.7: Values for an arbitrary profile before and after delay.

4.3.2 Model training approach

Once we have a usable data sample, the regression models to predict the profiles can be trained. We choose to model the change suffered by the profiles, rather than the actual values of the profiles, as we hypothesize this to be an easier objective to model. The prepared dataset containing the perturbation targets was randomly split between train, validation and test sets. We then train LightGBMRegressor models across a hyperparameter space (see Table 4.3), validating the results with the validation set. Then the final model is evaluated using the test set. To model changes, we perform hyperparameter tuning for a single target and then extend the hyperparameters found to a multi-output wrapper. In other words, we performed hyperparameter tuning for a single profile regression model and then used these hyperparameters to train the remaining models (responsible for predicting the rest of the profiles).

Hyperparameter	Range	Scale
Number of leaves	[5, 500]	Linear
Learning rate	[1e-3, 1]	Log
Boosting type	[gbdt, goss]	-
Lambda L1	[1e-3, 1]	Log
Lambda L2	[1e-3, 1]	Log

Table 4.3: Hyperparameter space for regression models



Baseline performance for predicting profiles.

Model performance for predicting profiles.

Figure 4.8: R^2 performance for baselines (left) and our model's (right)

4.3.3 Results

We compare the achieved results with a baseline that always assumes no change in the profiles. In Figure 4.8, each column corresponds to the R^2 score we would get using the baselines (left) and using our multi-output regressor (right). Note that the four occurrences of a perfect R^2 score on the baselines correspond to profiles which happen to be constant in the selected test set. Looking at the two visualizations, we notice there are significant improvements, but still some fairly low scores, which raises concerns. To investigate this further, we conducted an analysis over the residuals produced by the model - here we are interested in observing what are the maximum and minimum deviations we can expect. We found some large residuals that our model would produce when attempting to predict certain profiles - for example, profiles that attempt to measure the time since a previous occurrence. In order to identify under-performers, we normalize residuals by dividing them by their corresponding feature's standard deviation. We then compute the difference of the maximum and the minimum of these normalized residuals since ideally both would be kept low. We plot this difference across a two-dimensional plane along with their respective R^2 performance. We highlight the desirable region of interest in darker red (bottom right quadrant) in Figure 4.9. We experimented removing the features outside this region for the model training altogether and repeated the procedure followed to obtain the baseline models. Since the drop in performance was not statistically significant, we opted to discard these features for the rest of the project. The landscape for the performance on the remaining features can be seen in Figure 4.10.

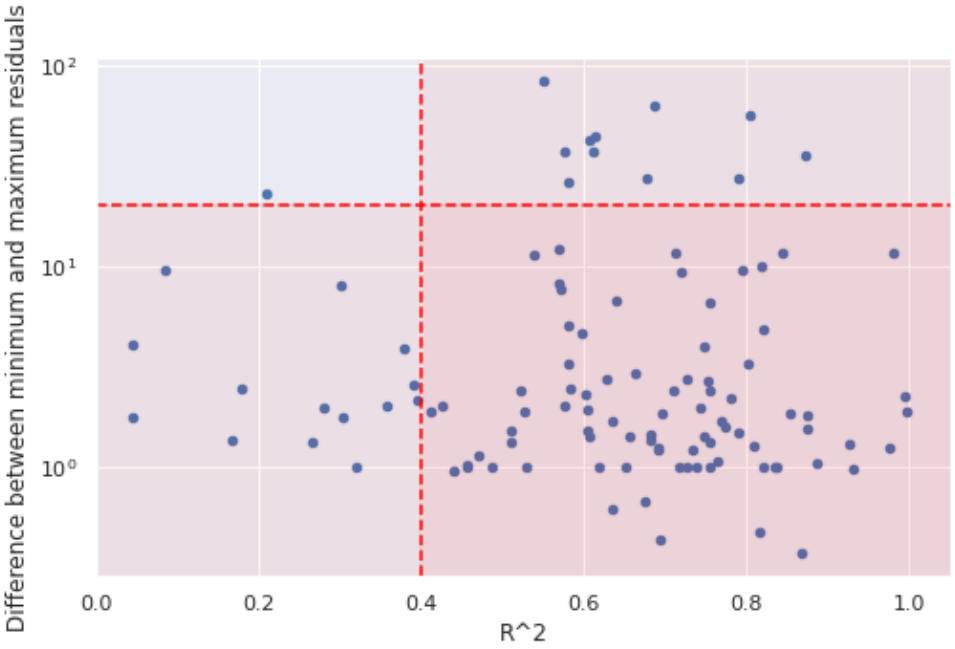


Figure 4.9: Region of interest for profile predictions. We discard large normalized residuals and profiles the regression model struggles to predict.

We consider this approach satisfactory because 1) the reduced features set contains enough signal to obtain a strong baseline model, and ii) the performance of the regressors for the surviving features is high enough to predict many of the profile shifts with good enough accuracy for adversarial training. We highlight that this method just provides an approximation to estimate some of the profiles, which are already a subset of the perturbed features. Since computing profile perturbations is only one component of the perturbations, we consider that allowing for some numerical error is acceptable to be able to run the experiments in a computationally feasible time.

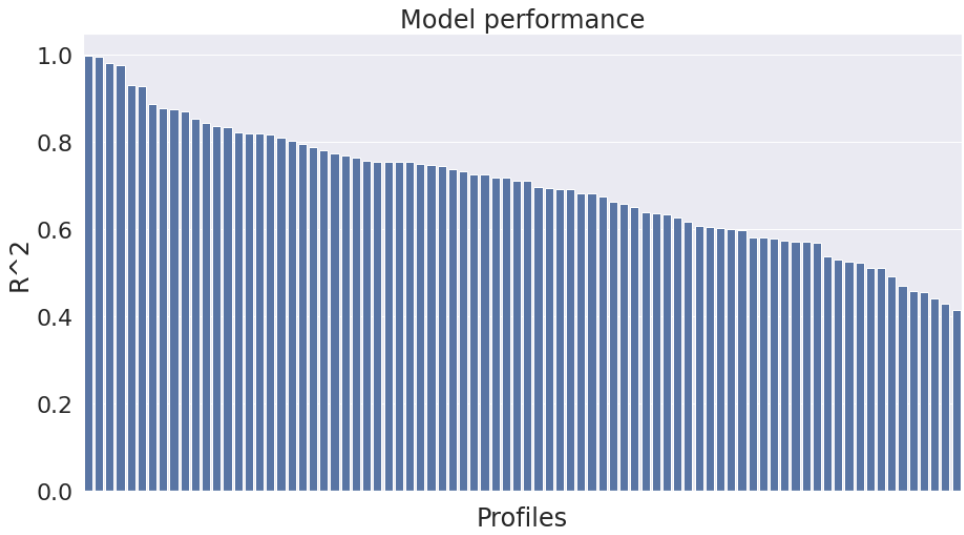


Figure 4.10: Model performance for predicting profiles after discarding underperformers.

5

Experiments

Contents

In this chapter we discuss the conducted experiments to evaluate the strength of the implemented attack strategies as well as the effectiveness of adversarial training to boost the robustness of the models. Therefore, this chapter is structured as follows: we start by benchmarking the attacks against the baselines by assessing their success rates under different norm constraints. In this analysis, we also pay close attention to how each strategy explores the perturbation space. Then, we discuss the conditions in which we run adversarial training. We explore a small space of parameters to pick the best combination of this parameter space. We then fix it while we adversarially train models using different norm constraints. We then compare each of the adversarially trained models by assessing their performance on clean test data and their robustness against attacks across various norms. We conclude this chapter by analyzing the results of some experiments including timestamp perturbations within the considered perturbation space, and assess its impact on training adversarially robust models.

5.1 Adversarial attacks

We compare the four formerly presented attack strategies: random search, the two SCD variants (greedy and cost-efficient) and greedy search. As previously mentioned, we use the random search as our attack baseline. We draw comparisons between strategies by analyzing the different norm distributions (i. e. how well we are exploiting the search space); the different success rate *versus* norm constraint trade-offs; and the score distribution shifts (i. e. how much we can influence the scores produced by the model under attack). Moreover, since these are black-box algorithms, it is also interesting to observe how many queries we need to develop successful adversarial attacks with each strategy.

5.1.1 Random search baseline

In this benchmark, we ran 500 iterations of Random Search over the whole set of victims. In Figure 5.1 we show the success rate versus the norm constraint on the perturbation size when using the **model importance** norm. In the figure we see that, as one would expect, as we perform perturbations that the model is more sensitive to on average (i.e. are more “important” to the model), we are bound to get more successful attacks. However, as previously discussed in 3.4, we consider the values produced by this norm to be hard to interpret and relate with the perturbations generated. On the other hand, in Figure 5.2 we observe the success rate in function of the norm constraint on the perturbation size, this time using the **fraudster cost** norm. We display two curves to show the difference in success rates when including and excluding temporal perturbations. We visualize the success rate as a function of the allowed perturbation magnitude, measured using the fraudster cost. Once again, we conduct our analyses using this norm as we consider it to have a clearer, more intuitive meaning compared to the feature importance norm. As expected, as we loosen the norm constraint we get naturally higher success rates,

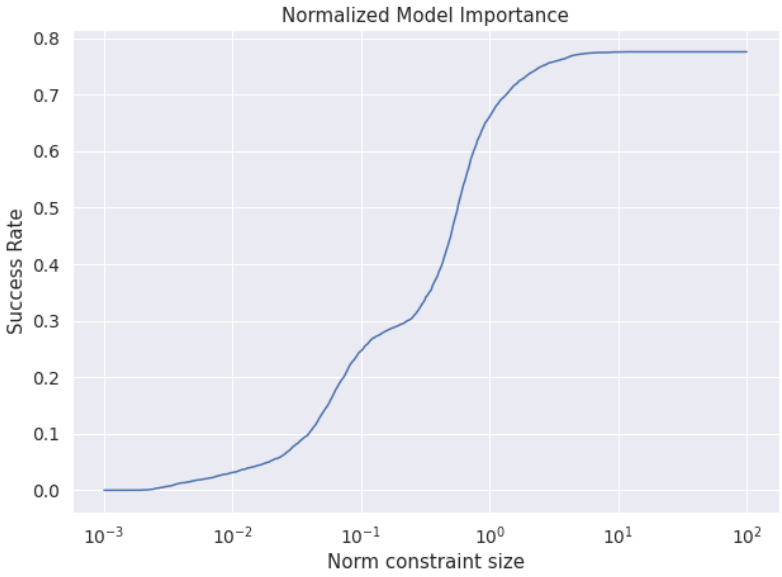


Figure 5.1: Normalized model importance against success rate when performing random search

that is, more successful adversarial attacks. This figure also reveals that there are two very effective discrete perturbations we can do: resetting a card and switching a card (which also demands resetting it). The cost of each of these is represented in the figure by the vertical dashed dotted lines. We point out that, given the attack evaluation procedure, this analysis is unfavourable against this strategy, as higher norm constraints will have a larger number of attacks being considered since in these large spaces it is harder for the random search to saturate the bound. Nonetheless, we observe the designed perturbation space can be reasonably well explored with a simple random search, finding attacks from as little as 20% success rate up to a little over 80%. When looking at the figure we can see the strategy reaches its peak well before it hits the maximum norm constraint (100), suggesting that other attack strategies can maybe find more successful attacks in larger, less constrained domains. We better explore the space when trying out more informed searches in the next section.

Before exploring SCD, we reflect upon the role of temporal perturbations. Measurements of execution times show that injecting temporal perturbations increases execution times fourfold, compared to runs without temporal perturbations. As a result, we opted to ignore them in most of the following experiments - meaning the fraudster cost norm can now assume a maximum value of 82 - though we will still consider them in some experiments (see subsection 5.2.3.

5.1.2 Results

In this section we now display results for both SCD methods and the non-stochastic greedy method, and compare them with the random search baseline. As explained in chapter 3, these three methods explore

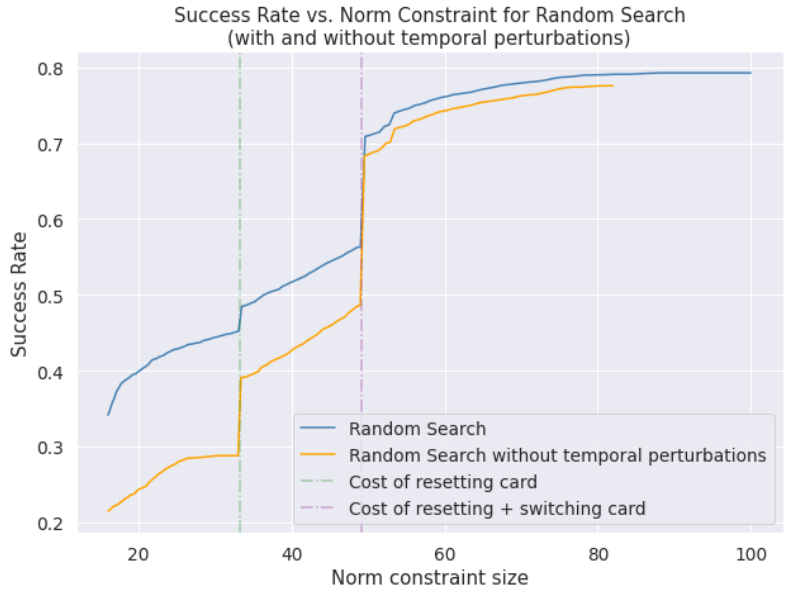


Figure 5.2: Success rate against norm constraint for Random Search with and without temporal perturbations

a set of directions before converging, i.e., being unable to lower the score of a victim sample any further. We thus include visualizations of the convergence of each victim, as well as the evolution of fraudster cost and model score through the strategy's iterations in appendix B.

5.1.2.A Fraudster Cost distributions

In Figure 5.3 we can visualize the fraudster cost distribution observed when attacking the baseline model with a norm constraint of 65 for the various methods. Each norm distribution has a shape consistent

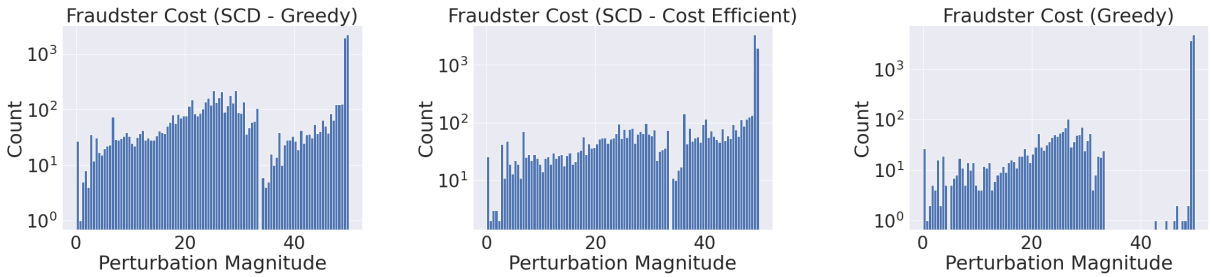


Figure 5.3: Fraudster cost distributions when searching under a norm constraint of 50

with our expected behavior discussed in section 3.5.2. We can see the greedy approach can get stuck in local minima, with some attacks often converging still within lower norms. The cost-efficient approach however, still displays some of these sub-optimal perturbations, but closer to the maximum cost, suggesting it is able to find more effective attacks. Lastly, the fully greedy approach is able to generate almost exclusively expensive attacks, with few attacks in lower norm constraints (note the log vertical

scale in all three histograms). We also note that the non-stochastic greedy strategy generates much fewer attacks between 33 (cost of resetting a card) and 49 (cost of switching a card), suggesting that whenever a card is reset, the strategy realizes it is best to also have it switched.

5.1.2.B Success Rate

Unlike random search, plotting a success rate at norm constraint curve requires running SCD multiple times with different norm constraints. For that reason, the plotted curves for the described algorithms cannot have as much resolution as our attack baseline. Nevertheless, we can still draw valuable comparisons between each strategy.

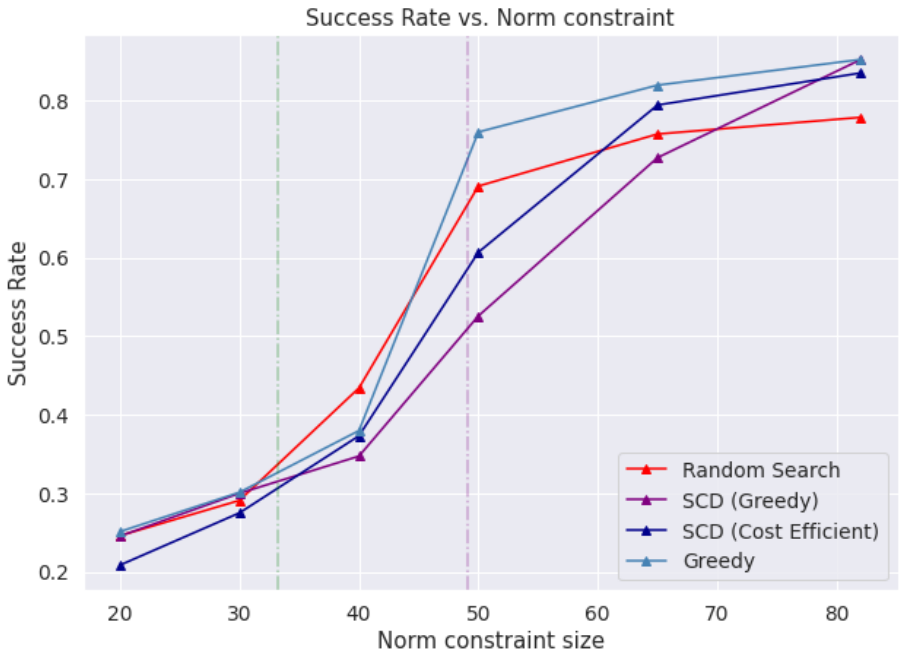


Figure 5.4: Success rate curves for multiple attack strategies against norm constraint

In Figure 5.4 we compare the performance of each strategy. Our expectations suggested in the discussion of the distributions of figure 5.3 are consistent with the results: the cost-efficient approach is able to get more efficient attacks than its greedy counterpart for most of the norm constraints. However, both of these still seem to struggle with beating our baselines, having only both achieved that after the 80 fraudster cost mark. These results are especially surprising considering we are including progressively more random search attacks as we relax the constraint - so lower norms arguably injure random search's performance as previously mentioned. Nonetheless, we can easily see the greedy approach can beat the benchmark for nearly all norm constraints. Moreover, it even seems that if we allowed perturbations to be bigger we could hit higher success rates. We detail the success rates for the various norm constraints across all four strategies in table 5.1.

Attack Strategy	20	30	40	50	65	82
Random Search	0.246	0.291	**0.434**	0.691	0.757	0.778
SCD Greedy	0.245	0.300	0.347	0.526	0.727	0.852
SCD Cost-effic.	0.208	0.275	0.373	0.607	0.794	0.835
Greedy	**0.251**	**0.301**	0.380	**0.760**	**0.819**	**0.852**

Table 5.1: Success Rate at different norm constraints

5.1.2.C Query counts

We assess how many queries we need until convergence for the different norm constraints in each algorithm. For an easier interpretation, we aggregate the total number of queries by averaging the number of queries per victim sample. There are a couple of points worth discussing from the visualization in Fig-

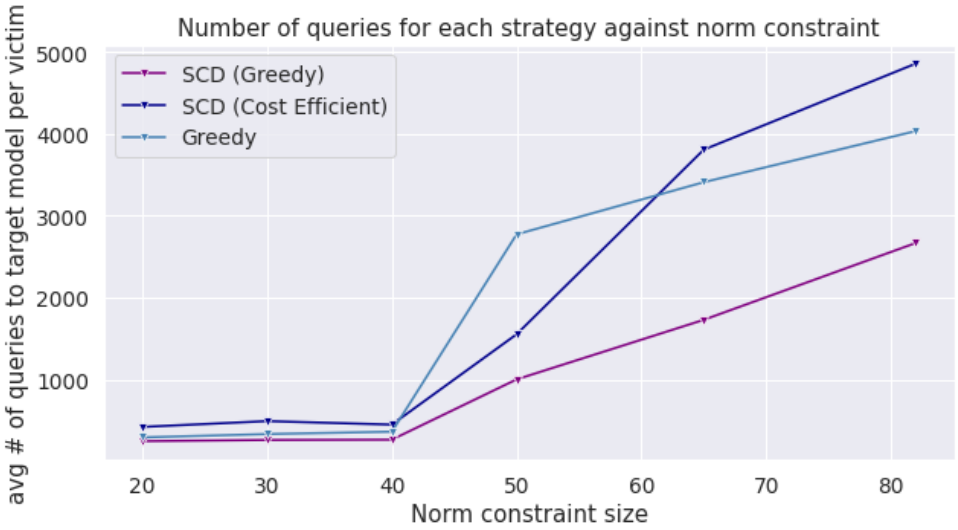


Figure 5.5: Number of queries executed in each strategy for various norm constraints

ure 5.5. Firstly, the rise in the number of queries when we reach a norm constraint of 50 can naturally be explained by the high number of combinations of different card features. The greedy variation explores these possible combinations for every victim, whereas stochastic strategies only occasionally consider that possibility. Moreover, we would expect the number of queries needed to be progressively larger throughout each strategy and norm constraint. This is seemingly what happens, up until we start reaching higher norm constraints. The main hypothesis here is that, as the norm constraint gets more relaxed, we get much more possibilities to reduce costs and to try out different perturbation combinations. This flexibility through the search space is most exploited by the cost-efficient strategy, hence outpacing the number of queries of a greedy strategy on larger spaces.

5.2 Adversarial training

In this section we detail the different combinations of parameters used to run adversarial training and discuss the results obtained.

We choose to fix the same hyperparameters for training the model adversarially to those already found for the baseline without adversarial training. We hypothesize the results should not vary too much if we were to perform hyperparameter tuning every adversarial round, since a large portion of the data is still clean (i.e. the same data used during training in the baselines). Nevertheless, there are certain parameters that we may vary when performing adversarial training. Thus, in this section we briefly explore this parameter space before committing to a specific combination. We vary three parameters: the norm constraint used to generate attacks, the fraction of positives that are adversarial in the training set and the frequency with which attacks are generated during the model training. We start by exploring a few possible combinations of frequency and adversarial fractions for two norm constraints. With this information, we select the best configuration to then sweep over the norm constraint space.

We expect the adversarial fraction to play an important role in balancing the performance in clean and adversarial settings - namely, as we increase the adversarial prevalence in the training dataset, we expect the model to yield a better adversarial performance at the expense of a worse performance on clean data. We consider an adversarial fraction of 0.05 and of 0.25.

Moreover, we also consider two approaches regarding the frequency with which we attack the model. We might want to attack it every few boosting rounds, or we might want to wait until it converges, as measured by adversarial pAUC, before attacking it again. We compare the two approaches. It is worth highlighting that we start this introductory exploration with a norm constraint of 30, meaning that no "strong" perturbations (like card resets) are present, and a norm constraint of 65, where all types of perturbations are already available.

After fixing the frequency and victim rate, we generate adversarial models using four distinct norm constraints. Once we obtain these adversarially robust models, we can assess their performance in the test set. We score them on a clean dataset and measure their performance. Moreover, we also attack each of these models with the same four norm constraints. The key idea here is to understand how models generalize against new, stronger attacks, and whether models trained against strong attacks also perform well against attacks developed in more constrained spaces. We perform this experiment first in a setting where there are no temporal perturbations allowed, and afterwards we perform a few final experiments also including them.

5.2.1 Parameter Tuning

We start by exploring the results obtained in validation in all four settings with a norm constraint of 30 and 65 - namely, the clean pAUC, the adversarial pAUC and the Success Rate through the training epochs. Before discussing the results, we clarify the distinction between frequently employing adversarial rounds and waiting until model convergence before attacking it. In Figure 5.6 we can observe the training loss (adversarial pAUC) when boosting the model in either strategy. In the "blue" strategy, we are

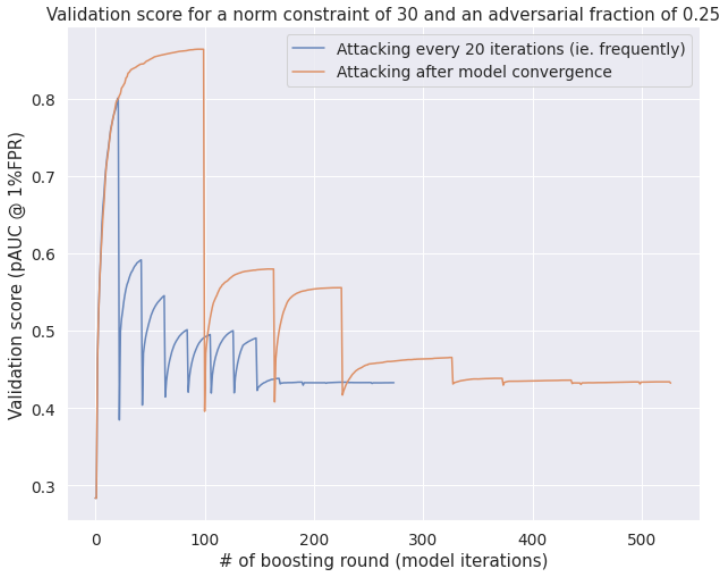


Figure 5.6: Adversarial training - validation score (adversarial pAUC) for a norm constraint of 30 and an adversarial fraction of 0.25

attacking the model every 20 boosting rounds. As a result, what we observe is essentially a rise in adversarial performance, after training once with the adversarially attacked sample, and periodic drops in performance every 20 boosting rounds. Eventually, the performance stabilizes and no longer varies with further attacks. A strategy in which we wait for the model to converge before attacking (orange line) yields a similar behavior - the main differences being the fewer adversarial rounds until the model converges and a much larger number of model boosting rounds. Considering that generating attacks is considerably more expensive than boosting rounds to the model, the latter strategy is faster computationally.

One important aspect to consider is the performance of the new adversarially robust model on clean data. In order to measure this, we show the evolution of clean pAUC through the training epochs in Figure 5.7. We remark how clean performance seems to vary significantly depending on the combination of parameters being explored. Indeed, it is evident that the performance on clean data suffers when including a larger adversarial sample in the training set. This is particularly true when exploring less constrained perturbation spaces, as can be seen from the significant drop in clean pAUC when using an adversarial fraction of 0.25 and a norm constraint of 65. Moreover, we notice very noisy curves in both

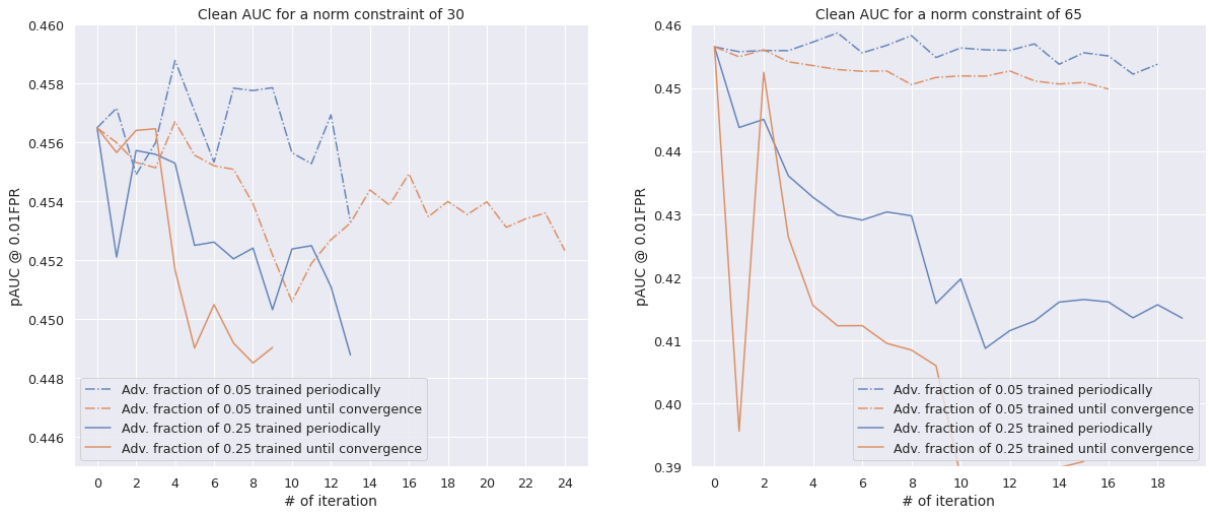


Figure 5.7: Adversarial training - clean pAUC evolution through epochs

figures, though once again, larger adversarial fractions tend to induce sharper variations than smaller ones.

On the other hand, the key metrics to indicate whether our models are getting more robust are adversarial pAUC and success rate. We start by analyzing adversarial pAUC, and illustrate this evolution in 5.8. In the left hand side plot, the results seem to suggest that larger adversarial fractions in the train-

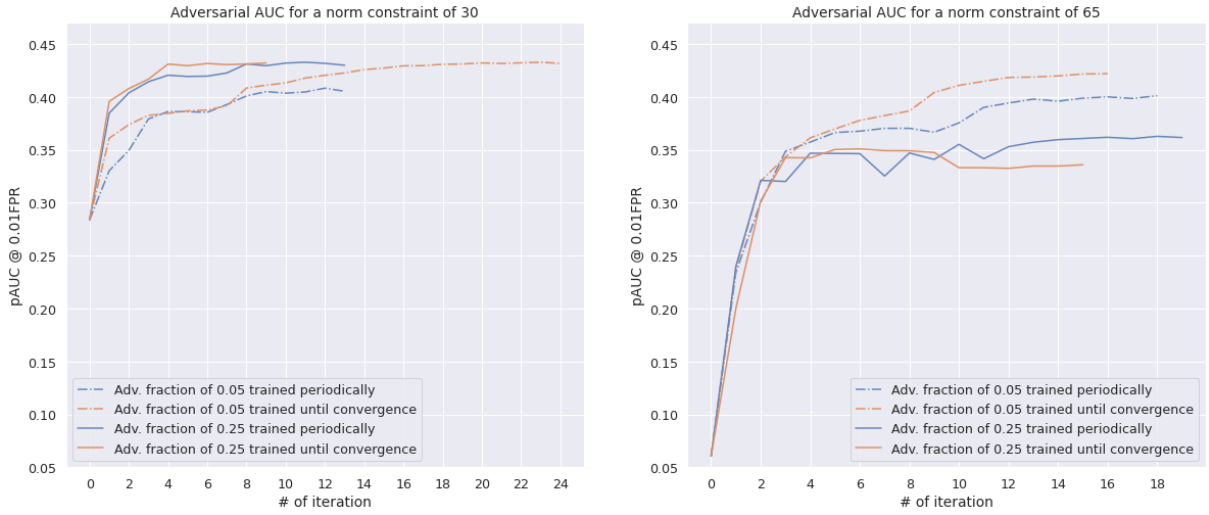


Figure 5.8: Adversarial training - adversarial pAUC evolution through epochs

ing set can yield larger improvements in adversarial performance in early iterations, but that eventually both converge to similar values. In the right-most plot however, this is no longer the case. In either case, with enough iterations, there seems to be no benefit in including a larger adversarial fraction as the training until convergence with the smaller fraction always achieves a result as good or better than

with the larger fraction (both for clean and adversarial pAUC – see also Figure 5.7). Conversely, we observe a similar phenomenon when measuring success rate in Figure 5.9: as we perform more rounds of adversarial training, our strategies find it increasingly harder to generate successful attacks, up until stabilization. Also observe that despite the different norm constraints, each with their own initial success rates, the best models converge toward similar values, showing that the model is able to protect itself against different attacks.

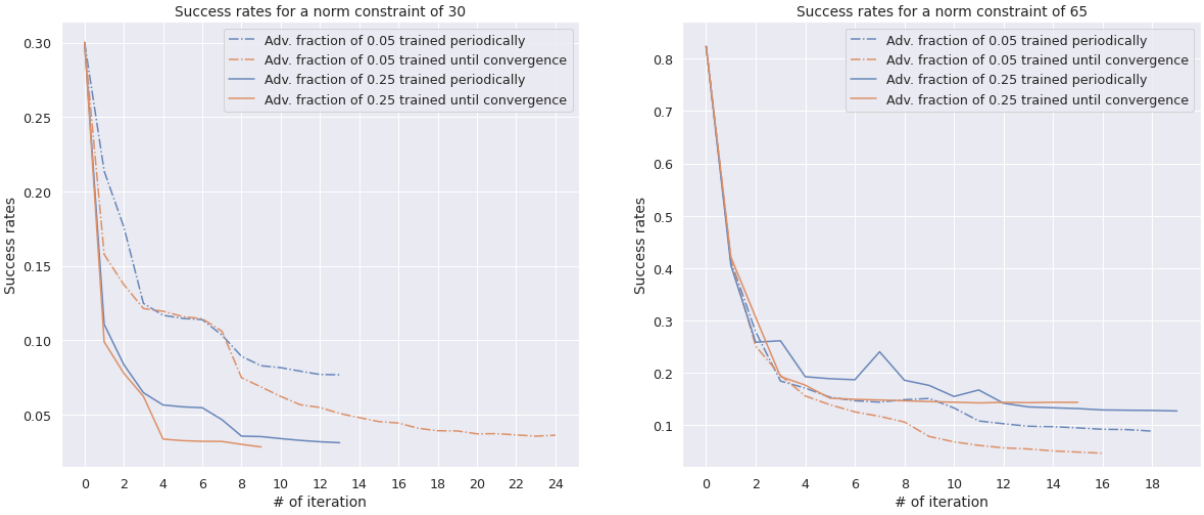


Figure 5.9: Adversarial training - success rate evolution through epochs

Now that we have explored a sample of the parameter space, we are interested in choosing a combination of parameters. In principle, we have a trade-off between obtaining the best adversarial pAUC on validation across different norm constraints or degrading the least the clean pAUC. We consider a simple linear combination of the two given by

$$score = (1 - \alpha) \cdot \text{pAUC}_{Clean} + \alpha \cdot \text{pAUC}_{Adv} \tag{5.1}$$

where α is a parameter in the interval $[0, 1]$ to control this trade-off. We explore the results for different values of α in Table 5.2. We highlight that when $\alpha = 0.5$ our score will correspond to an average between the two performance metrics, and when $\alpha = 1$, we will consider uniquely the adversarial performance metric. We notice that for a norm constraint of 30, the best iteration is the same across both values of α in most experiments. Within this norm constraint, we also notice that clean pAUC consistently assumes values of 0.45, making the experiment that offers the largest adversarial performance a prime candidate: training a model until convergence with an adversarial fraction of 0.05. On the right hand side of the table, α plays a larger role in determining which iteration is best. However, in this case the choice is more obvious, as the "0.05 until convergence" experiment seems to offer a much better adversarial performance with minimal clean pAUC costs for both α values. Given these results, we opted to use an

	Norm constraint	30				65			
	Freq. of adv. rounds	Frequently		Until convergence		Frequently		Until convergence	
	Adv. fraction	0.05	0.25	0.05	0.25	0.05	0.25	0.05	0.25
α = 0.5	Clean pAUC at 1% FPR	**0.4569**	0.4525	0.4536	0.4517	**0.4551**	0.4327	0.4509	0.4264
	Adv. pAUC at 1% FPR	0.4084	**0.4328**	**0.4328**	0.4311	0.4001	0.3468	**0.4217**	0.3426
α = 1	Clean pAUC at 1% FPR	**0.4569**	0.4525	0.4536	0.4490	**0.4538**	0.4156	0.4498	0.4123
	Adv. pAUC at 1% FPR	0.4084	**0.4328**	**0.4328**	0.4322	0.4012	0.3628	**0.4220**	0.3509

Table 5.2: Performance when selecting best iteration according to different criteria

adversarial fraction of 0.05 and to attack the model only after it has converged. As previously described, we now fix these two parameters and vary the norm constraint used to generate the attacks during adversarial training.

As a final note, we notice that, in general, both α values point us towards similar combinations of scores. For this reason, we will select the best iteration based solely on adversarial pAUC, for the sake of simplicity - that is $\alpha = 1$. We can observe the iterations of each experiment in Figure 5.10. We once again highlight that by selecting the iteration with the highest adversarial pAUC we do not incur in a significant loss in clean pAUC, as can be seen from the results in the figure.

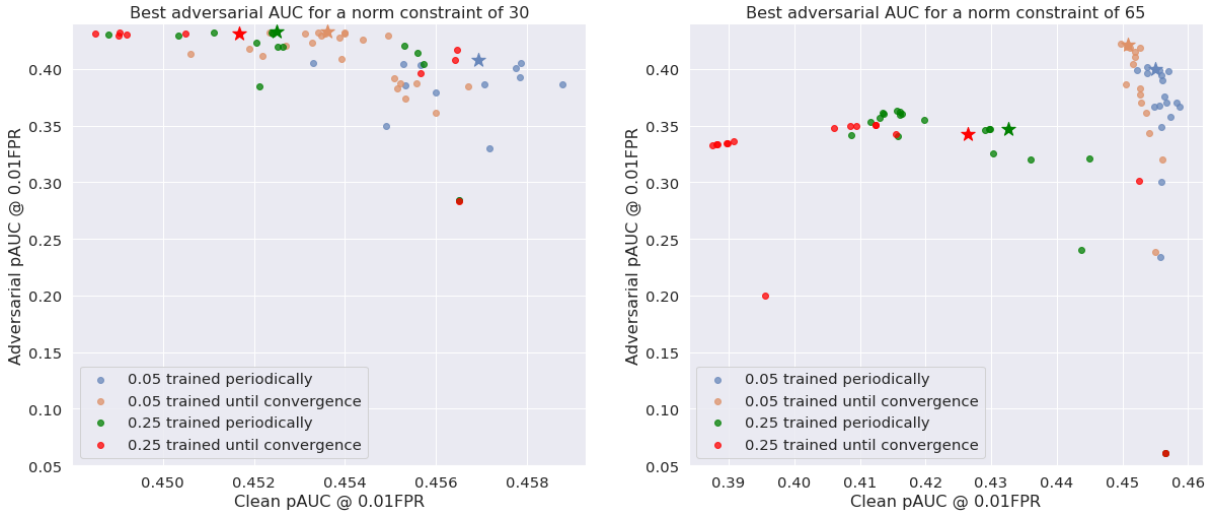


Figure 5.10: Adversarial training - adversarial against clean performance for each iteration in each experiment for a norm constraint of 30 (left) and 65 (right). At each experiment, we star the iteration with the best adversarial pAUC

5.2.2 Test Set Results

With the frequency of adversarial rounds and adversarial fraction fixed, our experiments will now consist of sweeping a range of norm constraints when adversarially training the model. To reiterate, we expect performance on clean datasets to decay substantially as we relax the norm constraint. Conversely, we hypothesize that robustness should increase in a similar proportion - with this increase being reflected both on adversarial pAUC and on success rate when attacking the final robust model with the test set positives as victims. We begin by assessing the change in performance in a completely clean training set for models trained under different norm constraints by showing their respective ROC curves. The

Figure 5.11: Adversarial training - ROC curve per robust model (left) and pAUC against norm constraint used to train the model (right)

results in Figure 5.11 somewhat meet our aforementioned expectations. Recall scores at 1% FPR seem to be ordered exactly like one would expect, with the exception of the model trained under a constraint of 40.

On the right-hand side of Figure 5.11 we have plotted the performance of each model along with their respective standard deviation. The standard deviation of pAUC was computed by randomly building a large number of bootstrap replications of the test set and computing the performance for each one of them. It is worth highlighting that this method only captures the estimation error in the test set, and not the variability from running adversarial training with different seeds. In the figure, note the black line which represents the baseline performance. These results are as expected: we are sacrificing some performance on clean, historical data in exchange for gains in robustness. In other words, when we include adversarial attacks during training, we "pay" a price in clean performance to gain adversarial robustness.

Robustness should improve as we use stronger attacks to train the model, the underlying assumption

being that a model robust against a strong attack will also be robust against a weak one. We assess these hypotheses by evaluating the pAUC after attacking each model and replacing the positives with these newly-generated adversarial attacks. We also analyze the success rate of these new attacks.

From the results presented in Figure 5.12 we can draw some relevant conclusions. First, we see that a larger norm does not necessarily produce a more robust model - observe how a model with norm constraint of 65 achieves comparable robustness compared to one with 82 (that is, the least constrained setting) - provided that both capture the same type of perturbations. We can also observe that a model that has been trained against amount and categorical perturbations (norm constraint of 30) still has a better-than-baseline performance when it comes to defending against larger spaces with card resets - even though it was not exposed to this type of perturbations during training. Our reasoning is that this difference in success rate between the baseline and the model with norm constraint of 30 comes from the fact that our model should now be more robust against amount and some categorical perturbations. All in all, we observe that all the trained models have become significantly more robust to the perturbations used to train them. In other words, we observe that when we cross the norm constraint threshold to train a model with attacks that capture the entirety of the perturbation space (larger than 49), there is a limited gain in including larger attacks during adversarial training.

These results are also observed when evaluating the new adversarial pAUC (see Figure 5.12). When analyzing adversarial pAUC is when we can most evidently see the importance of adversarial robustness, observing a severe drop in performance as we perform attacks with more lax norm constraints.

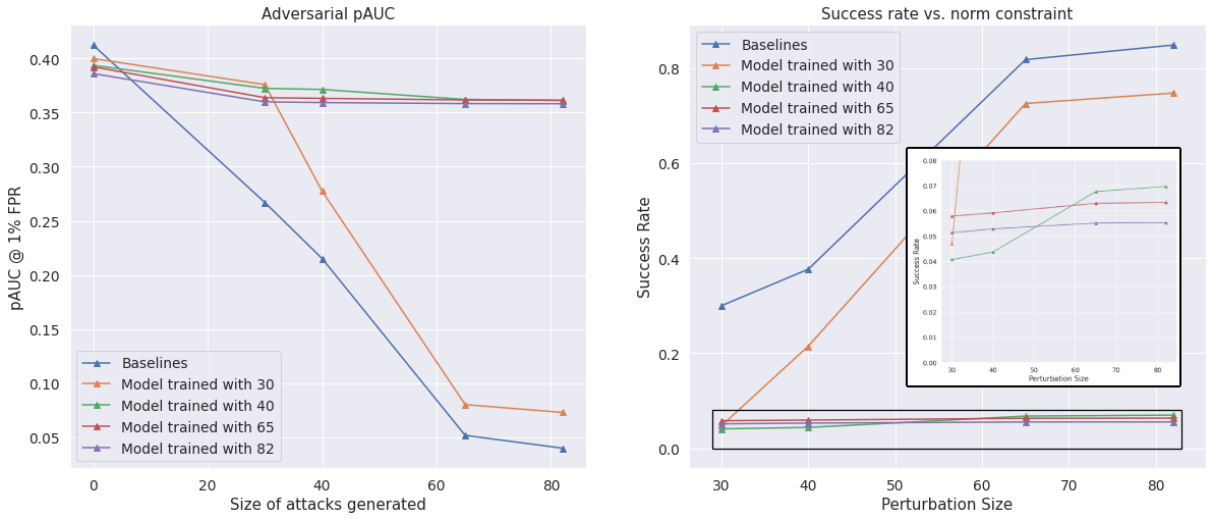


Figure 5.12: Adversarial training - pAUC scores for the norms used to train models (left) and success rates of attacking different models with different norm constraints (right).

5.2.3 Timestamp perturbations

In the final round of experiments, we include timestamp perturbations when performing adversarial training. As previously mentioned, timestamp perturbations are very computationally demanding, so performing adversarial training with this full perturbation space is an expensive task. For this reason, we have opted to keep the parameters previously deemed as best - an adversarial fraction of 0.05 and attacking a model only after it has converged. Moreover, we explore two norm constraints: a norm constraint of 30, effectively only allowing small perturbations (that is, without card resets/switches); and a norm constraint of 100, allowing the full exploration of the developed perturbation space.

Similarly to the previous experiments, we start by presenting the results obtained in validation. As we can see in Figure 5.14, the clean and adversarial pAUC curves in validation seem to evolve like the previously presented results. When subject to larger perturbations, models are forced to pay a larger price in clean pAUC to be able to learn how to classify large adversarial attacks. Moreover, as we perform more adversarial training rounds, performance over clean data tends to decrease slightly, and adversarial pAUC nears the clean pAUC score across both norm constraints.

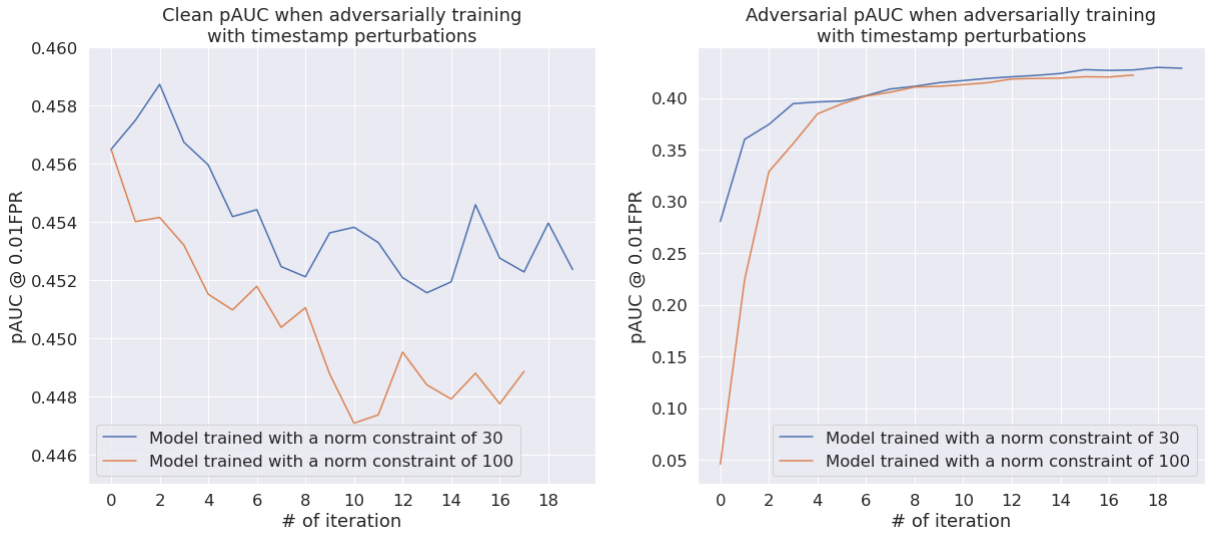


Figure 5.13: Adversarial training - clean pAUC scores evolution through epochs (left) and adversarial pAUC evolution through epochs (right).

We remind that temporal perturbations are extremely expensive to apply, and so this added cost is propagated when generating attacks, which in turn is multiplied when performing adversarial training (generating attacks multiple times). As a result, we ran a reduced number of experiments when including this perturbation. We also highlight that the points that are comparable between the two sets of experiments are at the 82 (without temporal perturbations) and 100 (with temporal perturbations) norm constraint marks, when considering the largest possible attacks.

We now discuss the results obtained in the test set. In general, the findings resemble the ones

already presented: as we include larger perturbations when performing adversarial training, we get a model that is more robust to a broader spectrum of perturbations (see Figure 5.14) at the expense of a larger cost in clean performance. We find that the performance of a model trained with this perturbation space that includes timestamp perturbations with a norm constraint of 30 is similar to the previously seen model trained under the same constrain (Figure 5.15). Interestingly, a model trained with a norm constraint of 100, fully using the space, obtains a slightly superior performance in clean performance compared to the model with a norm constraint of 82 without timestamp perturbations. This suggests that this broader space might result in models that generalize better for future data (reasonable performance in historical data with excellent robustness against future attacks), and reinforces the importance of designing a complete perturbation space.

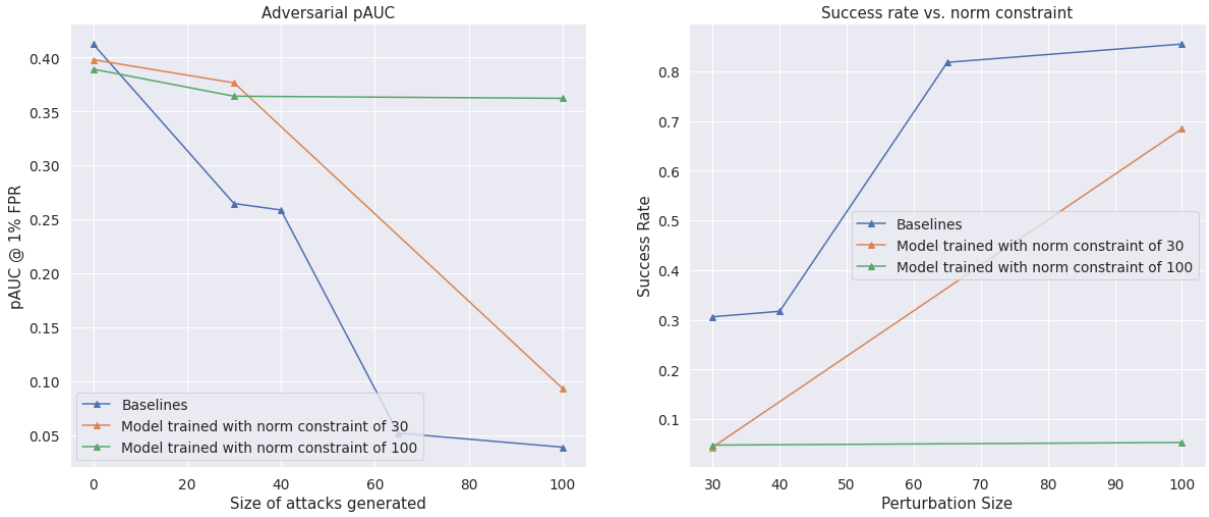


Figure 5.14: Adversarial training with temporal perturbations - pAUC scores for the norms used to train models (left) and success rates of attacking different models with different norm constraints (right).

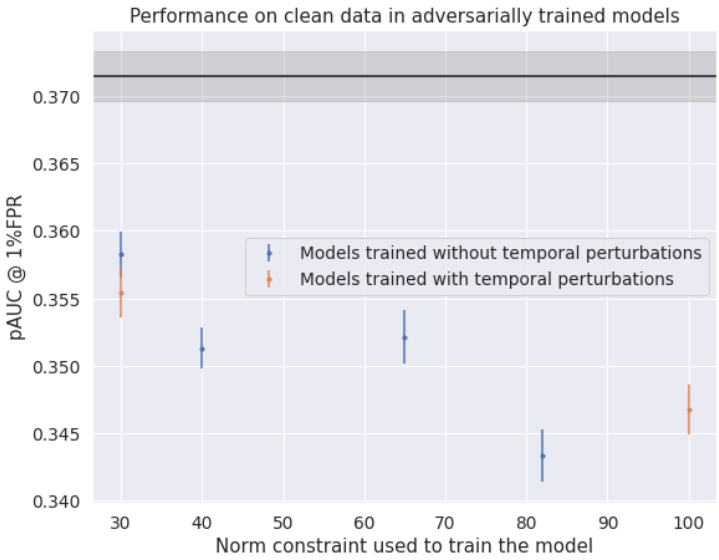


Figure 5.15: Adversarial training with temporal perturbations - pAUC against norm constraint used to train the model.

6

Conclusion

Contents

In this work, we have presented a framework to boost the adversarial robustness of fraud detection models. By defining a perturbation space, exploring it and including the best attacks found therein in the training set we have successfully developed models able to withstand strong attacks. We conclude this dissertation highlighting the most important findings, as well as some remarks on the development of the project. All in all, we consider to have successfully achieved all the goals outlined in Chapter 1, making adversarial robustness a promising direction to explore further within the context of fraud detection.

6.1 Conclusions

Regarding the definition of the perturbation space, we reiterate how important it is to define a complete, meaningful perturbation space. Moreover, we recall that, in this domain, this can only be done with recourse to domain expertise - more so considering that we are handling tabular data. If the perturbation space we generate is disconnected from reality, we might be making models more robust against perturbations that do not occur and still pay a price in historical performance.

We have also developed four strategies to explore the previously defined perturbation space. Namely, a random search that served us as a baseline, which randomly perturbs samples to generate attacks. We have observed that such a strategy is able to produce strong attacks, more so for larger norm constraints. We have also focused on algorithms applicable to discrete search spaces, such as stochastic coordinate descent. Lastly, we have developed a completely greedy approach that always picks the perturbation that lowers the score the most - which yielded the best results of all implemented strategies. We developed these in a black box partial observability setting, and found that the attacks are strong, i.e., have a high success rate on models trained solely on clean data. This may, however, be further improved by exploring other observability settings that will allow us to capitalize on the architecture of the model under attack.

Using the attacks from the strongest strategy, we have implemented and experimented several variations of adversarial training. We have found the parameters that most benefit adversarial robustness to be those that allow a smaller sample of adversarial examples in the training set, and to attack a model only after it has converged. Afterwards, we trained models under the most relevant norm constraints. We have verified the improvements in adversarial robustness for all experiments when attacking a model with the same norm it was trained with. Moreover, we also conclude that there is a limited gain in generating arbitrarily large attacks - both in terms of clean and adversarial performance - as long as the perturbation space is fully captured. Lastly, we have verified the trade-off between clean and adversarial performance, especially as we train attacks with larger perturbation spaces. Particularly, we remark that when we include adversarial attacks during training, the performance on clean, historical data is lower.

6.2 Future Work

We conclude by outlining promising directions for future work. As previously hinted, generating attacks in which the attacker has more access to the model could lead to the generation of more successful adversarial attacks, possibly under the same norm constraints. Establishing this comparison would be interesting on its own. Moreover, a set of more successful attacks could arguably further improve the robustness of the model in adversarial training. These superior attacks could possibly be achieved by conducting further exploration into other optimization algorithms that handle discrete/categorical optimization spaces. The framework presented here should also be applied to a different dataset, which implies the definition of a whole new perturbation space. Currently, this is a laborious task that could greatly benefit from automation - for example, by denoting the semantic meaning of each feature, the perturbation space could be generated automatically. Automating timestamp perturbations requires more effort, but is also possible if distinguish semantically high from low volume profiles. We can then generate the look-up table automatically for high-volume profiles and set-up an automated machine learning (AutoML) pipeline to train a multi-output regression model for the low-volume profiles. Moreover, in order to further validate the findings presented in this dissertation, we also aim to adversarially train a model using a dataset that has surely undergone a change in fraud patterns. As previously argued, this change might have come from a different fraud detection system being put into production.

Bibliography

[1] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, “Robustness may be at odds with accuracy,” 2019.

[2] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, “Distillation as a defense to adversarial perturbations against deep neural networks,” 2016.

[3] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, “Droid-sec: Deep learning in android malware detection,” in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM ’14. New York, NY, USA: Association for Computing Machinery, 2014, p. 371–372. [Online]. Available: https://doi.org/10.1145/2619239.2631434

[4] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, “Large-scale malware classification using random projections and neural networks,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 3422–3426.

[5] E. Knorr, “How paypal beats the bad guys with machine learning,” Apr 2015. [Online]. Available: https://www.infoworld.com/article/2907877/how-paypal-reduces-fraud-with-machine-learning.html

[6] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, “Intriguing properties of neural networks,” 2014.

[7] H. Chen, H. Zhang, D. Boning, and C.-J. Hsieh, “Robust decision trees against adversarial examples,” 2019.

[8] M. Cheng, T. Le, P.-Y. Chen, J. Yi, H. Zhang, and C.-J. Hsieh, “Query-efficient hard-label black-box attack:an optimization-based approach,” 2018.

[9] F. Cartella, O. Anunciacao, Y. Funabiki, D. Yamaguchi, T. Akishita, and O. Elshocht, “Adversarial attacks for tabular data: Application to fraud detection and imbalanced data,” 2021.

[10] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, “Towards deep learning models resistant to adversarial attacks,” *arXiv preprint arXiv:1706.06083*, 2017.

[11] H. Zhang, Y. Yu, J. Jiao, E. P. Xing, L. E. Ghaoui, and M. I. Jordan, “Theoretically principled trade-off between robustness and accuracy,” 2019.

[12] N. Dalvi, P. Domingos, Mausam, S. Sanghai, and D. Verma, “Adversarial classification,” in *IN KDD*. ACM Press, 2004, pp. 99–108.

[13] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, “Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Curran Associates, Inc., 2012. [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[15] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, “Evasion attacks against machine learning at test time,” *Lecture Notes in Computer Science*, p. 387–402, 2013. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40994-3_25

[16] Z. K. Madry and Aleksander. [Online]. Available: https://adversarial-ml-tutorial.org/

[17] V. Vapnik, “Principles of risk minimization for learning theory,” in *Advances in Neural Information Processing Systems*, J. Moody, S. Hanson, and R. P. Lippmann, Eds., vol. 4. Morgan-Kaufmann, 1992. [Online]. Available: https://proceedings.neurips.cc/paper/1991/file/ff4d5fbbafdf976cfdc032e3bde78de5-Paper.pdf

[18] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” 2015.

[19] A. Kurakin, I. Goodfellow, and S. Bengio, “Adversarial machine learning at scale,” 2017.

[20] N. Carlini and D. Wagner, “Towards evaluating the robustness of neural networks,” 2017.

[21] N. Narodytska and S. P. Kasiviswanathan, “Simple black-box adversarial perturbations for deep networks,” 2016.

[22] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, “Deepfool: a simple and accurate method to fool deep neural networks,” 2016.

[23] V. Ballet, X. Renard, J. Aigrain, T. Laugel, P. Frossard, and M. Detyniecki, “Imperceptible adversarial attacks on tabular data,” 2019.

[24] P.-Y. Chen, Y. Sharma, H. Zhang, J. Yi, and C.-J. Hsieh, "Ead: Elastic-net attacks to deep neural networks via adversarial examples," 2018.

[25] Y. Nesterov and V. G. Spokoiny, "Random gradient-free minimization of convex functions," *Foundations of Computational Mathematics*, vol. 17, pp. 527–566, 2017.

[26] S. Ghadimi and G. Lan, "Stochastic first- and zeroth-order methods for nonconvex stochastic programming," 2013.

[27] M. Cheng, S. Singh, P. Chen, P.-Y. Chen, S. Liu, and C.-J. Hsieh, "Sign-opt: A query-efficient hard-label adversarial attack," 2020.

[28] S. Boriah, V. Chandola, and V. Kumar, "Similarity measures for categorical data: A comparative evaluation," in *SDM*, 2008.

[29] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of Documentation*, vol. 28, pp. 11–21, 1972.

[30] D. Golovin, J. Karro, G. Kochanski, C. Lee, X. Song, and Q. Zhang, "Gradientless descent: High-dimensional zeroth-order optimization," 2020.

[31] C. Zhang, H. Zhang, and C.-J. Hsieh, "An efficient adversarial attack for tree ensembles," 2020.

[32] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry, "Adversarial examples are not bugs, they are features," 2019.

[33] S. Calzavara, C. Lucchese, and G. Tolomei, "Adversarial training of gradient-boosted decision trees," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, ser. CIKM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2429–2432. [Online]. Available: https://doi.org/10.1145/3357384.3358149

[34] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on tabular data?" 2022. [Online]. Available: https://arxiv.org/abs/2207.08815

[35] R. Shwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *CoRR*, vol. abs/2106.03253, 2021. [Online]. Available: https://arxiv.org/abs/2106.03253

[36] V. Borisov, T. Leemann, K. Seßler, J. Haug, M. Pawelczyk, and G. Kasneci, "Deep neural networks and tabular data: A survey," 2021. [Online]. Available: https://arxiv.org/abs/2110.01889

[37] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," *CoRR*, vol. abs/1603.02754, 2016. [Online]. Available: http://arxiv.org/abs/1603.02754

[38] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, “Lightgbm: A highly efficient gradient boosting decision tree,” *Advances in neural information processing systems*, vol. 30, pp. 3146–3154, 2017.

[39] Y. Nesterov, “Efficiency of coordinate descent methods on huge-scale optimization problems,” *SIAM Journal on Optimization*, vol. 22, no. 2, pp. 341–362, 2012. [Online]. Available: https://doi.org/10.1137/100802001

[40] C. Marzban, “The roc curve and the area under it as performance measures,” *Weather and Forecasting*, vol. 19, no. 6, pp. 1106 – 1114, 2004. [Online]. Available: https://journals.ametsoc.org/view/journals/wefo/19/6/825_1.xml

A

Hyperparameter space

The hyperparameter space to explore when tuning a model plays a vital role in determining its final performance. We include the results observed when performing this exploration using random search. We show the various results for the boosting algorithm used, the L1 and L2 regularization parameters, the learning rate and the minimum data per leaf. Though there were more parameters being tuned, these correspond to the most relevant.

A.1 Boosting Algorithm

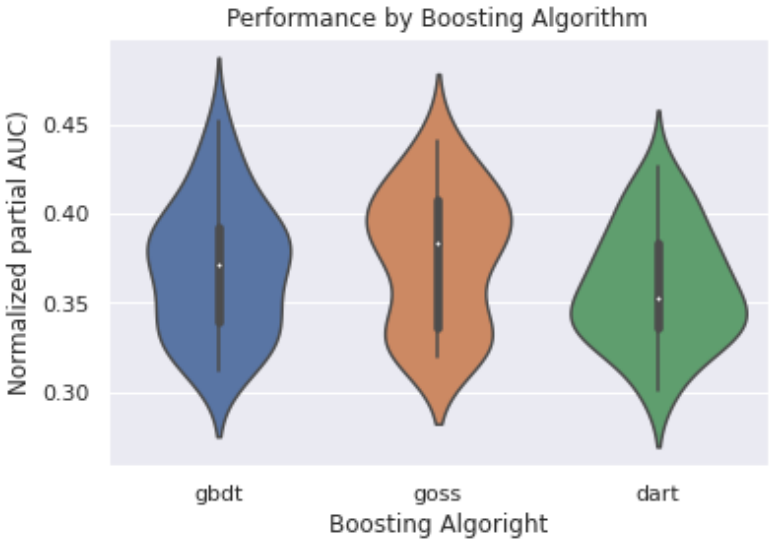


Figure A.1: Performance by boosting algorithm

A.2 Regularization parameter

Figure A.2: Performance by L1 and L2 regularization

A.3 Learning Rate

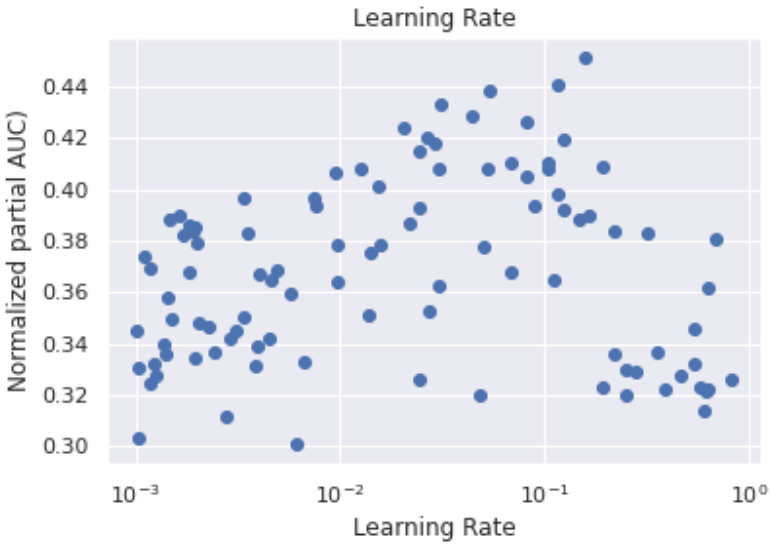


Figure A.3: Performance by learning rate across both folds

A.4 Minimum Data per Leaf

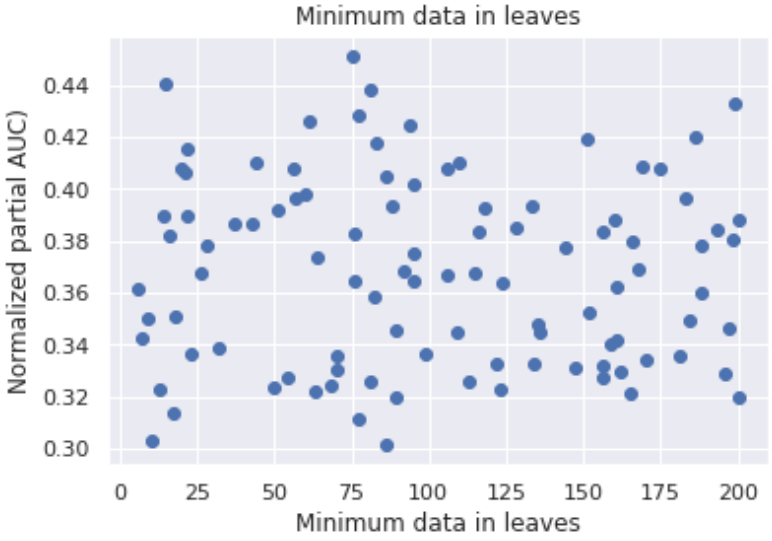


Figure A.4: Performance by minimum data per leaf parameter

B

Attack Convergence

We display the convergence of the victims. This section is split into 3 sub-sections: visualizing the evolution of the fraudster costs selected from the grid (for 10 randomly selected attacks), visualizing the evolution of the scores selected from the grid (likewise), and a summarized version of both. The order presented is the same in each subsection: greedy, cost-efficient and omnidirectional. These results were obtained for a norm constrain of 50.

The summarized plots show clearly how long each approach takes to converge. We can observe that the greedy implementation, on average, converges after exploring less than 10 directions. The cost-efficient approach can explore up to 100 directions before converging. And lastly, observe that the x-axis on the omni approach corresponds to the number of sweeps performed, hence the much less granular look. A sweep explores 5 directions, so we can expect the average victim to explore about $15 * 5 = 75$ directions before converging in the last implementation.

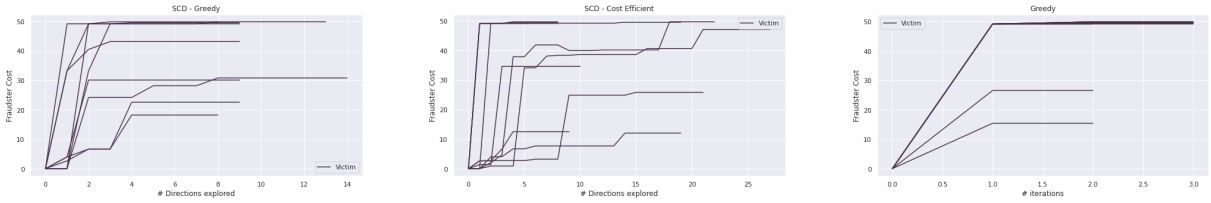


Figure B.1: Cost convergences distributions

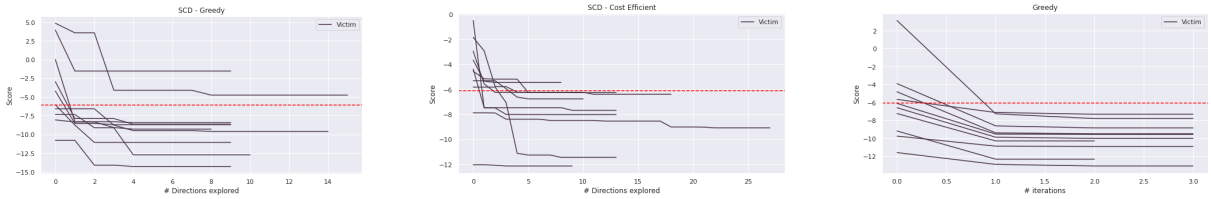


Figure B.2: Score convergences distributions

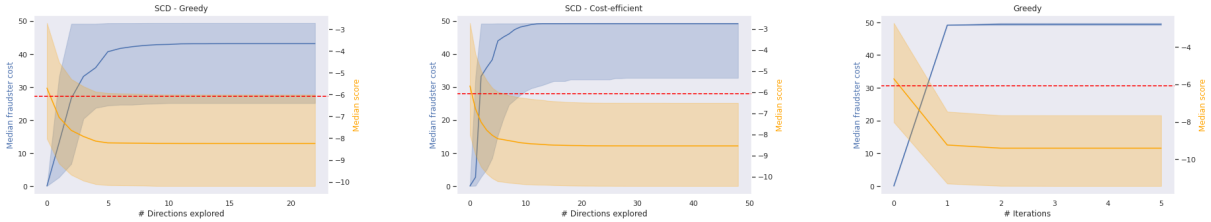


Figure B.3: Convergence summaries