PhD-FSTM-2020-61
The Faculty of Science, Technology and Medicine

DISSERTATION

Presented on 08/10/2020 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Ramiro Daniel CAMINO
Born on 4 July 1986 in Buenos Aires (Argentina)

MACHINE LEARNING TECHNIQUES FOR SUSPICIOUS TRANSACTION DETECTION AND ANALYSIS

Dissertation defence committee

To my mother, my sister and my wife.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 50 figures.

Ramiro Daniel Camino
October 2020

Acknowledgements

First and foremost, I would like to thank my supervisor A-Prof. Dr. Radu State, for offering me the opportunity to come to Luxembourg and give me the chance to pursue a Ph.D. I am also grateful to A-Prof. Dr. Djamila Aouada and Prof. Dr. Diego Fernandez Slezakfor joining my thesis committee for their advice and their questions. I want to thank also Prof. Dr. Petko Valtchev for his guidance during my first years of Ph.D. studies, and A-Prof. Dr. Raphaël Frank for accepting the role of chairman in my thesis defense. Furthermore, I am very grateful for all the time and effort Dr. Christian Hammerschmidt dedicated to me.

I am grateful to the Luxembourg National Research Fund, the Interdisciplinary Centre for Security, Reliability and Trust, the University of Luxembourg and Logos IT Services for funding my research, and also to the HPC facilities of the University of Luxembourg for allowing me to use their platform and resources to carry out my experiments.

I would also thank all my colleagues from the SEDAN team for interesting discussions and exchanges and pleasant conversations during coffee time. Additionally, I am very grateful to Valérie Gregoire for all her assistance and the patience she had with me regarding all the administrative tasks. Finally, I would like to thank my family and my friends for all their support, which gave me the strength to face the challenges I encountered during these past four years. I wouldn't get this far if it weren't for all of you.

Abstract

Financial services must monitor their transactions to prevent being used for money laundering and combat the financing of terrorism. Initially, organizations in charge of fraud regulation were only concerned about financial institutions such as banks. However, nowadays, the Fintech industry, online businesses, or platforms involving virtual assets can also be affected by similar criminal schemes. Regardless of the differences between the entities mentioned above, malicious activities affecting them share many common patterns. This dissertation's first goal is to compile and compare existing studies involving machine learning to detect and analyze suspicious transactions. The second goal is to synthesize methodologies from the last goal for tackling different use cases in an organized manner. Finally, the third goal is to assess the applicability of deep generative models for enhancing existing solutions.

In the first part of the thesis, we propose an unsupervised methodology for detecting suspicious transactions applied to two case studies. One is related to transactions from a money remittance network, and the other is related to a novel payment network based on distributed ledger technologies. Anomaly detection algorithms are applied to rank user accounts based on recency, frequency, and monetary features. The results are manually validated by domain experts, confirming known scenarios and finding unexpected new cases.

In the second part, we carry out an analogous analysis employing supervised methods, along with a case study where we classify Ethereum smart contracts into honeypots and non-honeypots. We take features from the source code, the transaction data, and the funds' flow characterization. The proposed classification models proved to generalize well to unseen honeypot instances and techniques and allowed us to characterize unknown techniques.

In the third part, we analyze the challenges that tabular data brings into the domain of deep generative models, a particular type of data used to represent financial transactions in the previous two parts. We propose a new model architecture by adapting state-of-the-art methods to output multiple variables from mixed types distributions. Additionally, we extend the evaluation metrics used in the literature to the multi-output setting, and we show empirically that our approach outperforms the existing methods.

Finally, in the last part, we extend the work from the third part by applying the presented models to enhance classification tasks from the second part, commonly containing a severe

class imbalance. We introduce the multi-input architecture to expand models alongside our previously proposed multi-output architecture. We compare three techniques to sample from deep generative models defining a transparent and fair large-scale experimental protocol and interesting visual analysis tools.

We showed that general machine learning detection and visualization techniques could help address the fraud detection domain's many challenges. In particular, deep generative models can add value to the classification task given the imbalanced nature of the fraudulent class, in exchange for implementation and time complexity. Future and promising applications for deep generative models include missing data imputation and sharing synthetic data or data generators preserving privacy constraints.

Table of contents

List of figures

List of tables

List of Algorithms

Chapter 1

Introduction

Imagine that a criminal obtained some profit by participating in some illegal activity, and that criminal wants to deposit his money into a bank for future use. However, if the sum is too large, for example, ten thousand or more, the transaction will raise an alarm that will lead to an investigation. Of course, the criminal wants to avoid any questions, and instead of doing that, he splits the amount into several smaller deposits. This technique is known as structuring. Banks and other financial institutions usually employ software solutions based on rules to detect known criminal techniques like this one. The problem is that the set detection of rules needs to be updated often to catch up with the evolving criminal tactics.

This dissertation's primary goal is to present tools and general methodologies to analyze and detect frauds, scams, suspicious activities, or any anomalous behavior related to users, accounts, or transactions between them. Nevertheless, the idea is to achieve this goal even in the presence of novel criminal tactics. The proposed solutions will be presented alongside practical use cases related to traditional banking and modern financial technologies, distinguishing scenarios where we can apply supervised or unsupervised methods. Additionally, we will dive into the topic of deep generative models to evaluate this technology's possible applications in the fraud detection domain.

1.1 Know Your Customer

The term *Know Your Customer (KYC)* denotes the process that banks and several associated businesses must follow to identify their clients and the origin of funds correctly. It is closely related to the practices implemented for *Combating the Financing of Terrorism (CFT)* and *Anti-Money Laundering (AML)*. The G7 Summit of 1989 established the *Financial Action Task Force on Money Laundering (FATF)* for these purposes. Another organization from the United States is the *Financial Crimes Enforcement Network (FinCEN)*, and related laws

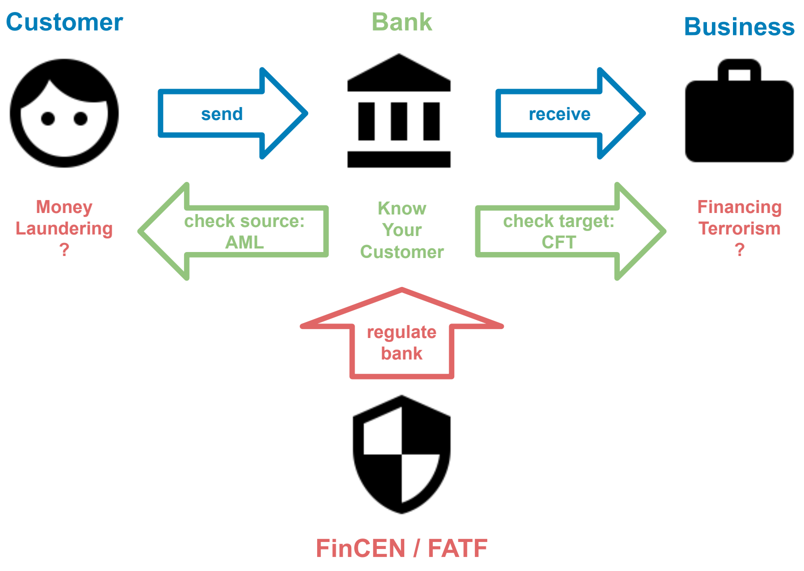


Fig. 1.1 Relationship between KYC, AML, and CFT.

include the *Bank Secrecy Act* and the *Patriot Act*. According to the introduced policies, bank employees must fill a *Currency Transaction Report* whenever a movement of funds involves more than u$d 10,000. Furthermore, employees in financial institutions must submit a *Suspicious Transaction Report (STR)* if they detect an odd transaction (but are forbidden from informing the client about it). A *Suspicious Activity Report (SAR)* is an extended version that can group multiple STRs along with a detailed narrative. The controls associated with those regulations usually include various types of identity checks and documentation requests. Banking systems must validate suspicious clients' names against known criminal lists and *Politically Exposed Persons*. The systems also calculate risk metrics for accounts based on questionable occupations, blacklisted nationalities and destination countries, the amount and currency of fund movements, and their frequencies in different time slots. More on the subject can be found in [146, 128].

The literature covered to a fair extent the topics of money laundering and terrorism financing. In [146], the authors discuss the variety of strategies and approaches used in the related illicit activities. To start with, they coin the term *structuring* (elsewhere called *smurfing*) as the stage of dividing a transaction whose volume exceeds regulated limits into a set of smaller ones. In [128], the authors define *placement* as the stage in which criminals introduce the money into the financial system, and define *layering* as the subsequent movements of funds to different accounts under potentially diverging jurisdictions. During the final stage, known as *integration*, the previously legitimated funds are used in standard purchasing transactions, for instance, to acquire assets or luxury goods. These techniques'

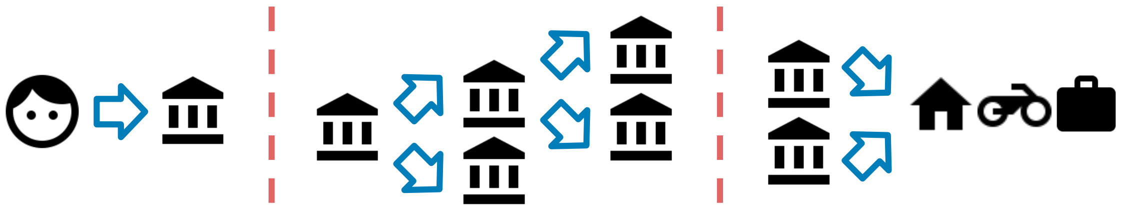


Fig. 1.2 Placement, layering, and integration.

goal is to avoid being reported due to regulatory breaches and thus becoming subjected to further investigations. In the case of laundering, the immediately sought effect is the obscuring of the money provenance, and in the case of terrorism financing, its ultimate beneficiary. We depict how all these concepts relate in Figure 1.1.

In this field, domain experts carefully craft a set of rules used in practical software tools. These rules check individual attribute values or a combination of them to see if a transaction posses suspicious characteristics. Rule-based systems usually apply one rule after another, and if none of them raise the alarm, the transaction is considered legitimate. The problem with such tools is that they are bound only to detect known suspicious schemes. To capture new cases, experts must first identify and define the underlying tactics and then add the new detection rules to the tool rule base. The set of rules tends to grow in size and complexity very fast, which complicates the system maintenance and is very prone to human errors [9]. Machine Learning and Data Mining algorithms offer the possibility of detecting suspicious movements in financial transactions, and also the chance of discovering new behavior patterns. The challenge in this field is the marked imbalance of classes: the number of irregularities tends to be very small compared to legitimate transactions. An even more troublesome issue is the lack of knowledge about anomalous cases: labeled datasets are hard to obtain or do not exist at all.

We face rapid growth in alternative payment methods, social trading platforms, and digital currencies. Consequently, authorities must constantly adapt regulations and policies to new criminal schemes exploiting these technologies' more profitable opportunities. However, while fraud detection methods need to be updated accordingly, the research over traditional and modern financial domains is still in its early stages.

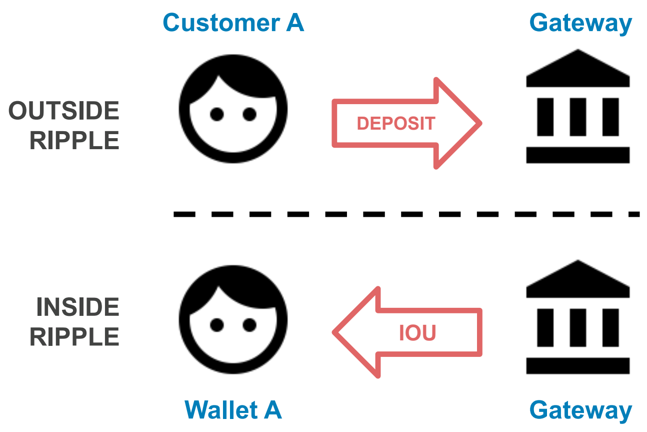


Fig. 1.3 Ripple gateway mechanism.

1.2 Ripple

Ripple[1] is a novel service that offers a remittance network, a currency exchange market, and a real-time settlement system. It was released for the first time in 2012 by the homonym company. Their protocol allows banks to send international payments across multiple networks faster than with traditional methods. Transactions are cryptographically signed, and they can move an internal currency denominated XRP. The distributed ledger records the balance of that currency for every account along with the transactions. For settlements based on other assets, Ripple registers debt obligations among trust lines previously defined between parties. This mechanism is also known as *I Owe You (IOU)* (see Figure 1.4). Some businesses and individuals defined as gateways [2] provide the means for moving money or other valuables in or out of the network (see Figure 1.3). They are responsible for complying with local regulations and notifying the appropriate authorities about irregularities.

The Ripple network represents information in a distributed and shared ledger [3]. The ledger is a collection of transactions, account information, a date, and a sequence number. The network stores a sequence of ledgers. Network nodes have to achieve a consensus to decide which transactions modify the last ledger to obtain a new version. A tree structure stores the data, with node types related to currency exchange offers, trust lines, and balances.

Among other information, the official Ripple Data API[4] provides the full transaction history. Every request can indicate several filters like the type of transaction, date ranges,

[1]https://ripple.com/
[2]https://ripple.com/build/gateway-guide/
[3]https://ripple.com/build/ledger-format/
[4]https://ripple.com/build/data-api-v2/

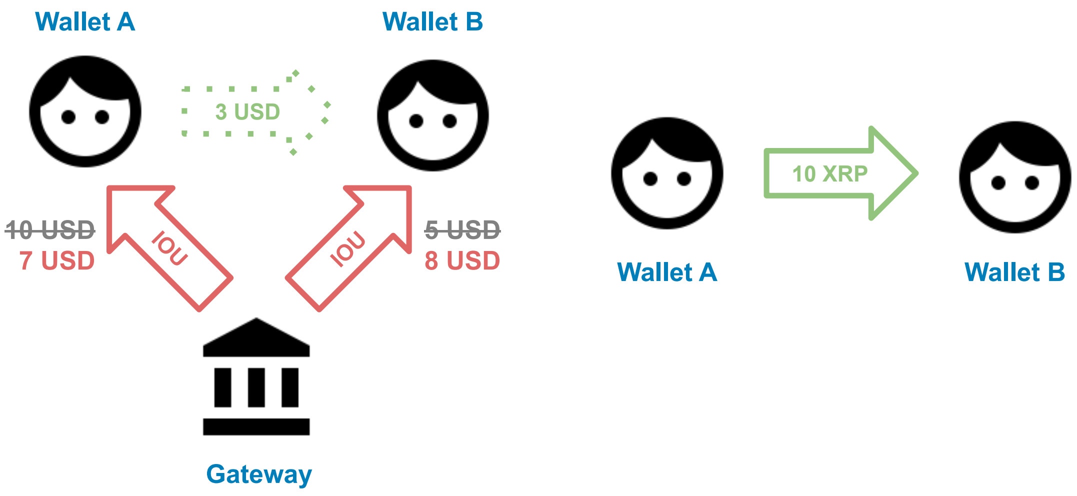


Fig. 1.4 Ripple fund movements: IOU transfer (left) and XRP transfer (right).

pagination, and order. The response contains the list of desired objects. Each of them describes the transaction with an identifying hash, the date of creation, the sequence number of the ledger that first included the operation, the transaction arguments, and metadata indicating which nodes the transaction changed in the ledger tree [5].

Every transaction contains a considerable amount of fields, but the most important one is the type of operation. It describes the intention of the transaction and also defines the existence of additional fields. We focus on three of these cases: payments, offer creations and offer cancellations. This dissertation does not cover the rest of the alternatives related to administrative tasks (e.g., creating trust lines).

An offer creation is equivalent to an order submission in a traditional foreign exchange market. The user indicates some currency he is disposed to deliver in exchange for another amount expressed in a different currency. The offer creator can also indicate a wide variety of options and flags to define additional logic for the order, but we center the study on the amounts and currencies. An offer is only *executed* when another one that matches its requisites is submitted. The user has no control over the order matching. The priority is given first to the orders with the best exchange rate, in case of a tie, to the ledger's older alternative. An order can also be *partially executed* when it crosses others, but some of the offered value remains. The ledger saves the remaining amount in a smaller offer with the initial exchange rate. Users can define flags in their offers to avoid this behavior. A user request can cancel an offer before it gets executed. However, other mechanisms can also remove an offer from the book, like lack of funds or expiration (see Figure 1.5).

[5] https://ripple.com/build/transactions/

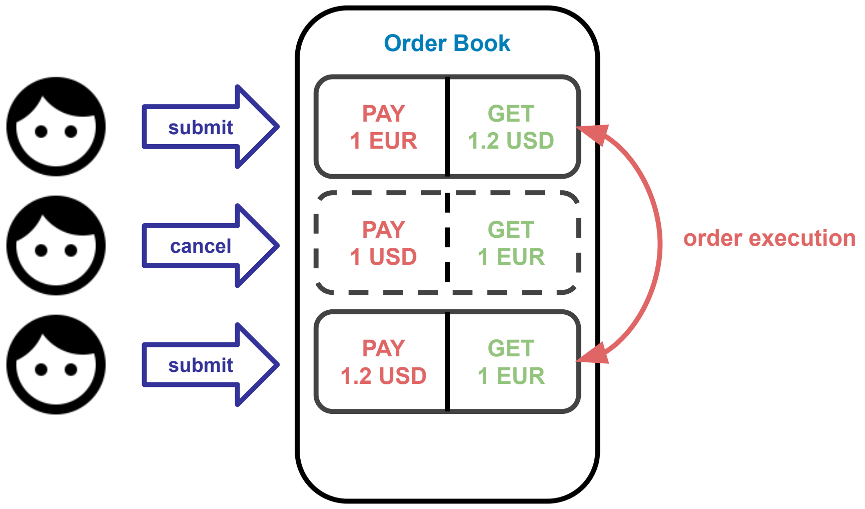


Fig. 1.5 Ripple order book.

Payments describe the movement of funds from a source account to a destination account. Ripple presents a complex collection of options to define the desired flow of funds[6]. It is possible to send money directly from source to destination or using paths of intermediaries. Users can employ one or more paths to send funds simultaneously. If no path is defined, there is a default strategy that depends on the transaction arguments. Ripple defines a path as a sequence of steps where the first one is the source, and the last one is the destination. A path step can be the debt movement from one account to another involving an issuer in the middle. This action is known as *rippling* and can only happen with the previous existence of trust lines between parties. Offers from the order book can also be used as a path step using different currencies or issuers, but the user cannot select the specific order to execute. Ripple picks the offer based on the best exchange rates at the moment of execution of the payment (when it enters in the ledger). Nevertheless, the transaction's originator can define a limit for the money he is willing to exchange. Traders in the Ripple network can take advantage of these mechanisms for *arbitrage* [126]: buying currencies with low and outdated prices from one trader and selling them to another one at a higher price. The exchange rates should be the same across issuers to have a competitive market. When there are discrepancies, arbitrage tactics can be used to obtain risk-free profit until consuming the cheap asset supply or until the price is corrected. Usually, this kind of event takes place in a short period due to high demand. To put this in practice, traders must create a *circular payment*: an operation using one or more paths with intermediaries where the source and the destination accounts

[6]https://ripple.com/build/paths/

belong to the same owner. Payments also work as an account creation mechanism when the destination is non-existent, and the amount sent is expressed in the internal currency.

1.3 Blockchain and Ethereum

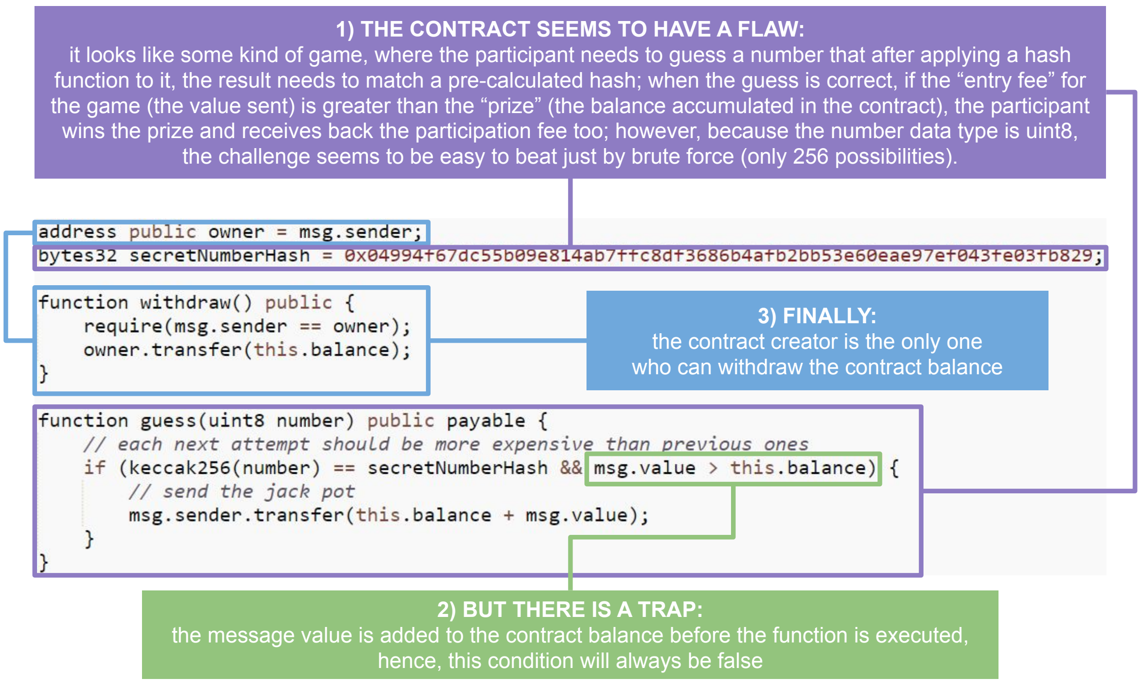
Blockchain technology revolutionized asset trading around the world. We can see a blockchain as an append-only ledger that is maintained by a decentralized peer-to-peer network. A blockchain records the transfer of assets in transactions and groups them into the so-called *blocks*. Cryptographic algorithms link blocks to their predecessors, thereby forming a "blockchain". The network participants store this chain of blocks, where a consensus protocol dictates which one to append next. Bitcoin [107] and Ethereum [148] are currently the two most prominent blockchain implementations. Blockchain is considered one of today's most significant innovations since it allows decentralizing any asset's governance. While the Bitcoin blockchain's purpose is to decentralize cash governance, thereby the banks' role, the Ethereum blockchain's goal is to decentralize the computer. As of the time of writing, Ethereum has a market capitalization of more than u$d 22 billion[7].

Ethereum builds on top of Bitcoin's principles and offers the execution of so-called *smart contracts*. Smart contracts are Turing-complete programs that are executed by the participants of the Ethereum peer-to-peer network. A 160-bit address identifies them, and they are deployed or invoked via transactions. We distinguish between external and internal transactions. External transactions refer to transactions initiated by user accounts, whereas internal transactions refer to operations triggered by smart contracts when they react to external transactions. In other words, every internal transaction has an external one as its origin. Smart contracts may interact with Ethereum's native cryptocurrency called *ether*, and receive data input from participants. Ether is a unit, and we can divide it into additional subunits, where the smallest one is called *Wei*. The blockchain enforces the execution of smart contracts precisely as they are programmed, without any possibility of censorship. The execution of smart contracts costs *gas*. Users can determine how much gas they want to provide to the execution of a smart contract (gas limit) and how much they want to pay per gas unit in terms of ether (gas price). If the execution of a transaction consumes the available gas before it ends, it terminates by throwing an error and reversing its effects.

Smart contracts are usually programmed using a high-level programming language, such as Solidity[8]. As with any program, smart contracts may contain bugs [8]. Attackers have already targetted several smart contracts in the past. The two most prominent hacks, the

[7]https://coinmarketcap.com/currencies/ethereum/

[8]https://solidity.readthedocs.io/en/v0.5.11/



1) THE CONTRACT SEEMS TO HAVE A FLAW:
it looks like some kind of game, where the participant needs to guess a number that after applying a hash function to it, the result needs to match a pre-calculated hash; when the guess is correct, if the “entry fee” for the game (the value sent) is greater than the “prize” (the balance accumulated in the contract), the participant wins the prize and receives back the participation fee too; however, because the number data type is uint8, the challenge seems to be easy to beat just by brute force (only 256 possibilities).

```
address public owner = msg.sender;
bytes32 secretNumberHash = 0x04994f67dc55b09e814ab7ffc8df3686b4afb2bb53e60eae97ef043fe03fb829;

function withdraw() public {
    require(msg.sender == owner);
    owner.transfer(this.balance);
}

function guess(uint8 number) public payable {
    // each next attempt should be more expensive than previous ones
    if (keccak256(number) == secretNumberHash && msg.value > this.balance) {
        // send the jack pot
        msg.sender.transfer(this.balance + msg.value);
    }
}
```

3) FINALLY:
the contract creator is the only one who can withdraw the contract balance

2) BUT THERE IS A TRAP:
the message value is added to the contract balance before the function is executed, hence, this condition will always be false

Fig. 1.6 Honeypot example in Solidity programming language.

DAO hack [130] and the Parity wallet hacks [117] caused together a loss of over $400 million. As a result, attackers started looking for vulnerable smart contracts [8], and the community proposed several tools to scan different types of vulnerabilities [91, 137, 138]. Interestingly, recent works show that attackers started deploying vulnerable-looking smart contracts to lure other attackers into traps [38, 89]. This new type of fraud is known as “honeypots” [129, 125]. Honeypots are smart contracts that appear to have a flaw in their design. This flaw typically allows any arbitrary user on the Ethereum blockchain to drain the contract’s funds. However, once a user tries to exploit this apparent vulnerability, a trapdoor obfuscated into the contract prevents the draining of funds from succeeding. The attempt to exploit these honeypot contracts does not come for free. Their exploitation always depends on the transfer of a certain amount of ether to the contract. The idea is that the victim solely focuses on the apparent vulnerability and does not consider the possibility that the contract might be hiding a trap inside. Moreover, to facilitate the detection of their “vulnerable" smart contract, honeypot creators often upload the source code to Etherscan [9], an online platform that allows anyone to navigate through the Ethereum blockchain and to publish the source code of any smart contract.

Torres et al. [38] present a tool called HoneyBadger that applies symbolic analysis to contract bytecode and relies on handcrafted rules to detect honeypots. Unfortunately,

[9]https://etherscan.io/

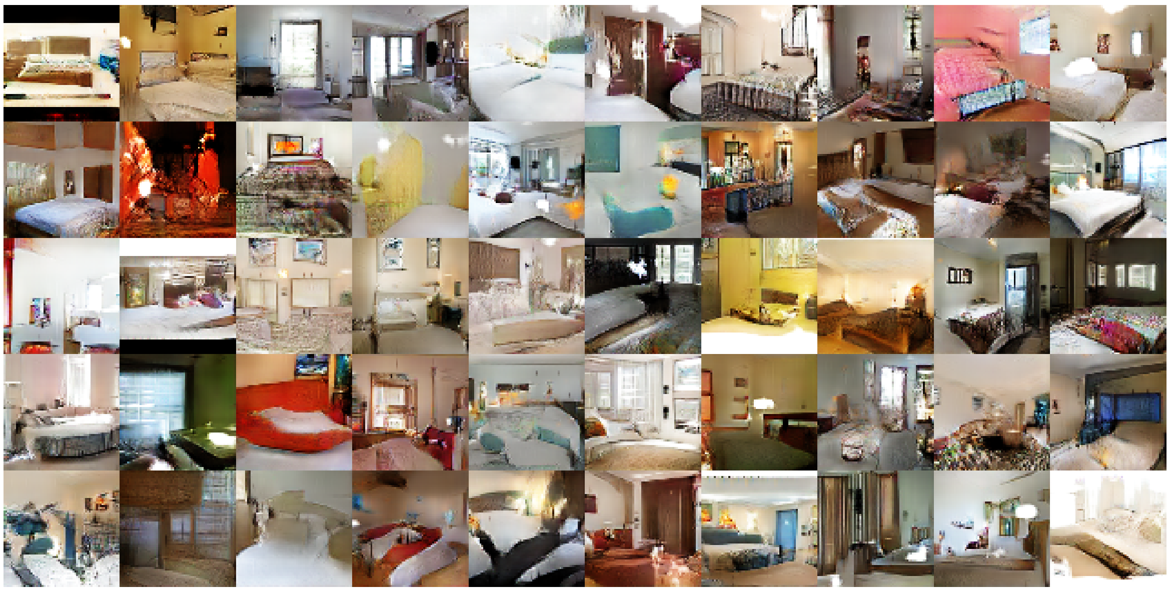


Fig. 1.7 Hotel room images generated with DCGAN.

this approach cannot recognize new honeypot techniques unless experts know them and implement specific detection rules. Moreover, HONEYBADGER only relies on the bytecode of a smart contract and does not consider transaction data. However, transaction data reflect user behavior, which might be useful to detect honeypot techniques.

1.4 Deep Generative Models

Deep generative models (DGMs) are algorithms that can learn to draw samples from the same distribution of a dataset implementing deep learning techniques. These models' key idea is that they employ a considerably smaller amount of parameters than the amount of data they use for training. Hence, to generate data points similar to the examples provided to them, DGMs cannot memorize samples. Instead, they need to capture the underlying concepts that define the training data.

The most popular DGMs in the field of computer vision, natural language processing, and speech recognition are currently variants of generative adversarial networks (GANs) [50] and variational autoencoders (VAEs) [66]. For example, in Figure 1.7 we present samples synthesized by DCGAN [119]. The authors trained the model based on pictures of hotel rooms with a resolution of 64x64 pixels. Later they utilized the trained model to generate the presented images, taking as input on each case 100 random numbers drawn from a uniform distribution. At first glance, the synthetic images look credible enough, but when closely inspected, they resemble something coming from a dream. However, many advances from

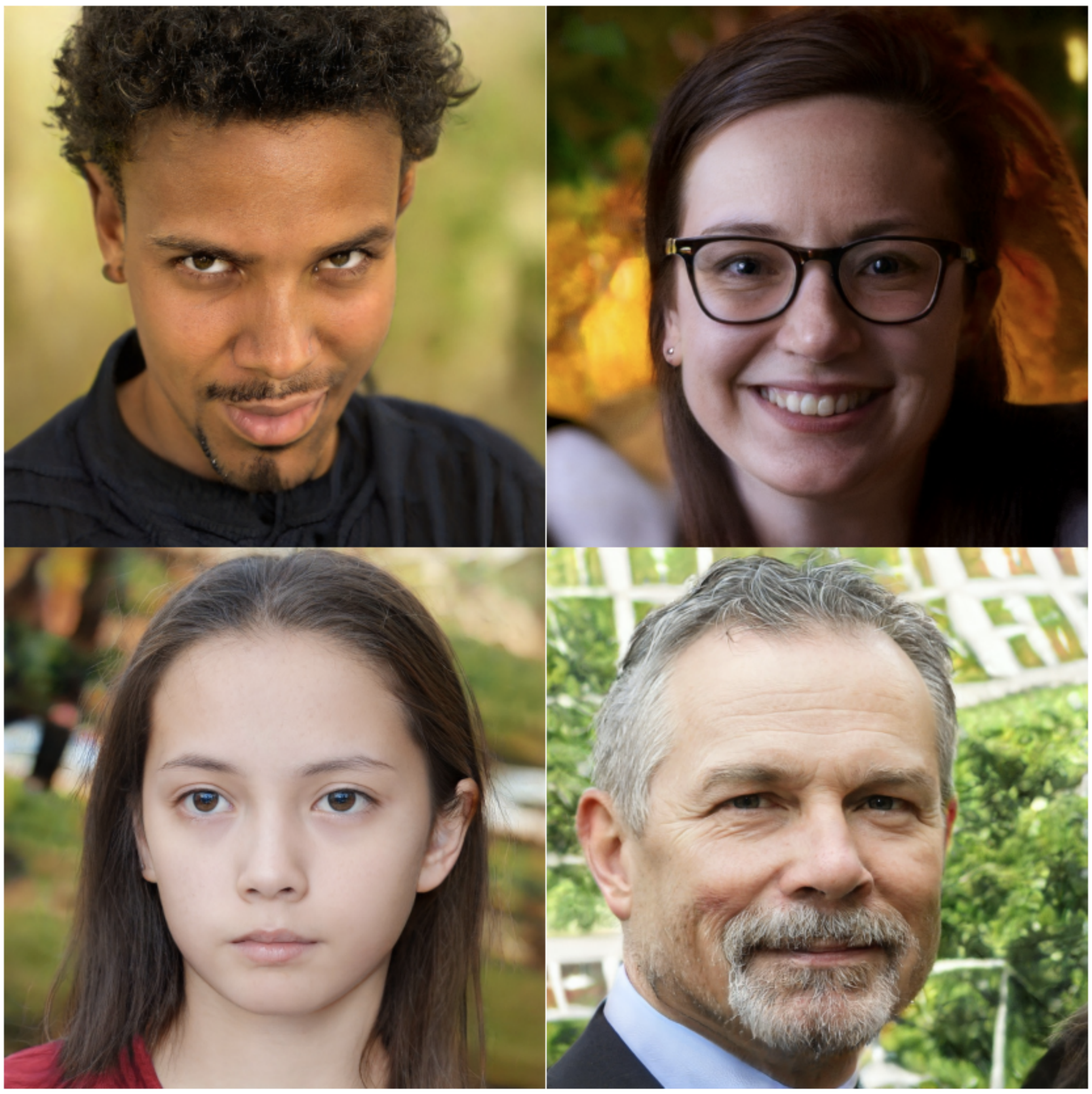


Fig. 1.8 Face images generated with StyleGAN2.

the past six years led to significant improvements in the generated images' realism and resolution. Another example shown in Figure 1.8 includes the faces of imaginary people with a resolution of 1024x1024 pixels synthesized with StyleGAN2 [61]. There is also a website that allows any user to reproduce these experiments and generate random faces of people that do not exist[10]. Many studies experimented with different methods to control sample synthesis. For example, in Figure 1.9 we can see an example where a VAE is used to generate sentences by interpolating between the latent codes of two input sentences [16]. Note that

[10]https://thispersondoesnotexist.com/

“ i want to talk to you . ”
“i want to be with you . ”
“i do n’t want to be with you . ”
i do n’t want to be with you .
she did n’t want to be with him .

he was silent for a long moment .
he was silent for a moment .
it was quiet for a moment .
it was dark and cold .
there was a pause .
it was my turn .

Fig. 1.9 Sentence interpolation using the latent space of a VAE.

the sentences are not only grammatically correct but are also semantically related between the neighboring sentences. In Figure 1.10 we included an example of the more recent and popular GPT-3 [17], a model that attracted a lot of attention from the media lately[11][12][13][14]. We can see how, based only on a title and a subtitle, the model can generate a short article that is grammatically correct, captures the intended topic, uses the names presented to it, and even employs quotes and other complex literary devices. GPT-3 is based on the transformer architecture and belongs to a different family of generative models than GANs and VAEs called *auto-regressive models*. The main secret behind this model’s success is training it with a large number of parameters (175 billion, ten times more than any previous non-sparse language model) and a large amount of data (around 570GB of plain text or 400 billion tokens). The creators released a paid API to use GTP-3[15].

With all the recent advances in DGMs, it is tempting to apply them as tools to problems in data science tasks. While these models work well on continuous-domain problems such as images, sound, and video, they struggle generating samples from mixed numerical and categorical distributions. The reason for this is partly due to the inherent difficulty of training deep learning models with discrete outputs: sampling from discrete distributions is a non-differentiable operation, which makes it impossible for gradients to flow through the networks using backpropagation. Another evident difference between the presented examples and problems in the data science domain is the number of available samples and the feature

[11]https://www.forbes.com/sites/robtoews/2020/07/19/gpt-3-is-amazingand-overhyped/
[12]https://www.nytimes.com/2020/07/29/opinion/gpt-3-ai-automation.html
[13]https://fortune.com/2020/07/28/how-a-i-is-helping-doctors-triage-patients-in-urgent-care/
[14]https://www.wired.com/story/ai-generated-text-is-the-scariest-deepfake-of-all/
[15]https://openai.com/blog/openai-api/

```
Title: Star's Tux Promise Draws Megyn Kelly's Sarcasm
Subtitle: Joaquin Phoenix pledged to not change for each awards event
Article: A year ago, Joaquin Phoenix made headlines when he appeared on
the red carpet at the Golden Globes wearing a tuxedo with a paper bag over
his head that read, "I am a shape-shifter. I can't change the world. I
can only change myself." It was a promise to not change to fit into the
Hollywood mold: "I think that's a really special thing, to not change
yourself. I think it's a really special thing to say, 'This is what's
inside of me, I'm proud of it, and I'm not going to be ashamed because of
the way that someone else thinks I should be.'" Now, it's the Oscars, and
Phoenix is at it again. But this time, his publicist is saying he'll be
wearing a tux no matter what.
Megyn Kelly was not impressed, and she let him have it on The Tonight Show.
"You know, I feel like, I feel like you could have worn the tux," she says.
"But you're saying you're a shape-shifter. I don't know if you can change
your tux, but you can change your mind. You can change your mind. You can
change your mind." Phoenix says he did, but it didn't stick. "I was like,
'Okay, I'm going to wear a tuxedo to this thing.' And then I thought, 'I
don't want to wear a tuxedo to this thing.'" Kelly goes on to encourage him
to change his mind again, but Phoenix says it's too late: "I'm committed to
wearing this."
```

Fig. 1.10 An article generated with GPT-3 given a title and a subtitle.

space's size. We need to address these limitations to use DGMs for imputation, simulation, feature extraction, transfer learning, or sampling artificial data points.

1.5 Research Questions

- Can we define a robust analysis and detection methodology for different scenarios in the absence of labels or ground truth?
- Can the methodology be extended when labels are available and generalize well even in the presence of imbalanced classes?
- Is it possible to generate better simulations based on real data?
- How can we assess the quality of the synthetic data?
- Can we enhance the imbalanced classification with synthetic data?

1.6 Thesis Structure

- The current chapter presents the introduction of the topic.

- Chapter 2 presents background information and a literature review.
- Chapter 3 presents the unsupervised approach to the problem with two use cases, one related to a private company and another involving the Ripple network.
- Chapter 4 presents the supervised approach to the problem with a use case involving the Ethereum Blockchain.
- Chapter 5 describes how to generate tabular data with deep generative models and how to measure the quality of resulting samples.
- Chapter 6 discusses if it is possible to use data generated by deep generative models to improve the performance of imbalanced classification tasks.
- Chapter 7 closes the dissertation with a discussion, the conclusions, and future work.

1.7 Publications

1. Ramiro D. Camino, Christian A. Hammerschmidt, and Radu State. Working with Deep Generative Models and Tabular Data Imputation. Workshop on the Art of Learning with Missing Values (Artemiss) hosted by the 37^{th} International Conference on Machine Learning (ICML), 2020. **Part of Chapter 2**.
2. Ramiro D. Camino, Leandro Montero, Petko Valtchev, and Radu State. Finding Suspicious Activities in Financial Transactions and Distributed Ledgers. IEEE International Conference on Data Mining Workshops (ICDMW), 2017. **Main text of Chapter 3**.
3. Ramiro D. Camino, Christof Ferreira Torres, Mathis Baden, and Radu State. A Data Science Approach for Honeypot Detection in Ethereum. International Conference on Blockchain, 2020. **Main text of Chapter 4**.
4. Ramiro D. Camino, Christian A. Hammerschmidt, and Radu State. Generating Multi-Categorical Samples with Generative Adversarial Networks. Workshop on Theoretical Foundations and Applications of Deep Generative Models hosted by the 35^{th} International Conference on Machine Learning (ICML), 2018. **Main text of Chapter 5**.
5. Ramiro D. Camino, Christian A. Hammerschmidt, and Radu State. Oversampling Tabular Data with Deep Generative Models: Is it worth the effort? ICBINB Workshop hosted by Neural Information Processing Systems (NeurIPS 2020).
   Main text of Chapter 6.

Chapter 2

Background

2.1 Machine Learning

Machine Learning (ML) is the sub-field of Artificial Intelligence that provides computer programs with the ability to improve from experience without being explicitly programmed [104]. *Deep Learning* (DL) is a particular kind of machine learning that achieves great power and flexibility by learning to represent the world as a nested hierarchy of knowledge, with each concept defined by simpler ones and more abstract representations computed in terms of less abstract ones [49]. *Artificial intelligence* (AI) is a broader concept than machine learning, which addresses the use of computers to mimic the cognitive functions of humans [122]. Nevertheless, once a problem is "well studied", and society incorporates computers to solve a particular task, the program or technique usually leaves the AI category. For example, the first psychiatrist chatbot called ELIZA [59] was considered at the time as an AI, even if it just consisted of handcrafted rules using regular expressions. For today's standards, nobody would classify ELIZA as an AI. Fuzzy logic and different tree search algorithms went through the same path at some point [122]. Today machine learning and deep learning are in vogue (again) [49], hence it is tough to find AI algorithms outside these categories, but it could change in the future. The machine learning discipline is closely related to other fields such as statistics, data science, data mining, and business intelligence. Some differences between statistics and machine learning are that the latter sometimes can find relationships that do not have a statistical justification. Additionally, machine learning focuses on computational aspects such as performance, memory consumption, and hardware implementations (e.g., GPUs) or distributed algorithms. Data science, data mining, data analytics, and business intelligence are disciplines that draw techniques from different fields like mathematics, statistics, computer science, and information science, intending to discover insights from real-world data, presenting reports and possibly making business decisions based on the gathered information. The line separating those disciplines is quite blurry. Some practitioners argue that one of those concepts is more general than the others, while other experts suggest that data science is more focused on the scientific aspects, and the rest emphasize the business processes. Nevertheless, many of these terms sometimes can be interchanged.

In the following part of this section, we can summarize the basics of machine learning and related techniques we employ throughout the dissertation. Supervised machine learning algorithms can identify patterns that relate *features* (measurable characteristics of the data) to *labels* (a particular property of the data). The *learning* or *training* process occurs when the algorithms search for patterns based on examples for which the labels are known. The result is a *model*, an approximation of the underlying relationship between features and labels. During the *testing* step, the model assigns *predictions* to samples that were not part of the training phase, and we compare these assignments to known labels. The purpose is to evaluate how well the model generalizes to unseen cases. If each label represents one option from a set of possible *classes*, we define a supervised machine learning problem as a *classification*, but if the labels are continuous values, we define it as a *regression*. *XGBoost* [25] is a famous tool among the data science community for supervised learning, which is a fast implementation of an older algorithm known as gradient boosted machines [41]. An *ensemble* is a model that combines the results from several other models to correct every case's weaknesses. Most of the alternatives to this technique can be classified into *bagging* or *boosting* [12]. A boosting algorithm is an ensemble of sequential models where each model corrects the previous one's errors. In particular, gradient boosting refers to the technique of minimizing the prediction residuals using gradient descent. The default and most common ensemble members for XGBoost are Decision Trees [12], which possess a high variance that boosting can correct. Like many other machine learning models, XGBoost outputs for each sample the probability of belonging to each class. For binary classification, the decision comes from splitting the predicted probability in two with a 0.5 threshold.

Anomaly detection is a technique that allows separating cases from a sample that deviates from normal behavior. A straightforward approach in anomaly detection is to take every variable and calculate the *z-score* of each sample, which is related to the distance of the point to the mean measured in standard deviations. The problem is that defining the anomaly score based on each dimension separately ignores the complex interactions between features. There is an extension to this technique for multiple dimensions involving the Mahalanobis distance, but it assumes that the data distribution is normal, and sometimes it is not the case. The *Isolation Forest (IF)* algorithm [80] is based on the idea that an individual that is easier to separate from the rest in random subdivisions of the feature space must be an anomaly. It starts by drawing a sample from the dataset and takes a dimension at random. It then takes a random value inside the range of that dimension to divide the sample into two parts. With the chosen dimension and splitting point, the algorithm creates the root node of a tree. More nodes are created recursively for subsamples until subdividing is impossible or reaching an arbitrary tree depth. In this tree, a point that is closer to the root node corresponds to a case that is more prone to be isolated, but this phenomenon could be caused just by chance. The algorithm then repeats the whole tree creation method for new samples until obtaining a desired number of trees. Finally, it computes the average traversal path length among all the trees as the anomaly score. In [80] the authors claim that the algorithm can tackle *masking* problems (clustered anomalies) and *swamping* issues (wrongly classifying standard cases for being surrounded by anomalous ones) better than other alternatives. On the other side, it cannot handle well high-dimensional spaces because when the number of features increases, the chance of getting the wrong dimension to split the sample is bigger [2]. *One Class SVM (OCSVM)* [127] is an unsupervised variant of the classical *Support Vector Machine (SVM)* where a boundary is created around normal samples to separate them from the anomalous ones. It can find non-linear boundaries with kernel functions, but if too many anomalies are present in the training dataset, they can degrade the algorithm's accuracy [2]. When using *Gaussian Mixture Models (GMM)* [2], one must assume that a combination of Gaussian distributions generated the data. These components' parameters are inferred from the dataset using the *Expectation-Maximization (EM)* algorithm. We can test a sample against every component, obtaining a membership probability, and we can define an anomaly as a sample that is very unlikely to be generated by the mixture. The technique's advantage is that it can work with different types of datasets, only using other distributions in the mixture. On the other side, the disadvantage is that we must make assumptions about the generative process's nature, which beats the purposes of understanding and diagnosing strange behaviors. As mentioned before, every anomaly detection algorithm has advantages and disadvantages. The authors in [28] tested different anomaly detection ensembles, but they claim that taking a simple average score between different algorithms is a simple and effective solution.

For more detailed definitions, please refer to the literature covering topics like machine learning and pattern recognition [104, 12], anomaly and outlier detection [2], artificial intelligence [122], natural language processing and speech recognition [59] or deep learning [49].

2.2 Fraud Detection

In this section, we group machine learning, data mining, or data science studies dedicated to detecting fraudulent activities or suspicious behavior. The analyzed criminal schemes involve credit cards, health care and insurance claims, telecommunications, identity theft, tax evasion, and plagiarism. All these topics share common behavioral patterns with money laundering [9]. Please refer to [118] for an extensive survey on data mining for fraud detection.

2.2.1 Anti-Money Laundering

This section presents a small survey of anti-money laundering studies, including the most notorious publications in the past 15 years. We classify the articles into architecture, surveys, theory, and experiments. Figure 2.1 exhibits the number of publications by year following this categorization, where the experiment category is the most predominant. Architecture articles [44, 144, 82, 152, 43, 158, 63, 4, 52] propose designs of anti-money laundering systems with different desirable properties, both in streaming or batch mode, that require for example to handle large volumes of transactions at a high-speed rate. Surveys [13, 145, 46, 74, 131, 121, 88, 123] include a collection of work from several authors in the domain, some of them with statistics or comparisons, precisely like this section. Theory papers propose mathematical models [155, 83, 11] or application concepts [42, 81] for money laundering detection but without any empirical validation. The last category, the experiment studies, are the main interest of this section. These studies involve some hypotheses and modeling tested against data. About 67% of the articles in the experiment category involve private data, either from a single [142, 92, 71, 75, 143, 72, 73, 84, 70, 19, 101, 102, 27, 5, 60] or multiple [99, 36] anonymous banks for which only the country is disclosed, a government institution from a known country [134, 62, 132, 115], or an anonymous private company [85, 120, 3, 30, 161]. Around 51% of the publications (which we describe below) use synthetic data to run their experiments. Furthermore, only 17% of the experimental studies employ both private and generated data simultaneously, and sadly none of them involve publicly available datasets or provided open data along with the publication.

One of the most famous studies from the experimental category [134] works with 80 simulated accounts mixed with 5000 real ones from the Wuhan Branch of the Agriculture Bank in south-central China, collecting around 1.2 million records over seven months. However, the data generation is unspecified: the authors just describe the fraudulent accounts as “obviously deviated” data points. Furthermore, there is no baseline to compare against, and the detection rate of anomalies is lower than 70%. A similar study presented in [92] works with 6000 accounts from an anonymous bank, collecting one million records over eight months, and generates anomalous cases by adding “sin noise” to existing samples (not well specified). The anomaly detection is carried out by RBF neural networks and compared against OCSVM. Another article [45] mixes transactions from an anonymous bank with synthetic fraudulent cases. This article also ranks transactions from the most to the least suspicious based on a clustering algorithm known as Local Outlier Factor (LOF) [2].

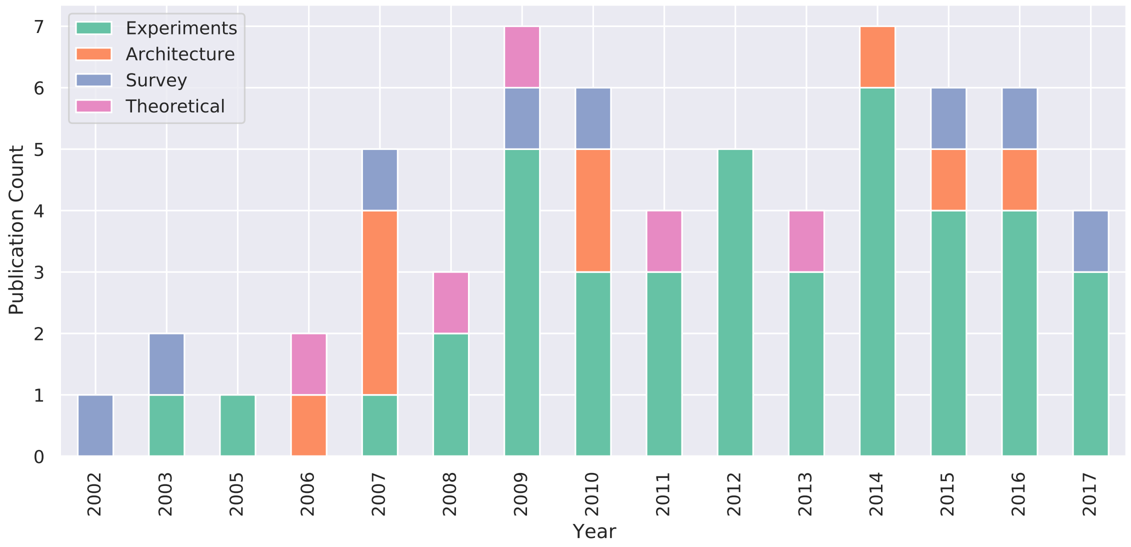


Fig. 2.1 Amount of Anti-Money Laundering papers in the past 15 years.

However, the data generation process is not well specified, and there are no baselines to compare the results. The authors in [86] propose a money-laundering simulator, generating transaction attributes in a configurable range to create known money laundering patterns along the placement, layering, and placement stages. The study does not employ real data, it does not specify the data generation (no details about distributions), and it does not contain any experiments. However, the authors continue their work in [87], where they present a more detailed data generation: the distribution depends on configurable Markov models, and the transaction amount depends on the account age. Detection algorithms are carried out with Weka [147], comparing Random Tree, C4.5, Random Forest, and JRip algorithms against Naive Bayes as the baseline. Weka is also employed in [160], in this case implementing the PART, C4.5, and Random Forest algorithms to detect accounts owned by *smurfs* or *money mules*. Fraudsters recruit the owners of these accounts as intermediaries who accept money from one fraudster and forward it to another one for a fee. The authors of [160] generate suspicious cases at random based on known “habits” or transaction patterns (e.g., a chain from one criminal to another with several intermediate smurfs). Nevertheless, the algorithm only captures long chains, and the information about data distributions is missing. In [120] the authors discretize transaction amounts in bins and use them to detect mules by transitioning between states in Markov models. A different model is built per role to classify users (e.g., regular users versus mules). However, the study has several weaknesses: the authors make strong assumptions about the data; the authors propose a very simplistic model (using only the transaction amount); the scenarios are minimal; many distribution details are missing; the results contain a considerable amount of false positives, and the authors have no baseline to compare. To some degree, all the enumerated anti-money laundering publications work with similar features. The attributes that typically describe individual transactions are the type, date, currency, amount, sender, and receiver. Additionally, the aggregated transaction properties shared across studies include counts or frequencies of withdrawals and deposits, ratios between withdrawals and deposits, and aggregated values (min, max, sum, mean, std) for the transaction amount the time between transactions.

2.2.2 Market Abuse Detection

The term *market abuse* refers to practices surrounding the financial market that put some investors under an unreasonable disadvantage against others. These practices are deemed illegal in several parts of the world. Some studies implement machine learning techniques for the detection of *insider trading*, which is the trading of securities of a public company based on private information about the company. In [34], the authors search correlations between options trading information and market news using decision trees, support vector machines, and neural networks. Another study [32] covers the impact on the market originated from forum threads involved in *ramping*, which consists of trying to persuade people that shares are worth more than they are. This event may happen when investors want to sell shares at an increased price and profit quickly. Linear regressions, decision trees (C5.0), and neural networks are employed in [156], to analyze the correlation between indicators of market abuse, and the number of spam emails found that spread false rumors or illegal information (also known as *touting*). Furthermore, [133] discusses how insiders' trading behavior differs based on their companies' roles, the transaction types, the company sectors, their relationships with other insiders, and how that behavior changes over time.

Another kind of abuse of a financial market is *trade-based manipulation* (or *market manipulation*), which damages the proper functioning and integrity of the market by merely buying and selling. Price and volume are usually the two main targets to manipulate. Many studies approach this issue with anomaly detection techniques. In [21], the authors transform the time-varying financial trading data into a pseudo-stationary time series and apply machine learning algorithms to detect price manipulation. Anomaly detection over order prices is applied in [20], employing Hidden Markov Models and Gaussian Mixture Models, along with a complex feature extraction mechanism based on signal processing and differentiation. Another study [76] employs neural networks to detect *pump-and-dump* mechanisms, which involves artificially inflating the price of an owned stock through false and misleading positive statements (*pump*) to sell the cheaply purchased stock at a higher price (*dump*). Two-dimensional Gaussian Models are also employed in [76] to detect *spoofing*, a strategy to manipulate the price of an instrument through a combination of buy and sell orders. A hybrid solution between [21] and [20] is presented in [157], adding random undersampling plus the SMOTE oversampling algorithm [24] to overcome the class imbalance and the Mann-Whitney U test [95] for detecting concept drift and adapt the models. Besides anomaly detection, some other machine learning algorithms are used, like using supervised methods for the detection of market price manipulation [112, 47], generating explainable models based on financial variables for the detection of price manipulation in closing hours [33], or simulating honest versus spoofing trades with Markov Decision Processes [97]. Additional techniques that derive from graph algorithms (but not necessarily employing machine learning) include analyzing irregular trading behavior with spectral clustering [40], detecting collusive trader groups with graph clustering [113] or finding cliques [141], and the detection of special nodes like *blackholes* (very high in-degree over out-degree ratio) and *volcanoes* (very low in-degree over out-degree ratio) [78]. *Wash trade* is a particular market abuse tactic in which a group of collusive investors sells and buys the same financial instruments in parallel. This tactic creates an illusion of high demand, and brokers can unfairly profit from inflated commission fees. Although the concept and money laundering are not directly related, the patterns and detection methods fit our domain. In [22] the authors look for trading circuits where the exchanged volume returns almost entirely to the origin and the time window between submissions and executions is abnormally narrow.

2.2.3 User Behavior and Profiling

In [14], the authors separate credit card purchases in time windows, obtaining vectors of purchase volumes. Using Euclidean distance, for every client, they get the closest clients. Then, they calculate the client's distance from his group for every time step using a t-test, which they use as an anomaly score. The authors use a second approach to analyze only one client's transactions using time windows and measure the difference between consecutive time windows using a t-test as the anomaly score. Another study [58] constructs curves for eBay auction bids over time, using the data points as knots to compute splines. Then the curves are clustered using k-medoids, measuring the similarity of vector polynomial coefficients with euclidean distance. The authors of [54] build user profiles to detect frauds in telecommunication networks. One vector is computed per day, accumulating each phone call count and duration in different categories. They then concatenate those vectors using an arbitrary window size to obtain a single vector representing the user profile. The authors compute deviations in user behavior by measuring the similarity between profile vector pairs. Anomalies in sequences of system calls are detected in [151], modeling the transitions between n-grams of those sequences with a Markov Reward Process, based on labeled data and a technique from the Reinforcement Learning domain called Temporal Difference Learning. X-means is employed in [23] to cluster fraudulent users in online auctions, and decision trees are trained over the cluster labeling to obtain a more explainable model. Additionally, the experiments are repeated with different user history proportions to see if they can detect fraud in the early stages. In [140], the study represents user behavior as a set of finite actions (clicks on an interface). The time between actions is also discretized in fixed bins, obtaining traces of behavior when interleaved with the user actions. If we consider actions as symbols or characters and the traces as strings, we can transform behavior into vectors of n-gram frequencies. In the vector representation, we can translate behavior similarity into string similarity. The study uses cosine similarity to cluster users using Divisive Hierarchical Clustering, calculating the number of cluster divisions depending on the modularity measure. Furthermore, the study describes every cluster with aggregated features and compares them using the chi-squared test.

2.2.4 Criticism

In summary, we can enumerate the challenges in the fraud detection domain and the issues with the state-of-the-art techniques and studies as follows:

- It is not easy (or impossible) to reproduce the current studies:
  - Private data:
    - not available for everyone, hence, not reproducible;
    - some publications do not describe the features or their distributions;
    - some publications do not describe the feature engineering or preprocessing.
  - Synthetic data:
    - some publications have unspecified distributions;
    - some publications have strong baseless assumptions;
    - some publications have simplistic money laundering generated scenarios;
    - some publications have very small scenarios;
    - some publications only test anomalies and not necessarily money laundering;
  - Private + Synthetic data: some publications base their simulations on private data.
- There is a lack of labels, ground truth, and proper validation:
  - some publications only validate visually;

  - some publications only inspect cherry-picked cases;
  - some publications only report anomaly rankings;
  - some publications have no baselines.

- Imbalanced classes: the amount of money laundering related transactions is minimal. Nonetheless, they have to be detected because the impact is high.
- Partial information: many money laundering techniques or patterns span through several networks, and usually, data from only one system is available (e.g., one bank).
- Low quality or misleading metrics:
  - some publications have very low true positives;
  - it is common to have a high amount of false positives or false alarms.

2.3 Blockchain and Ethereum

In [38], Torres et al. proposed the first taxonomy of honeypots. They identified eight different honeypot techniques and implemented a tool, HONEYBADGER, to detect them. HONEYBADGER uses static analysis to investigate Ethereum smart contracts. Before that, in the domain of smart contracts, static analysis has been used to detect vulnerabilities [91, 137], and verify the compliance and violation of patterns [138]. While other such tools exist, they focus on smart contracts' security and not on honeypots or scams. In [10] the authors investigate scams on Ethereum (specifically Ponzi schemes). However, they analyze smart contracts' source code manually. The study in [26] presents a tool for detecting Ponzi schemes using machine learning. Machine learning has also been used in [135] to adapt to new attack patterns rapidly. Other machine learning studies in this domain emphasized transactions but for security reasons [110, 68], or to predict the bitcoin price [100].

2.3.1 Honeypot Techniques

Torres et al. [38] identified eight different honeypot techniques. In general, honeypots use bait in the form of funds to lure users into traps. After looking at the source code[1], users are tricked into believing that there is a vulnerability and that they will receive more funds than they require to trigger the vulnerability. However, it is not clear to the users that the honeypot

[1]The source code is often made publicly available to lure victims.

contains a hidden trapdoor that causes all or most of the funds to remain in the honeypot. In the following, we provide a summary of the honeypot techniques used to deceive users.

Balance Disorder (BD). This honeypot technique uses the fact that some users ignore the exact moment when a contract's balance is updated. A honeypot using this technique promises to return its balance plus the value it receives if the former is smaller than the latter. However, before executing the code, the contract adds the received value to its balance. As a result, the balance will never be smaller than the value received. An example of this type was presented in Figure 1.6 of Chapter 1.

Inheritance Disorder (ID). This honeypot technique involves two contracts and an inheritance relationship between them. Both the parent and the child contract define a variable with the same name that guards access to funds' withdrawal. Although novice users may believe that both variables are the same, the compiler will map the two variables internally to two different storage locations. As a result, changing one variable's content does not change the content of the other variable. A honeypot can implement this trick to limit the contract balance's withdrawal solely to the contract creator.

Skip Empty String Literal (SESL). A bug previously contained in the Solidity compiler skips hardcoded empty string literals in function calls' parameters. Thus, the compiler shifts the parameters following the empty string literal upwards. Honeypots can use this to redirect funds to the attacker instead of sending it to the victims.

Type Deduction Overflow (TDO). Although developers usually declare variables with specific types, type deduction is also supported[2]. A type is then inferred, without the guarantee that it is sufficiently large. Honeypots can abuse this mechanism to cause integer overflows that prematurely terminate execution, deviating from the expected behavior.

Uninitialized Struct (US). Structs are a convenient tool in Solidity to group variables under a common namespace. However, if developers try to modify an uninitialized struct, the EVM writes by default the beginning of the storage. This mechanism can overwrite existing values and modify the contract's state. Thus, variables required to perform the transfer of funds may be overwritten and thereby prevent the transfer.

Hidden State Update (HSU). Internal transactions are a consequence of external ones, but blocks only store the latter. However, both can cause changes in a smart contract's state. This honeypot technique assumes that users mostly pay attention to external transactions and that an internal one may go unnoticed. For example, Etherscan does not display internal transactions that do not transfer any value. A honeypot can use this effect to hide state updates from users and make them believe the honeypot is in a different state.

[2]Until compiler version 0.5.0 https://solidity.readthedocs.io/en/v0.5.0/.

Hidden Transfer (HT). Platforms such as Etherscan allow users to view the source code of published smart contracts. However, due to the limited size of the source code window (and given that there is no line-wrapping), publishers can insert white spaces to hide the displaying of certain source code lines by pushing them outside the window. This property allows publishers to inject and hide code that transfers funds to them instead of users.

Straw Man Contract (SMC). When uploading a contract's source code to Etherscan, only the contract itself is verified and not the code of other contracts that the contract may call. For example, a source code file may contain two contracts, where the first contract makes calls to the second contract. The second one, however, is not used after deployment. Instead, it is a straw man standing in the place of another contract that selectively reverts transactions.

2.3.2 Criticism

HONEYBADGER is a form of a rule-based system built to detect honeypots among Ethereum smart contracts. As mentioned before, the rules that this software employs depend solely on static byte code analysis. One of the weaknesses of this approach is that it ignores any pattern derived from transaction behavior. Nevertheless, most importantly, given that the system developers crafted rules to capture each specific technique, the system would not detect any instance belonging to a technique unknown to them.

2.4 Deep Generative Models for Tabular Data

Unsupervised learning from datasets is of interest in many fields, serving different purposes. Examples include inference tasks as in [96], where the authors learn Bayesian models using non-parametric priors over multivariate tabular data, as well as unsupervised feature discovery, extraction, and transfer [64, 119, 56], and synthetic data for privacy-aware applications [29, 105]. On continuous data, Generative Adversarial Networks (GAN) [50] have proven to be good at learning data distributions in an unsupervised fashion. By feeding in additional information (such as labels), conditional GANs [103] can learn to generate samples for specific inputs. On discrete data (e.g., text sequences, categorical data), GANs face problems leading to comparatively worse performance. However, several approaches exist to deal with this. The Gumbel-Softmax [57] and the Concrete-Distribution [94] were simultaneously proposed to tackle this problem in the domain of Variational Autoencoders (VAE) [66]. Later, [69] adapted the technique to GANs for sequences of discrete elements. Addressing the same problem, a reinforcement learning approach called SeqGAN [154] interprets the generator as a stochastic policy and performs gradient policy updates to avoid the backpropagation problem with discrete sequences. The discriminator outputs the reinforcement learning reward for a full sequence and executes several simulations generated with a Monte Carlo tree search to complete the missing steps. An alternative approach that avoids backpropagating through discrete samples, called Adversarially Regularized Autoencoders (ARAE) [159] transform sequences from a discrete vocabulary into a continuous latent space while simultaneously training both a generator (to output samples from the same latent distribution) and a discriminator (to distinguish between real and fake latent codes). The approach relies on Wasserstein GAN (WGAN) [6] to improve training stability and obtain a loss more correlated with sample quality. MedGAN [29], while architecturally similar, is used to synthesize realistic health care patient records. The authors analyze the problem of generating simultaneously discrete and continuous samples while representing records as a mix of binary and numeric features. The method pre-trains an autoencoder, and then the generator returns latent codes as in the previous case, but they pass that output to the decoder before sending it to the discriminator; therefore, the discriminator receives either fake or real samples directly instead of latent codes. They propose using shortcut connections in the generator. Additionally, the authors present the mini-batch averaging technique to evaluate better the generation of a whole batch instead of individual isolated samples. Before feeding a batch into the discriminator, mini-batch averaging appends the batch's average value per dimension to itself. One study [51] presents an improved version of WGAN to address the difficulty of training GANs. It adds a gradient penalty to the critic loss (WGAN-GP) and removes the critic parameters' size limitation. Tabular GAN (TGAN) [150] presented a method to generate data composed of numerical and categorical variables, where the generator outputs variable values in an ordered sequence using RNNs.

2.4.1 Classification with Imbalanced Classes

Some studies already experiment with the possibility of using deep learning models as a preprocessing technique to improve classification tasks where the amount of samples per class is highly imbalanced (e.g., when the size of the minority class is less than 10% of the size of the majority class). The method known as Dual Autoencoders (DualAE) [109] consists of training two stacked autoencoders in parallel using sigmoid and tanh activations. The method then encodes each sample with two models, and the concatenation of both codes can replace the original features. This procedure is a form of representation learning that aims to improve one model's synthesis power by combining two models that might capture different information. Stacking autoencoders was a common technique to train a concatenation of single layer autoencoders in separated incremental steps, but they were replaced over time by deep autoencoders [49]. We can also find a few approaches specifically implementing deep generative models to oversample the minority class for posterior classification. For that goal, the authors in [35] propose an oversampling approach using conditional GANs, and in another study [39], the authors train a simple GAN only with the minority class from a credit card fraud dataset. Furthermore, the authors of [149] transformed TGAN into a conditional model (CTGAN) to generate samples from the minority class. Finally, HexaGAN [55] proposed to train three models in alternating steps to compute a unified objective function: a conditional GAN for oversampling, an imputation model for missing values, and a classification model.

2.4.2 Missing Data Imputation

There are numerous studies related to image completion with DGMs like [139] that uses denoising autoencoders for image imputing. In the natural language processing domain, [16] presented a VAE model with a recurrent architecture for sentence generation and imputation. The concept of imputing missing values with deep generative models also exist for tabular data. Generative Adversarial Imputation Nets (GAIN) [153] adapted the GAN architecture to this problem. The generator outputs imputed samples from inputs with random noise replacing the missing values. The discriminator then tries to predict which values were imputed and which values are original. The authors of Multiple Imputation Denoising Autoencoders (MIDA) [48] initially replace missing data by the mean or the mode of the corresponding feature. Then they send the first imputation through a denoising autoencoder (implemented with a dropout on the input layer). The data reconstruction coming from the denoising autoencoder represents the final imputation. Additionally, several models are trained with different random initialization to achieve multiple imputations. Importance Weighted Autoencoders (IWAE) [18] presents a generalization of the evidence lower bound defined in VAE that can approximate asymptotically better the posterior distribution. Two studies, Heterogeneous-Incomplete VAE (HI-VAE) [108] and Missing data IWAE (MIWAE) [98], extended the work of IWAE to the field of multiple data imputation by separating variables into missing and observed. Additionally, HI-VAE presents a more extensive collection of losses that depend on the type of each variable.

2.4.3 Criticism

The acquisition of knowledge is an iterative process in the scientific domain. Scientists need to understand, incorporate, and challenge the ideas of their pairs continuously. In this context, the ability to reproduce experiments is essential. However, modern academic times and practices can give birth to publications that leave many details to interpretation.

Table 2.1 Five UCI repository datasets including the word “breast” in their names.

Full Name	Samples	Variables			
		Predictive	Non-Predictive	Target	Total
Breast Cancer	286	9	0	1	10
Breast Cancer Wisconsin (Original)	699	9	1	1	11
Breast Cancer Wisconsin (Diagnostic)	569	30	1	1	32
Breast Cancer Wisconsin (Prognostic)	198	33	1	1	35
Breast Tissue	106	9	0	1	10

Table 2.2 DGM studies that mention the “breast” or “breast cancer” dataset.

Study	Dataset Description Source	Samples	Variables	Dataset Match
HI-VAE [108]	publication	699	10	BCW (Original)
MIDA [48]	publication	699	**11**	
GAIN [153]	supplementary materials	569	30	BCW (Diagnostic)
HexaGAN [55]	supplementary materials	569	30	
MIWAE [98]	code example (scikit-learn)	569	30	
cGAN [35]	publication	106	9	Breast Tissue
DualAE [109]	publication	**350**	9	**No Match**

Many seemingly trivial pieces of information might cause discrepancies in our results and wrong conclusions. In our everyday work, we found that small details related to the datasets, preprocessing, hyperparameters, and the reported metric can trigger a cascade of problems.

UCI Repository

In the domain of computer vision, there are very well known benchmarks based on datasets like MNIST[3], ImageNet[4], and CIFAR[5]. However, there is no popular framework to compare studies in the domain of deep learning for tabular data imputation. There is a popular source of datasets called the UCI Repository [37], but there is no standardized protocol for the usage of this resource. The problem is that the datasets are usually identified with arbitrary short names or aliases, which can lead to confusion. For example, Table 2.1 describes five datasets with similar names but different amounts of samples and features. Occasionally authors append tables like this one to their publications as supplementary material. These tables could solve dataset identification, but sometimes the numbers do not match between papers or with the information provided on the repository. Following the previous example, Table 2.2 enumerates seven deep learning studies which are supposedly employing one of the datasets mentioned above. However, one of the datasets has no match. Furthermore,

[3]http://yann.lecun.com/exdb/mnist/
[4]http://www.image-net.org/
[5]https://www.cs.toronto.edu/ kriz/cifar.html

the repository publishes some datasets like “adult”[6] separated into train and test. Using the entire set or a portion may cause discrepancies in the number of samples reported. The same occurs when the dataset contains missing values, and some authors decide to discard them, but others do the opposite. In the same way, inconsistencies in the number of features can appear when non-explanatory variables (e.g., ID numbers) are discarded or included in the description tables. We consider that the simplest solution would be to identify the datasets by the full URL on the repository. Machine learning software like *scikit-learn* [116] provides a collection of datasets to work out-of-the-box, but they are usually considered “toy datasets” because of the dimensions or the complexity of the related task. An ideal approach would be something closer to *torchvision*, the image repository provided by *PyTorch* [114].

Preprocessing and Metrics

Even if it is not always necessary, scaling numerical features is considered good practice for training deep learning models. The procedure usually affects the convergence of the training loss but also changes the magnitude of some metrics. Furthermore, the encoding of categorical variables can also affect the experiments, but studies rarely specify it. One could assume that one-hot-encoding is more reasonable because it makes no sense to measure a distance between two label-encoded categorical variables since the numbers assigned to each category are arbitrary, and they possess no valid order. Nevertheless, a one-hot-encoding can significantly increase the number of feature dimensions, leading to many problems. Besides, deep learning libraries like PyTorch [114], and Tensorflow [1] contain embedding layers for categorical variables that expect label encoded inputs but also contain softmax layers that require one-hot-encoding to handle categorical outputs. In any case, note also that one-hot-encoded categorical variables and min-max scaled numerical variables are contained in the range $[0,1]$, but in other combinations of formats, if the ranges are different, then losses like MSE might assign different weights to each type of variable during training. If all the variables are in the same range, another possibility could be to aggregate different losses per variable type like in [108], but it is harder to implement.

Hyperparameters

In most of the deep learning related papers, hyperparameters are not indicated or only partially defined. Furthermore, some studies provide their hyperparameters, but they rarely explain the logic behind the selection. Most of the deep learning setups need to define some standard settings: the batch size, the learning rate, the amount and the size of hidden layers, the

[6]http://archive.ics.uci.edu/ml/datasets/adult

optimization algorithm (and extra hyperparameters that it may have), decide if dropouts are needed, decide if batch normalization is used and determine whether any kind of parameter normalization is used (plus related weights). Additionally, for any GAN derived model, the number of training steps for each component needs to be defined, and for any model involving a latent space, the size also needs to be determined. Regarding more specific models: the parameter clamp for WGAN, the gradient penalty weight for WGAN-GP, the hint probability and the reconstruction loss weight of GAIN, and the temperature of Gumbel-softmax.

We could argue that some of these choices are not part of the hyperparameter selection but part of the architecture design. For example, the use of *LeakyReLU* in the discriminator [119]. There are also implementation decisions that have an impact on the results, but studies rarely document them. For example, there is a vast collection of “GAN training tricks” usually hidden among the implementations, some of which can be found in [124].

With this massive amount of decisions to make, even running a grid search with a small number of alternative values for each hyperparameter could cost a considerable amount of resources. Hence, if the authors of other models do not specify their configurations (or do not publish the entire code for their experiments), and an exhaustive search is not possible, selecting arbitrary values seems very tempting. However, researchers put the proper effort into searching hyperparameters for their proposals, getting an advantage over competitors with default values. Additionally, many studies do not consider each model’s capacity when comparing them (the number of trainable parameters or weights). For these reasons, one might suspect that in many articles, the competition between models is unfair. The authors of [90] suggest that with enough capacity and training time, many different GAN alternatives can achieve the same results regardless of how complex they look. Also, in the metric learning domain, [106] shows that even if articles across the years claim that the field is going steadily forward, some methods from over a decade ago can be competitive nowadays by using them properly and measuring the right thing. Useful comparisons should list and tune all parameters for the proposed method and the competitors too.

Chapter 3

Finding Suspicious Activities in Financial Transactions

3.1 Motivation

Banks and financial institutions worldwide must comply with several policies to prevent money laundering and combat the financing of terrorism. Nowadays, there is a rise in the popularity of novel financial technologies such as digital currencies, social trading platforms, and distributed ledger payments, but there is a lack of approaches to enforce the regulations mentioned above. Software tools usually detect suspicious transactions based on knowledge from experts in the domain, but as new criminal tactics emerge, detection mechanisms must be updated. Suspicious activity examples are scarce or nonexistent, hindering the use of supervised machine learning methods. In this chapter, we describe a methodology for analyzing financial information without the use of ground truth. Using an ensemble of anomaly detection algorithms, we generate a user suspicion ranking to facilitate human expert validation. We apply our procedure over two case studies: one of them related to money remittance transactions from a private company and the other concerning Ripple network transactions. We illustrate how both examples share interesting similarities and that the resulting user ranking leads to suspicious findings, showing that anomaly detection is a must in both traditional and modern payment systems.

3.2 Methodology

We now describe the methodology's general guidelines for anomaly detection on financial transactions. The first preprocessing step consists of filtering the rows from the dataset

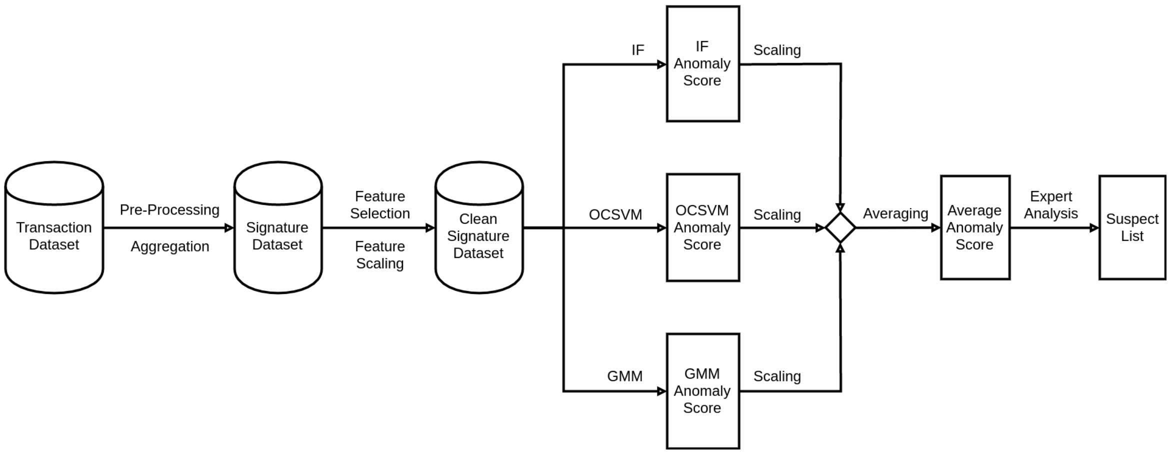


Fig. 3.1 Methodology outline.

that contain invalid values. Experts in the domain should indicate the expected format of every variable. Invalid samples could originate from software, programming, or data entry errors. In some cases, criminals intentionally forge invalid values to bypass different checks. Therefore, invalid transactions should be marked as suspicious in the early preprocessing stage and filtered out from the rest of the analysis.

We build a collection of vectors resulting from the aggregation of transactions by user. Most of the fraudulent schemes present an abnormal frequency of transactions and destinations, similar contiguous volumes, and very short delays between events. *RFM (Recency, Frequency, and Monetary)* variables can capture this kind of behavior and are widely used for other scenarios in fraud analysis [9]. Any user that contains an uncommon value in one of the RFM variables can be considered suspicious. Nevertheless, unusual combinations of variable values are more challenging to define. The computation of RFM variables from transactions results in a collection of time series per account. Each account time series' length depends on the number of transactions that the account possesses. The aggregation of RFM variables per user is necessary to compare vectors of fixed size, but it provokes an inevitable loss of information. We recommend representing each time series with more than one statistic feature simultaneously, like the median, the mean, or the standard deviation. We will provide examples of feature engineering approaches in each case study.

As we deal with an unsupervised scenario, most of the traditional feature selection techniques do not apply. The first basic check we can run consists of counting the number of values in every feature across all samples belonging to the generated dataset. An empty column should be re-designed or directly dropped because it does not contribute to any analysis or decision. We should take a similar approach with features that have almost always the same value. This situation can be easily spotted when a column presents an insignificant variance. Furthermore, we study how features affect each other, measuring the Pearson correlation coefficient [9]. Calculating this metric for every column pair gives us the possibility to drop a column when we obtain values close to 1 or -1.

Columns in the dataset might have some of their values missing. Depending on the feature's semantics, an empty value can invalidate a sample, but it can be acceptable in other cases. Nevertheless, anomaly detection algorithms can be negatively affected by missing values [2]. If the number of missing values for one feature across all the samples is greater than some predefined threshold (e.g., 80%), the feature can be considered non-informative and discarded. In the opposite case, the feature can be used by keeping the observed values and imputing the missing ones. For this chapter, we take the naive approach of filling the dataset gaps using the respective column's median.

Intervals where features are very dissimilar can also degrade the accuracy of anomaly detection algorithms [2]. We scale every column by subtracting its mean from it and then dividing by its standard deviation. After this, every feature is centered around 0 and *most of its values* reside in $[-1,1]$.

With the obtained dataset, we train the three anomaly detection algorithms using implementations from the *scikit-learn* [116] machine learning software. To do that, we need to define some model parameters in every case. For Isolation Forest (IF), we use the recommended default values from [80]: sub-sampling size $\phi = 256$ and the number of trees $t = 100$. In the case of One-Class SVM (OCSVM), we chose the *Radial Basis Function (RBF)* [127] kernel and the default parameters from the library. Finally, for Gaussian Mixture Models (GMM), we need to experiment with the number of components. We run the algorithm for several values of this parameter, and we plot them against the *Bayesian Information Criterion (BIC)* [2] of the corresponding models. This metric is not interpretable by itself, but lower values are preferred when comparing the BIC between two models. We define the number of components by looking at the point where there the curve stops decreasing significantly. This practice is known as *the elbow method* [2].

Every algorithm gives a score for every user. Those values are in different scales and are not comparable without additional treatment. We first scale the output of each algorithm in a way that the most suspicious individual in the dataset gets a score of 1 and the least suspicious one gets a value of 0 according to this formula [9]:

$$score_{new} = \frac{score_{old} - min(score_{old})}{max(score_{old}) - min(score_{old})} \tag{3.1}$$

We proceed to get a fourth score for every data point by averaging the three previous scores. We can plot the distribution of the four scores to be sure that they are generally consistent. If we obtain very different histograms, maybe the algorithms are not the right choice for the dataset, or the feature engineering should be revised. We can also compare how pairs of scores relate to each other in scatter plots, where points residing in the positive diagonal represent similar score pairs. We can expect the average to exhibit a strong correlation with individual scores, but some discrepancies can arise between other specific pairs. The goal of mixing results from different techniques is to improve the quality of the averaged score. Therefore, it is not a problem if a few samples present very different individual scores. However, in the extreme situation of a wide discrepancy between scores, alternative anomaly detection algorithms or parameters should be explored.

Finally, we use the averaged anomaly score to sort the users in a ranking. We consider the accounts at the top of the list as the most suspicious cases, and we analyze them manually with the help of domain experts. Figure 3.1 depicts an outline of the methodology.

3.3 Case Studies

3.3.1 Private Company

Our first case study is related to a private company that develops transaction monitoring products for financial institutions. The company provided us with a dataset composed of financial transactions typical of Money Remittance and Currency Exchange providers. The goal is to compare results with a rule-based detection expert system.

Dataset analysis

The dataset contains one million rows, each representing a transaction with ten different variables: the transaction ID, the source and destination IDs, the amount, and six categorical variables. For privacy reasons, every ID was adequately replaced by a hash before handling them to us, and we did not receive any personal information about the users.

Around 27% of the source accounts are different, and 59% in the case of destination accounts. The distribution of transactions per account is very long-tailed in both cases, implying that most accounts appear only a few times in the dataset, but with some extreme exceptions. These users are the first suspects and will be analyzed later. For understanding the relationship between sources and destinations, we build a directed graph where nodes represent accounts, and directed edges define transactions between them. The result is a bipartite digraph: one subset of nodes only with outgoing arcs (known as *source nodes*) corresponding to transaction originators and another subset of nodes only with ingoing arcs (known as *sink nodes*) corresponding to transaction destinies. That means that no account sends and receives funds simultaneously (there are no internal nodes). This phenomenon also

Table 3.1 Graph structure (Private Company)

	deg_{out}	deg_{in}
Mean	3.729	1.706
Std.	4.558	1.568
Min.	1	1
25%	1	1
Median	2	1
75%	5	2
Max.	397	120

Internal	Sources	Sinks
0	268090	586032

indicates that the number of transactions an account has is equal to the out-degree for source nodes and equal to the in-degree for destination nodes. Another peculiarity of the structure is that destination nodes only receive arcs from one source node. Therefore, destination nodes only have more than one transaction when they receive funds multiple times from the same source. The graph structure is described in Table 3.1.

The transactions are distributed uniformly over 15 months. Regarding the days of the week, there is a positive peak on Fridays, a negative peek on weekends, and for the rest of the days, the activity is mildly uniform. The amount is a positive floating-point number representing the quantity of money transferred between source and destination accounts. The company transformed all values to USD with conversion rates relative to the transaction's creation date to work with comparable samples. The rest of the variables describe different aspects that can change the semantic of each transaction. The code determines if the transaction is a remittance, currency exchange, or a regular wire transfer. The type and the direction semantic depend on the previous variable: remittances and wire transfers can denote sending, receiving, or canceling operations, and currency exchanges can denote buying or selling operations. The payment mode defines how the sender interacted with the company, and the delivery is the analog for the receiver. Finally, the purpose indicates why the sender initiated the transaction (e.g., bill payment or a gift).

Feature engineering

We start filtering invalid samples: transactions with undefined source or destination account, invalid formats for the date or the amount, and not listed values for categorical variables. Around 0.02% of the samples fall into this category, and we file them for further examination.

Given the transaction graph structure described in the previous section, we aggregate transactions to obtain a signature vector only for source accounts. We define two counting

Table 3.2 Anomaly Score for the users with maximum feature values (Private Company)

Feature	Score
# transactions	0.728
# destinations	0.666
median(amount)	0.599
range(amount)	0.596
median(delay)	0.463
range(delay)	0.466

features: the number of transactions and the number of different user destinations. We collect the amounts from all the user's transactions and measure the time between them, obtaining two time series. For both collections, we add as features their medians and the ranges. Finally, we create one feature for each possible value of each categorical variable holding the corresponding frequency of occurrence.

Anomaly detection

We chose the parameters for the anomaly detection algorithms, as mentioned in Section 3.2. For the GMM, we observe in Figure 3.2 how the BIC value starts decreasing slower around 20 components. Hence, we decided to use that value for our experiments. We plot histograms of scores for every algorithm in Figure 3.3. We can observe that IF gives a long right-tailed distribution, GMM shows an extreme separation between anomalies and standard cases, and OCSVM contains many jumps. However, the AVG algorithm smooths everything. In Figure 3.4, we can visualize how the scores positively correlate with each other in most cases. There is a zone of disagreement between algorithms in the left bottom corner. Nevertheless, some users are evident anomalies for all the algorithms in the right top corner, and the red color indicates that those points have a high AVG score as expected.

After obtaining the AVG score for every user, we analyze how that metric relates to each continuous feature. We can see in Table 3.2 that the account with the highest value for one feature is associated with a high score most of the time. The only exceptions are the features corresponding to the time between events, but it is reasonable to consider users with high delays as normal because most of the users in the dataset behave like that.

Besides the anomalously high values in every dimension, we also analyze each user individually in the top positions in the ranking. We see in Table 3.3 the complete list of transactions of the most suspicious user. It clearly shows an example of structuring, where consecutive operations arise in a short period, with different destination accounts and the same amount of money in USD. Another case in Table 3.4 presents several transactions at

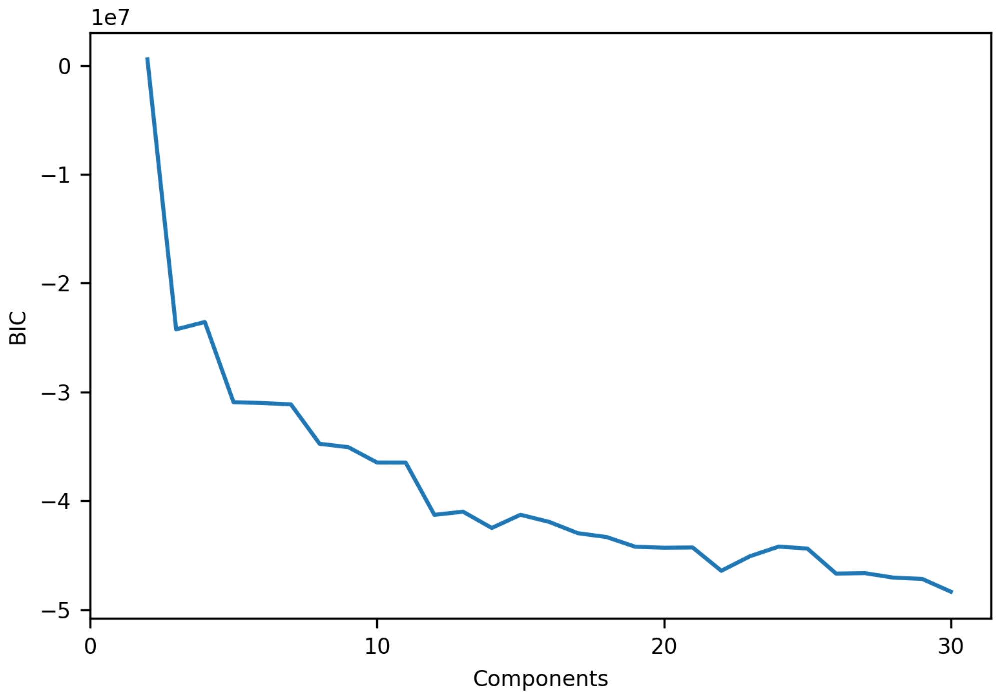


Fig. 3.2 Amount of GMM components vs. BIC score (Private Company).

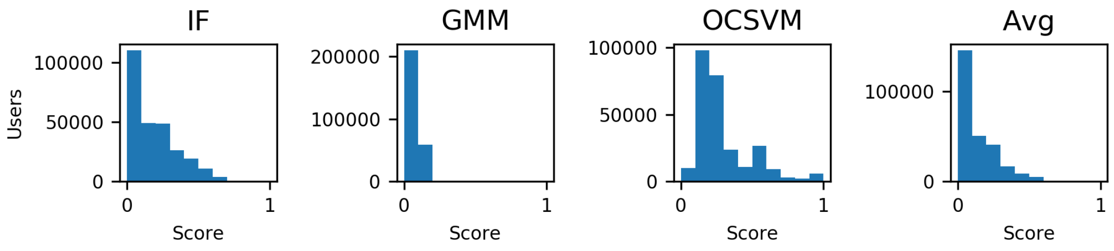


Fig. 3.3 Anomaly score distribution for different algorithms (Private Company).

the same time to the same destination with different amounts. The same account participated in 53 additional cases with a similar pattern.

Validation

We presented our data analysis and findings to the company and their experts in the domain. They explained that the peculiar graph structure we found is typical for a Money Remittance and Currency Exchange business. Most of the time, clients of that business send remittances to destinies for which financial institutions maintain no accounts and no unique identifiers.

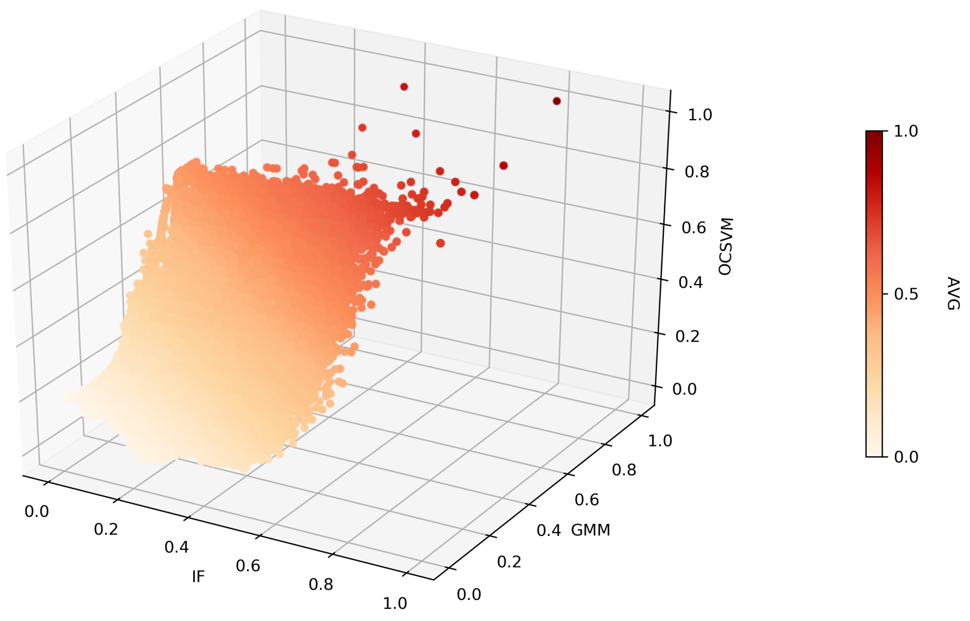


Fig. 3.4 Scores for the three algorithms (Private Company).

Table 3.3 Structuring example 1: close time, different destinations, and the same amount.

Destination	Date	Amount (USD)
user0	2014-09-11 12:51:51	13541.00
user1	2014-09-11 12:59:35	13541.00
user2	2014-09-11 13:01:05	13541.00
user3	2014-09-11 13:21:22	13541.00
user4	2014-09-11 13:27:25	13541.00

Table 3.4 Structuring example 2: same time, same destination but different amounts.

Destination	Date	Amount (USD)
user0	2014-10-18 20:08:35	1861.00
user0	2014-10-18 20:08:35	2647.00
user0	2014-10-18 20:08:35	1672.00
user0	2014-10-18 20:08:35	6.00
user0	2014-10-18 20:08:35	15.00

This issue also seems to apply to global inter-bank payments. The company knows that the set of destination users contains a significant amount of duplicates. Merging accounts that correspond to the same entity in real life is not a trivial task, given that the same person can register his name with slight changes or even with fake identities. Also, dates of birth, addresses, phone numbers, and other attributes can differ for several reasons. This problem is widely studied under many names [53], like *entity resolution* or *record linkage*.

The weekday structure is related to the working culture in the countries where the data originates. Usually, workers receive their salaries at the end of the week, and they send part of the money to their families abroad. This phenomenon is consistent with our reports.

The company checked the anomaly ranking for users and compared our results to their rule-based detection setup. Their experts verified and validated those cases manually. They confirmed that we gave high values to all the accounts the rule-based detection setup marked as suspicious. Furthermore, they informed us that they were able to spot additional suspicious schemes thanks to our methodology. They also stated that the two approaches should be seen as complementary since rule-based regulations are mandatory, and also because our method presented a few false negatives that their system could cover.

3.3.2 Ripple

Our second case study is related to Ripple. There are several tools to analyze and inspect the network with modern visualizations like Ripple Charts [1]. Another example is the online tool Bithomp[2] that allow us to identify some accounts by nicknames and inspect their transactions. Additionally, [7] presents an overview of Ripple with statistics about the platform's use and growth. Nevertheless, we found no previous work related to this technology on how to detect suspicious behavior. The goal of this case study is to explore that possibility.

Dataset analysis

We collected Ripple transactions limited to the successful ones. The information we use in this chapter belongs to the time interval between January and June of 2016.

We first measure the distribution of transaction types in Table 3.5. Although our dataset comes from a posterior period than the study presented in [7], we observe the same kind of results. There is a considerable amount of order cancellations, and at the same time, more than 82% of the submitted orders were replacing a previous order (an alternative cancellation mechanism). Furthermore, roughly 20% of the created offers get executed at the moment of submission. The movement of actual value between accounts only represents 30% of the transactions. To study users' trading behavior, we simplify the dataset considering only the payments and the executed or partially executed orders.

[1]https://xrpcharts.ripple.com/
[2]https://bithomp.com/

Table 3.5 Transaction type count (Ripple)

Type	Count
Order Submission	79831744
Order Cancellation	11017512
Payment	4735046
Others	3224138
Total	98808440

Table 3.6 Graph structure (Ripple)

	deg_{out}	deg_{in}
Mean	248.772	117.751
Std.	4213.015	1975.149
Min.	0	0
25%	0	1
Median	1	2
75%	3	4
Max.	330548	314426

Internal	Sources	Sinks
15901	6574	17969

Depending on each transaction type, we split the dataset into sequences per user sorted by date. For payments, we add the transaction to the sender's sequence and the receiver's sequence. If the payment affects offers in its path, we also add the order executions to the corresponding sequences. For order submissions, we add the transaction to the order creator's sequence and the sequences of the crossed orders' owners. Our resulting dataset consists of 14,560,124 samples corresponding to 40,754 different wallets. We represented every fund movement with the transaction ID, the source and destination IDs, the amount and the currency sent, and the amount and currency received. The reason for the existence of multiple destinations depends, as we mentioned before, on the type of transaction. For order submissions, there can be one or more matching orders, and in the case of the payments, there is always a unique sender, but it is possible to have multiple payment paths involving one or more intermediary accounts. Furthermore, among these transactions, we find 191 currencies. The distribution of transactions per currency is very long-tailed. Hence we present the ten most frequent currencies in Figure 3.5.

To understand the general relations of fund movements between accounts, we build a directed graph as in the previous case study. The graph structure is described in Table 3.6. In this case, we find more extreme right tails in the number of ingoing and outgoing transactions per user, but a more balanced distribution between internal, source, and sink nodes.

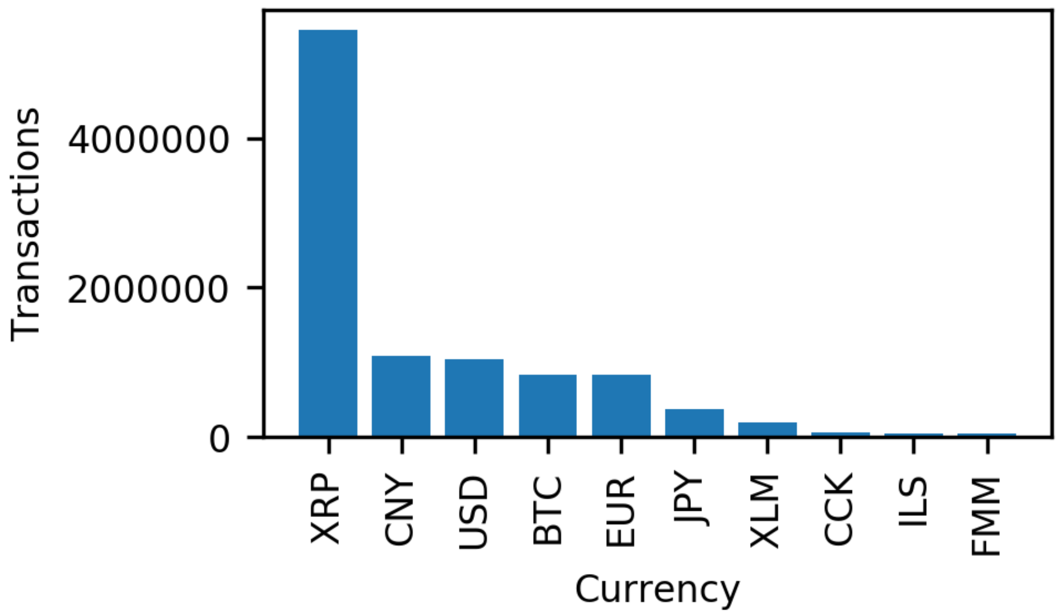


Fig. 3.5 Most used currencies in the Ripple dataset.

Feature engineering

In this case study, we have information for both sources and destinations of fund movements. Hence, we create a signature vector aggregating transactions for every account. We have five counting features for every user:

- the number of transactions involving the user,
- the number of transactions where the user is the sender,
- the number of transactions where the user is the receiver,
- the number of different accounts who sent funds to the user,
- and the number of different accounts who received funds from the user.

We collect all the incoming and outgoing fund movements separately for the seven most frequent currencies, obtaining 14 different time series. We also measure the number of seconds between every event generating an additional vector. We build two features for all these collections based on the median and the range. Finally, we create the same two features for the time series of delay between transactions involving the user. The resulting dataset contains 35 non-empty features, and each of them has a significant variance.

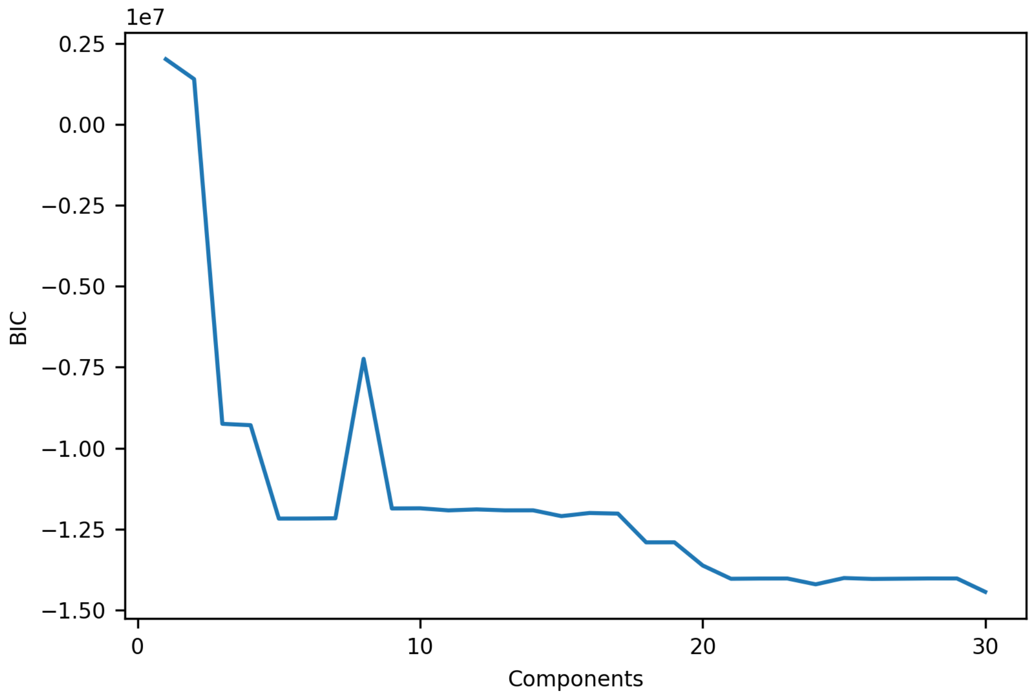


Fig. 3.6 Amount of GMM components vs. BIC score (Ripple).

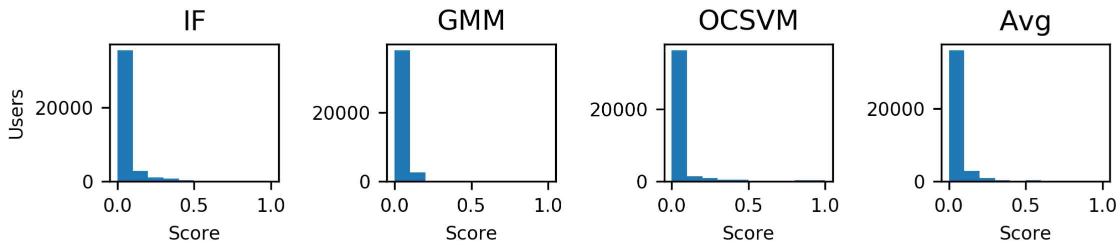


Fig. 3.7 Anomaly score distribution for different algorithms (Ripple).

Anomaly detection

We select the parameters again for the anomaly detection algorithms, as mentioned in Section 3.2. Casually we used also 20 components for the GMM in this case, based on the BIC curve that we observe in Figure 3.6. The histogram plots of scores for every algorithm in Figure 3.7 show this time more similar curves with very long tails to the right. Additionally, the 3d scatter plot in Figure 3.8 shows how correlated are the scores among the algorithms.

In this case study, we have more features related to continuous variables. We show the user score with the highest value of each feature in Table 3.7. We can see that the features

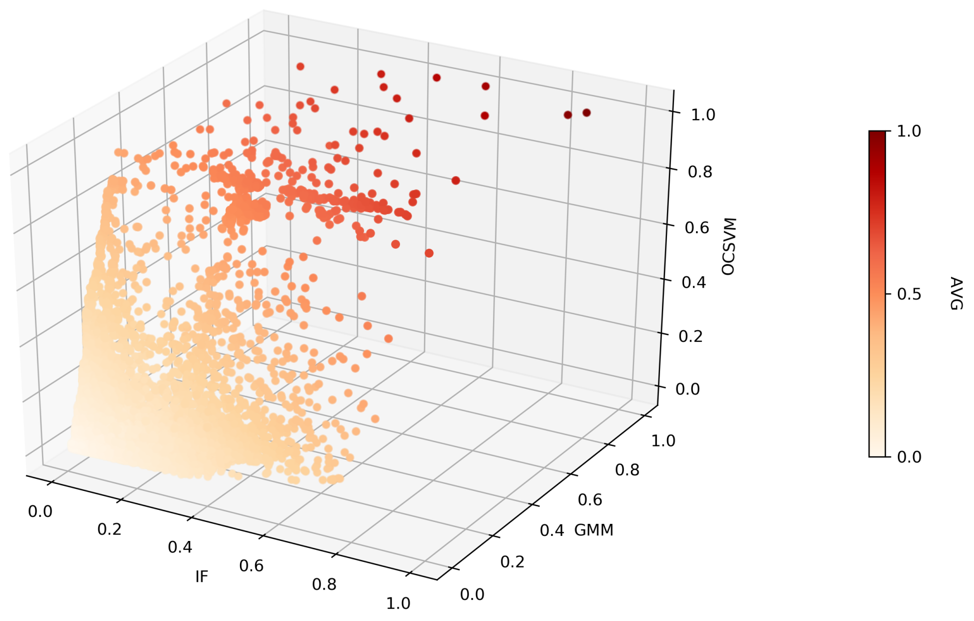


Fig. 3.8 Scores for the three algorithms (Ripple).

corresponding to time delays between transactions, the number of transactions sent, and total transactions have low anomaly scores.

Validation

In most cases, the Ripple accounts in the highest positions of the anomaly ranking had a considerable amount of transactions. It is tedious to analyze all that information manually, and the used anomaly detection algorithms do not offer a human-comprehensible interpretation for their decisions. Nevertheless, the Ripple community is very active, and its members can be considered experts in the field. We can find discussions about unusual transactions and suspicious accounts in the official issue tracker[3] and either the official[4] or the unofficial[5] forum. A collection of relevant accounts among the top suspects is shown in Table 3.8.

We found using Bithomp that some of them are related to known gateways like TokyoJPY, SnapSwap, Bitstamp, Mr. Ripple, and RippleFox. It seems reasonable that the algorithms consider gateways as anomalies because they behave differently from most users. We found similar cases in the forums[6] where users suspect that some accounts are related to banks

[3] https://github.com/ripple/rippled/issues
[4] https://forum.ripple.com/
[5] https://www.xrpchat.com/
[6] https://www.xrpchat.com/topic/1398-banks-on-ripple/

Table 3.7 Anomaly Score for the user with the maximum value of each feature (Ripple)

Feature	Score	Feature	Score
transactions	0.582	median($USD_{receive}$)	0.660
sent	0.582	range($USD_{receive}$)	0.752
received	0.696	median(BTC_{send})	0.639
senders	0.733	range(BTC_{send})	0.667
receivers	0.622	median($BTC_{receive}$)	0.867
		range($BTC_{receive}$)	0.820
median(delay)	0.395	median(EUR_{send})	0.646
range(delay)	0.332	range(EUR_{send})	0.667
median(XRP_{send})	0.753	median($EUR_{receive}$)	0.660
range(XRP_{send})	0.753	range($EUR_{receive}$)	0.692
median($XRP_{receive}$)	0.518	median(JPY_{send})	0.639
range($XRP_{receive}$)	0.623	range(JPY_{send})	0.609
median(CNY_{send})	0.762	median($JPY_{receive}$)	0.754
range(CNY_{send})	0.667	range($JPY_{receive}$)	0.708
median($CNY_{receive}$)	0.678	median(XLM_{send})	0.439
range($CNY_{receive}$)	0.713	range(XLM_{send})	0.667
median(USD_{send})	0.646	median($XLM_{receive}$)	0.445
range(USD_{send})	0.667	range($XLM_{receive}$)	0.622

Table 3.8 Ripple Accounts With High Anomaly Ranking

Rank	Account	Details
1	r94s8px6kSw1uZ1MV98dhSRTvc6VMPoPcN	gateway TokyoJPY
2	rMwjYedjc7qqtKYVLiAccJSmCwih4LnE2q	gateway SnapSwap
3	rQpCJ9uou6X7YXiHJiuyHNk72Ak1Pm9XWS	suspected bank
4	rogue5HnPRSszD9CWGSUz8UGHMVwSSKF6	arbitrage bot
5	rvYAfWj5gh67oV6fW32ZzP3Aw4Eubs59B	gateway Bitstamp
6	rB3gZey7VWHYRqJHLoHDEJXJ2pEPNieKiS	gateway Mr. Ripple
7	rghL9q8iPW6P4ZqG53nv3VNkVBKWWngdd	gateway RippleFox
10	rMqKD6oS578ekZkhsREXcYBQWwx9HHqWEy	suspected bank
15	rnCA4c4gn1ZL5ufyNNqKnjvVWQn3P7rxKc	suspected bank
17	rEiUs9rEiGHmpaprkYDNyXnJYg4ANxWLy9	arbitrage bot
20	rJnZ4YHCUsHvQu7R6mZohevKJDHFzVD6Zr	whale account
22	rfh3pFHkCXv3TgzsEJgyCzF1CduZHCLi9o	price manipulation
24	rQhMHRiXpkn1X1mWDRJxsu1VEpfS8VAaxK	dumping
51	rGcQo3R21J8BpextdCS1KNhweUJht6c8Pv	stealing via exploit

acting as gateways, but their identities were never confirmed. The Ripple community detected another interesting case[7], where a single account referred to as a *whale account* holds an abnormally high amount of resources.

At the top of the ranking, we found two accounts that appear in the forums[8] and the issue tracker [9] as examples of arbitrage bots. The number of trades, destinations, and currencies they use is considerably higher than usual, and they also present anomalous delays between transactions because they trade very often. Even if arbitrage is not illegal, it shares a typical pattern with the wash trade technique (see Chapter 1). Thus, we should carefully examine transactions and accounts related to this practice. In another forum, article[10] users suspect that an account may be a trading bot trying to manipulate prices somehow by putting sell orders very close to buy orders with abnormally small amounts. Another suspected manipulation case[11] is related to an account in our ranking that received a large amount of XRP directly from Ripple and sold all his reserves in a short period (*dumping*). If the funds' movement was between members of the same party, they could profit from the sudden price change.

Finally, we found an account accused of exploiting a Ripple vulnerability[12]. The attack presents a pattern similar to circular payments involving at least one victim, a *good* gateway, and another *bad* one that is either out of service or openly recognized as malicious. If the victim has previous trust lines with both gateways, the technique allows the attacker to exchange his IOUs from the bad gateway with the victim's IOUs from the good gateway.

3.4 Discussion

To obtain a different understanding of the anomalies, we tried using decision trees with anomaly/non-anomaly as a target variable for every account. We can generate these labels by experimenting with different anomaly thresholds for splitting the anomaly score in two. We employed the trained decision trees to extract rules that decide when an account is an anomaly or not [9]. Nevertheless, the rule sets we obtained were too large and complicated for human interpretation.

Additional graphical representations could also improve the validation of suspicious cases. We experimented with *t-SNE* [93], an algorithm for analyzing data reducing the dimensionality of the feature space. We created several 2D, and 3D scatter plots exploring

[7]https://www.xrpchat.com/topic/432-again-this-wallet-is-selling-like-crazy- ripplewallet18972352/
[8]https://github.com/ripple/rippled/issues/1257
[9]https://www.xrpchat.com/topic/2910-example-of-code-for-pseudo-circular-payments/
[10]https://www.xrpchat.com/topic/228-bot-pushing-price-down-sort-of-wierd/
[11]https://www.xrpchat.com/topic/1293- saxony-related-accounts-got-xrp-from-rippleinc-is-dumping/
[12]https://www.xrpchat.com/topic/3216-solved-a-bug-and-a-new-kind-of-attack-on-rcl/

different parameter combinations according to the guidelines of an online tool[13], but we could never obtain any meaningful insights on account relations.

Dimensionality reduction techniques can also be used as an alternative to feature selection. We could apply *Principal Component Analysis (PCA)* before running anomaly detection algorithms to deal with smaller and more dense feature spaces [9]. The problem is that when we keep the dimensions related to the principal components that can explain better most of the variation in the dataset, we are not only discarding noise but also the particularities of the anomalous cases. Additionally, the resulting transformed space's data points are even more challenging to interpret for humans than the original ones.

Finally, even if we have no certainty about money laundering or terrorist financing in the Ripple network, we applied the same methodology to both case studies and found suspicious cases with similar patterns. Although Ripple transaction information is public, the number of transactions for the most active users is more significant than in the private company case, and some of those users could be trying to obfuscate the source or the final use for their founds. Furthermore, we could find accounts exploiting vulnerabilities that we were not aware of beforehand with our approach.

3.5 Conclusion

We presented a methodology to analyze financial transactions from different sources, using unsupervised learning methods. We presented two case studies, one representing a commercial application we explored with our industrial partner and another from a publicly available data source (Ripple transaction data). In the first case, we were able to validate our findings with experts from the private company. In particular, our tool was able to spot exciting cases in addition to the suspicious samples that traditional rule-based systems can detect. In the second one, we validated our findings based on information from community forums. We detected expected anomalies (e.g., gateway accounts) and cases unknown to us beforehand, like arbitrage bots and vulnerability attacks. These findings confirm that anomaly detection on user behavior is a must in both traditional and modern payment systems.

In the future, we plan to focus on detecting unexpected changes in user behavior and extending the analysis methods to fit streaming environments.

[13]http://distill.pub/2016/misread-tsne

Chapter 4

Honeypot Detection in Ethereum

4.1 Motivation

Ethereum smart contracts have recently drawn a considerable amount of attention from the media, the financial industry, and academia. With the increase in popularity, malicious users found new opportunities to profit by deceiving newcomers. Consequently, attackers started luring other attackers into contracts that seem to have exploitable flaws but contain an elaborate hidden trap that benefits the contract creator. The blockchain community identifies these contracts as honeypots. A recent study presented a tool called HONEYBADGER that uses symbolic execution to detect honeypots by analyzing contract bytecode. In this chapter, we present a data science detection approach based foremost on contract transaction behavior. We create a partition of all the possible fund movement scenarios between the contract creator, the contract, the transaction sender, and other participants. To this end, we add transaction aggregated features, such as the number of transactions and the corresponding mean value and other contract features, for example, compilation information and source code length. We find that all categories mentioned above of features contain useful information for the detection of honeypots. Moreover, our approach allows us to detect new, previously undetected honeypots of already known techniques. We furthermore employ our method to test the detection of unknown honeypot techniques by sequentially removing one technique from the training set. We show that our method is capable of discovering the removed honeypot techniques. Additionally, we present two new techniques discovered with our method.

4.2 Data acquisition and feature extraction

4.2.1 Data

We limited the experiments in this chapter to the 2,019,434 smart contracts created in the first 6.5M blocks of the Ethereum blockchain because we need to match the range of blocks used in [38] to be able to use the labels from that study. Using the Etherscan API[1], we acquired compilation and source code information for 158,863 contracts downloaded around 141M external and 4.6M internal transactions.

4.2.2 Labels

We compile a list of contracts labeled as honeypots from HONEYBADGER's public code repository[2]. The repository provides a list of honeypots for each technique where source code is available on Etherscan. The authors developed a tool that generated the labels, and for each case, they indicated if the labeling was correct after a manual inspection of the contract source code. The authors only included one representative address in their list for groups of honeypot contracts that share the same byte code. To obtain a label for every contract, we calculate the *sha256* hash for the byte code of each contract and group them based on the hashes. We then transfer the label of each representative contract to every contract in its group. We only label as honeypot the contracts manually confirmed by the authors and ignore their tool's false positives.

4.2.3 Features

We create a set of features to understand the data better, verify that the behavior of transactions reflects the used honeypot technique, and automate the detection of honeypots using machine learning. We divide the features into three categories, depending on source code information, transaction properties, and fund flows. We describe the extraction process for each category's features and their selection reasons in the following sections.

Source Code Features

- *hasByteCode:bool* $\rightarrow$ indicates if the contract contains bytecode. The absence of bytecode means that the contract creation was erroneous and that it has no behavior.

[1]https://etherscan.io/apis

[2]https://github.com/christoftorres/HoneyBadger

- *hasSourceCode:bool* $\rightarrow$ we hypothesize that the number of source code lines of a honeypot is a decent approximation of the contract's readability or interpretability.
- *numSourceCodeLines:int* $\rightarrow$ counting the number of lines in the source code does not entirely reflect readability and interpretability characteristics, but it is a decent approximation that does not require static code analysis or symbolic execution.
- *compilerRuns:int* $\rightarrow$ we are interested in determining if a high amount of optimizations during compilation could lead to unexpected behavior that attackers can exploit.
- *library:int* $\rightarrow$ we collected all the possible libraries and created a categorical variable of 749 unique values. We are interested in knowing if the adoption of a particular library correlates with the occurrence of honeypots.

And finally, we split the *compilerVersion:str* into three parts: major, minor, and patch. We are interested in finding if a major, minor, or patch version significantly affected honeypots, caused by some vulnerability in the solidity language compiler. For each part, we collected all the possible values and calculated two categorical variables:

- *compilerMinorVersion:int* $\rightarrow$ 4 unique values.
- *compilerPatchVersion:int* $\rightarrow$ 330 unique values.

The compiler major version is 0 for all the examined contracts, so we discard the feature for not providing any information.

Transaction Features

We begin with the features we generate for both external and internal transactions.

- *[internal|external]TransactionCount:int* $\rightarrow$ the number of transactions of the corresponding type associated with the contract. We expect to filter out contracts with an abundance of movements, which are probably not honeypots.
- *[internal|external]TransactionOtherSenderRatio:float* $\rightarrow$ we calculate this by dividing the number of unique senders by the number of total senders, omitting the contract creator in both. The feature should be one if every transaction is sent by a different sender, close to zero if few senders send many transactions, and we define it as zero if the creator is the only sender. We would like to know if the participation of multiple senders is a characteristic of honeypots or not.

Next, we compute the average and the standard deviation for each contract and each property of transactions involving fund movements. This yields a total of six features per transaction type, external and internal:

- *[internal|external]TransactionValue[Mean|Std]:float*
- *[internal|external]TransactionGas[Mean|Std]:float*
- *[internal|external]TransactionGasUsed[Mean|Std]:float*

The value is only collected if there are no errors in the transaction. If there is an error, the system rolls back the transaction and any transferred funds, but it still spends the gas. We expect the feature means to be positive but small for honeypots, more varied for popular non-honeypot contracts, and zero for contracts with testing purposes. Furthermore, for the feature standard deviations, we have some weak intuitions we would like to test: high standard deviations for gas and gas used may be correlated with contracts with many different execution paths, becoming a proxy indicator for the complexity of the contract code. Additionally, the standard deviation for the value may point out how different or how similar are the transactions of a contract in terms of the amount of funds.

Moreover, we extract features related to time and block numbers for external transactions. Internal transactions do not carry their timestamps or block numbers, and we cannot assume that Etherscan sorts them chronologically. Our intuition is that honeypots have short lives compared to other contracts, which was also pointed out in [38]:

- *externalTransactionBlockSpan:int* $\rightarrow$ derived from the block number of the last transaction minus the first transaction's block number.
- *externalTransactionTimeSpan:int* $\rightarrow$ computed from the timestamp of the last transaction minus the timestamp of the first transaction.

We also compute the block and time differences between transactions. Both the mean and standard deviation indicate how often the contract receives transactions (only for contracts with more than one transaction):

- *externalTransactionBlockDelta[Mean|Std]:float*
- *externalTransactionTimeDelta[Mean|Std]:float*

Lastly, features extracted only for internal transactions:

Table 4.1 Fund Flow Variables.

Name	Possible Values
sender	creator, other
creation	yes, no
error	yes, no
balanceCreator	up, unchanged, down
balanceContract	up, unchanged, down
balanceSender	up, unchanged, down
balanceOtherPositive	yes, no
balanceOtherNegative	yes, no

- *internalTransactionCreationCount:int* $\rightarrow$ indicates how many contracts this contract created. We hypothesize that honeypots are less likely to create other contracts.
- *internalTransactionToOtherRatio:float* $\rightarrow$ this is analogous to the feature *transactionOtherSenderRatio*, but it is calculated for the receivers instead of for the senders.

Fund Flow Features

According to the definition of honeypots, we expect to find the following events at least once:

1. A contract is created successfully.
2. The contract's creator sends funds to the contract. We can interpret this event as "setting the trap" because funds attract attackers to exploit the contract.
3. A user that is not the contract's creator sends funds, and the contract keeps them. We can interpret this event as a victim falling for the trap.
4. The contract creator withdraws its profit from the contract.

Most importantly, a honeypot should rarely, if ever, present a withdrawal from an account other than its creator.

We define different events triggered by external transactions (and potentially followed by internal transactions), where each possible event corresponds to precisely one case. We describe these cases with a combination of eight variables in Table 4.1.

For each external transaction, we collect the resulting internal transactions. We determine the *sender* variable by comparing the transaction source to the contract creator. The *creation* variable is set to *yes* only for the creation transaction. However, the *error* variable value, in this case, is an aggregation: it is set as *True* if the external transaction or any of the triggered internal transactions contain an error. For the balance changes, we use the following algorithm: we initialize every account with zero funds moved, and for each transaction (both external and internal), we subtract the value from the *transaction.from* account and add it to the *transaction.to* account (or to the *transaction.contractAddress* account if a contract is created). After iterating over the transactions, we consider the funds moved into the *creator*, *contract*, and *sender* accounts. We mark them as *up*, *unchanged*, or *down* if the values are positive, zero, or negative, respectively. If the balance of any account other than the three mentioned above is changed, we have two special flags that indicate if at least one other account balance went up or down respectively. We prefer to keep a small and fixed number of variables, and computing all the possible combinations yields $2^5 \times 3^3 = 864$ cases, many of them invalid. For example, *creation=yes* only makes if when *sender=creator*, and we only care about *balanceSender* if *sender=other*. Also, at least one balance goes up if and only if at least one balance goes down. After discarding invalid combinations, we retain 244 valid cases. Finally, we extract the frequency of each case for every contract by counting how many times a case appears divided by the total amount of cases.

4.3 Exploratory Analysis

In the following section, we analyze each of our variables in search of distribution differences between honeypots and non-honeypots. We divide the study not by the categories presented in the previous section but by the variable type (numerical and categorical) because the same mechanisms can describe variables with the same type.

4.3.1 Most Relevant Numerical Features

We take all the numerical features and study how their distributions differ when dividing them by honeypots and non-honeypots. We include only the results for three features where the differences are more evident: the number of code lines, the number of external transactions, and the transaction value. We describe the distribution of the three variables in Table 4.2.

First, we observe that honeypots seem to have a reasonable amount of code lines in general, with a minimum that could include a fair amount of logic and a still readable maximum. On the other side, non-honeypots present extreme values, from contracts with only one line of code to a contract with more than 11K lines. Second, we see that three-quarters of the contracts have a small number of external transactions. However, the maximum amount is much larger for non-honeypots than for honeypots. Lastly, it seems that all honeypots always move some amount of ether, but most non-honeypot contracts (over 75%) do not exchange

Table 4.2 Distribution description for the most relevant numerical features.

Feature	*numSourceCodeLines*		*normalTransactionCount*		*normalTransactionValueMean*	
Honeypot	Yes	No	Yes	No	Yes	No
mean	57.00	279.70	5.13	680.61	0.27	5.46
std	22.80	278.42	4.26	33198.52	0.22	570.30
min	19.00	1.00	1.00	1.00	0.00	0.00
25%	41.00	109.00	3.00	1.00	0.10	0.00
50%	54.00	177.00	4.00	2.00	0.25	0.00
75%	67.00	386.00	6.00	4.00	0.38	0.00
max	185.00	11409.00	32.00	10412943.00	1.67	204365.82

value at all. Nonetheless, non-honeypots have very extreme maximum value movements, while honeypots move on average no more than 2 ether.

We could use handcrafted rules to cut off the extremes of all these features. However, one of those rules' problems is that they overfit the dataset used to craft them. Even if the boundaries are relaxed, if a new honeypot crosses one of those boundaries (both accidentally or on purpose), it would be automatically classified as a non-honeypot disregarding the rest of its properties. In any case, the rules we could craft can capture all the honeypots but do not exclude a sufficiently large portion of non-honeypots. We could start combining several rules until we find the best honeypot filter, but machine learning can do that for us.

4.3.2 Categorical Variables

All the extracted categorical features belong to the compilation and source code information category: the compiler mayor, minor and patch versions, and the library. The minor compiler version has only four possible values. Most contracts belong to the minor version 0.4, and in particular, all but one honeypot among our samples belongs to that version. These statistics agree with a popularity peak in Ethereum described by the authors of HONEYBADGER [38]. The compiler patch version has 330 possible values. One compiler version (0.4.19+commit.c4cbbb05) is more common than others for honeypots, but no specific honeypot technique is correlated. We repeat the procedure of counting honeypot techniques for the rest of the compiler versions obtaining the same results. This finding suggests that we cannot link honeypot activity to any compiler version. Instead, it seems that the high frequency for some versions is related to the time when they were active, which could be at the same time linked with Ethereum's popularity. Finally, none of the honeypots have a library. Hence, we can presume that libraries do not correlate with honeypot development or that no library contains vulnerabilities exploitable for honeypot related purposes.

4.3.3 Fund Flow Cases

We study the events related to fund movements to verify if honeypots behave as we expect. For this, we query our data by partially defining some of the fund flow variables and adding the frequencies of all matching cases.

We assumed that honeypots should receive a deposit from its creator to work since funds are needed to attract attackers. However, we find that 29 honeypots (around 10% of them) do not match this criterion. Inspecting individual cases, we find that these contracts are honeypots but rely on initial deposits from victims to attract even more victims. Another explanation could be that the creators of honeypots sometimes use a different account to set their trap. We do not find clear evidence of such practice among our data, but future work could address this topic, for example, using graph techniques to find colluding accounts.

We find 192 honeypots that do not have any deposits from non-creator accounts, which means that only 103 (roughly 35% of them) were able to lure victims into sending funds. This metric is close to the statistics presented in [38]. Moreover, only 108 honeypot contracts ($\sim$37% of them) present withdraws from the creator. It makes sense to expect the creator to withdraw funds only if the trap was effective. However, the amount of honeypots with creator withdraws is higher than the number of effective ones, which indicates that the creator sometimes withdraws the bait when the trap seems not to be working.

Honeypots should never present withdraws from non-creator accounts, but 50 cases ($\sim$17%) matching this criterion. We find some interesting cases among this group where the honeypot creator failed to set the trap correctly. For example, one contract[3] that belongs to the Hidden State Update category. This kind of honeypot needs a transaction to change the value of one or more variables in the contract. This update mechanism is usually hard to track because Etherscan does not publish internal transactions that carry no value, and victims fall into a trap when they do not know the contract's state. However, for our contract example, the hidden state change transaction is missing, and the users that were supposed to be victims made a profit.

In Table 4.3 we present the top most frequent balance movement cases. The count represents the number of external transactions that match the fund flow case. The IDs are arbitrary values used to identify the features later on. We do not include the Boolean variables with False values and the balance variables without changes for readability purposes. We can observe some expected cases for honeypot contracts: creator deposits and withdrawals, deposits from non-creator accounts, and contract creation (without any fund movement). This category's unexpected top case represents transactions from non-creator accounts that

[3]address=0xbC272B58E7cD0a6002c95aFD1F208898D756C580

Table 4.3 The five most frequent fund flow cases by honeypot and non-honeypot.

Honeypots		
Count	ID	Description
341	205	*sender=other*
304	83	*sender=creator, balanceCreator=negative, balanceContract=positive*
292	33	*sender=creator, creation=True*
168	201	*sender=other, balanceContract=positive, balanceSender=negative*
136	73	*sender=creator, balanceCreator=positive, balanceContract=negative*
Non-Honeypots		
Count	ID	Description
94475857	205	*sender=other*
9667149	207	*sender=other, balanceSender=negative, balanceOtherPositive=True*
9130582	201	*sender=other, balanceContract=positive, balanceSender=negative*
8569634	77	*sender=creator*
5647532	127	*sender=other, error=True*

do not generate any fund movement. Furthermore, we can see that deposits from non-creator accounts are also prevalent for non-honeypot contracts.

4.4 Experiments

4.4.1 Additional Pre-Processing

Before starting any of the machine learning experiments, we need to take further data cleaning steps. Missing values in our dataset are mostly a product of our feature extraction. We cannot compute the value aggregated features for any transaction type when there is an error because the system rolls back the funds. For external transactions, we cannot measure the difference of time or block number between consecutive external transactions when the contract only has one external transaction. The rest of the external transaction features are always defined because there is always at least one external transaction (the creation, even if it contains errors). Nevertheless, for all these situations, we found it reasonable to fill the missing values with zeros. However, a contract may not have any internal transactions. Consequently, all the related aggregated features are undefined. Even if we could fill those features with zeros, only 18% of the contracts have internal transactions. Hence, we discard the aggregated features related to internal transactions for the following experiments, but we add a new Boolean feature that indicates the presence of internal transactions.

Furthermore, not every contract is useful for our classification task. We discard contracts without bytecode because the system cannot execute them. We also discard contracts without source code for different reasons:

- It is harder to lure victims looking for vulnerabilities in contracts if the source code is not publicly available.
- We only have labels for contracts with source code.
- We cannot compute features from the compilation and source code category for contracts without source code.

After the filtering step, we reduced the number of samples in our dataset to 158,568 non-honeypots and 295 honeypots.

Also, we find that not all fund flow cases are useful. Therefore, we discard all those that do not have any frequency across our reduced dataset. Thus, we end up with 74 out of 244 cases that are useful for describing the transaction behavior. We also check each feature's variance, but none present a variance very close to zero. Thus we do not discard any other dataset columns.

Our last step is taking any “unbounded" numerical feature (the ones that are neither frequencies nor ratios) and apply min-max scaling to shrink them into the range $[0,1]$. We use this technique to have every feature in the same range, allowing us to understand feature importance better later on.

4.4.2 Honeypot Detection

In this section, we train a machine learning model called XGBoost [25] to classify contracts into honeypots (the positive class) or non-honeypots (the negative class).

We use k-fold cross-validation to ensure that the models generalize to unseen data. We split the data into $k = 10$ separate subsets called folds, where one is selected for testing, while we train the algorithm on all the remaining folds. We repeat the procedure k times so that each fold is used once for testing. In particular, we use stratified k-fold cross-validation, in which the folds preserve the proportion of classes.

Furthermore, we deal with an imbalanced classification problem: our dataset contains many more non-honeypot contracts than honeypot contracts. This issue harms the classification capabilities of machine learning algorithms. To address this issue, we configure XGBoost to train with a scaling weight for the positive class[4].

[4]https://xgboost.readthedocs.io/en/latest/tutorials/param_tuning.html#handleimbalanceddataset

Table 4.4 Honeypot classification experiments for different categories of features.

Features	Train	Test
All	0.985 ± 0.002	0.968 ± 0.015
Only Transactions	0.966 ± 0.004	0.954 ± 0.030
Only Source Code	0.953 ± 0.002	0.942 ± 0.025
Only Fund Flow	0.952 ± 0.002	0.938 ± 0.023

We use the AUROC (Area Under the Receiver Operating Characteristics) to quantify our model's power on each fold. The ROC curve describes the TPR (True Positive Rate) against the FPR (False Positive Rate) for different thresholds over the classification probability. The area under that curve summarizes how well the model can distinguish between the two classes over different threshold configurations.

In Table 4.4 we compare train and test AUROC for four sets of features: the features based only on source code information, based only on aggregated transaction data, based only on fund flow case frequencies, or using all the features together. In general, for every experiment, the difference between train and test is not large, indicating that all the models can generalize well to unseen samples. In particular, each individual set of features presents promising results, with a slight advantage when using all the feature sets together. To understand better the difference between the four alternatives, we list the three most essential features per feature set in Table 4.5. First, the source code group's most important feature is the number of source code lines, and the following most important is related to the compiler version. The problem with relying on the compiler version is that even if some values correlate with honeypots in the present dataset, those relations might become irrelevant for new compiler versions. The compiler versions we cover in this study can still be used in the future but might become less popular over time. Second, the frequency of deposits to a contract from its creator (*fundFlowCase83*) is a very distinctive fund flow feature, and the mean value of external transactions is the most relevant of the transaction aggregated features. The problem of both categories is that they rely only on transaction information. Hence, when a contract did not execute any transactions, there is no information to make a classification. Lastly, the advantage of using all the feature groups is that while the most critical features from each group are at the top of the joint importance ranking, they also cover each set's weaknesses.

Notice also that the metrics in Table 4.4 suggest that detection is possible using only source code features. Hence, the transaction information might not be necessary to detect honeypots. We test this assumption by experimenting with contracts that have no transactions using only source code features. We obtain a train AUROC of 0.986 ± 0.002 and test AUROC of 0.885 ± 0.201. The cross-validation shows a high difference between train and test metrics

Table 4.5 The three most important features per feature category.

Category	Feature	Importance
All	fundFlowCase83	0.657
All	normalTransactionValueMean	0.107
All	numSourceCodeLines	0.071
Only Transactions	normalTransactionValueMean	0.576
Only Transactions	normalTransactionValueStd	0.117
Only Transactions	normalTransactionGasUsedStd	0.077
Only Source Code	numSourceCodeLines	0.424
Only Source Code	compilerPatchVersion136	0.129
Only Source Code	compilerPatchVersion125	0.070
Only Fund Flow	fundFlowCase83	0.799
Only Fund Flow	fundFlowCase201	0.036
Only Fund Flow	fundFlowCase79	0.032

(a case of overfitting). The source of this problem could be the low number of positive instances on this experiment (only ten honeypots versus 64,333 non-honeypots), but this result could also point out that the transaction information is necessary.

4.4.3 Simulated Detection of Unknown Honeypot Techniques

The previous experiment measures how well the machine learning model generalizes the classification to contracts unseen during training, but those contracts share honeypot techniques with other contracts present in the training set. In the following experiment, we intend to simulate what would happen if we need to classify a contract that belongs to a honeypot technique that we never encountered before. We change the cross-validation strategy by picking one honeypot technique at a time: we build the testing set only with the samples belonging to it and the training set with the remaining samples. This results in the eight different experiments that are shown in Table 4.6. Because the test set comprises positive cases, we measure the test $Recall = TP/(TP+FN)$.

The high recall for all cases implies that the machine learning approach can detect honeypots excluded during training. This property is a clear advantage over HONEYBADGER. Because it relies on expert knowledge of the different honeypot techniques to craft detection rules, it cannot detect new yet unknown techniques.

Table 4.6 Simulating the detection of unknown honeypot techniques.

Removed Honeypot Technique	FN	TP	Recall
Type Deduction Overflow	0	4	1.000
Uninitialised Struct	2	37	0.949
Hidden Transfer	1	12	0.923
Hidden State Update	12	123	0.911
Inheritance Disorder	4	39	0.907
Straw Man Contract	3	28	0.903
Skip Empty String Literal	1	9	0.900
Balance Disorder	3	17	0.850

Table 4.7 Number of new honeypots discovered by technique.

BD	ID	TDO	US	HSU	HT	SMC	UC	MKET
1	13	1	6	16	8	5	6	1

4.4.4 Detection of New Honeypots and Honeypot Techniques

The labels we employ correspond to contracts marked as honeypots by HONEYBADGER and manually validated by its authors. Nevertheless, it does not mean that the rest of the contracts are necessarily non-honeypots. A considerable amount of human work is required to inspect millions of instances of Solidity source code. In this section, we describe a mechanism to review the unlabeled contracts in a time-efficient manner. Firstly we train ten different XGBoost models using ten stratified folds, and for each contract, we calculate the probability of being a honeypot with each model. Secondly, we compute the mean and standard deviation of the ten values for each contract. Lastly, we rank the contracts in descending order using the mean probability of being a honeypot. We can filter out the contracts for which the models did not agree on the prediction based on the standard deviation. The resulting ranking serves as an order for the experts to inspect the contracts. We decided to validate the first 100 samples of the list manually and discovered 57 new instances of honeypots[5]. From those instances, we found three extra copies by matching hashes of contract byte code. Table 4.7 describes the number of honeypot instances by technique. The table introduces two techniques that were not defined and were not detected by HONEYBADGER: UC and MKET.

The first of the newly found honeypots uses a *call* to transfer funds to a potential attacker, provided they sent funds first. However, the *call* is not complete: it is prepared but never executed, as the parenthesis required for the execution are missing. HONEYBADGER cannot detect this honeypot, as most compiler versions remove the unnecessary bytecode that sets up the *call*. Furthermore, since the system never executes the *call*, there is also no *CALL* opcode

[5]New honeypots: https://github.com/rcamino/honeypot-detection/blob/master/paper_new_honeypots.csv

within the bytecode. With the approach proposed in this chapter, we can detect this honeypot technique as more information is analyzed (e.g., funds sent to non-creator accounts). We propose to name this honeypot technique *Unexecuted Call (UC)*.

The second honeypot[6] that has been found with our approach is detailed in the following. It implements an intricate mechanism to define an owner of the contract's funds and can transfer them only to that account. Additionally, anyone can pay a fee and call a function that could seemingly overwrite the owner. Hence, the honeypot suggests that anyone can transfer all contained funds to themselves. The contract appears to offer a game of speed where users compete to call the function to become the owner, followed by another call to withdraw the funds before anybody else does the same. However, the strange mechanism contains a trap that does not allow changing the owner of the funds. The contract makes use of a hashmap to store the owner variable. This mapping uses a string, "Stephen," as the key to access the owner. The contract employs the correct key when writing the owner during deployment and whenever reading the owner variable. However, to overwrite the owner, a slightly but not noticeably different key is employed. A Cyrillic letter "e"[7] is introduced as the second "e" in "Stephen". It is so similar to the standard "e"[8] that a user cannot easily recognize it. Nevertheless, the code makes a distinction. Therefore, the misleading function cannot update the owner. Instead, it stores the new value under an altogether different key. Moreover, only the real owner can retrieve the funds. We propose to name this honeypot technique *Map Key Encoding Trick (MKET)*.

4.5 Future Work

There are several ways in which we could extend this chapter. We could explore the possibility of extracting features from the contract names based on the work presented in [111], by crossing the occurrence of words in the contract names with a relevant term list for Ethereum. Another alternative could be to calculate a global state of account balances after executing all the transactions of a contract instead of computing them per transaction.

We conceived the fund flow cases with the initial goal of analyzing the temporal dependencies between each other. In other words, we wanted to study the behavior of contracts using sequences of discrete symbols, which is equivalent to work with character strings or lists of words. However, as we found that most of the honeypots present very short sequences, the approach was not successful in finding useful patterns. In the end, we only used in this

[6]For example: 0xf5615138a7f2605e382375fa33ab368661e017ff.
[7]Unicode code 1077: https://www.codetable.net/decimal/1077
[8]Unicode code 101: https://www.codetable.net/decimal/101

chapter uni-gram frequencies and discarded other alternatives like Markov Chains, n-gram frequencies, or Recurrent Neural Networks. Nevertheless, we plan to employ fund flow case sequences in future studies to classify contracts (or even accounts) with more extensive transaction history into categories that do not involve honeypots.

We mentioned in Section4.4.2 that this study's method is weaker when a contract did not execute any transaction. HONEYBADGER does not have this problem, given that it works only with the contract byte code. One solution we consider for the future is adding more features that do not require any transactions, for example, extracting information by parsing the source code. Nevertheless, most importantly, we think that the symbolic analysis and machine learning approaches could complement each other. We should implement them simultaneously to cover mutual weaknesses.

4.6 Conclusion

In this chapter, we presented a step-by-step methodology to obtain, process, and analyze Ethereum contracts for the task of honeypot detection. We showed how we could contrast assumptions and hypotheses about honeypot behavior with real data and derived features for classification models[9]. The machine learning models proved to generalize well, even when we removed all the contracts belonging to one honeypot technique from training. Most importantly, we showed that our technique detected honeypots from two new techniques, which would be impossible to achieve using byte code analysis without manually crafting new detection rules. Furthermore, we proved that HONEYBADGER's rules covering known techniques were not capturing all the existing honeypot instances.

[9]Source code available at https://github.com/rcamino/honeypot-detection

Chapter 5

Multi-Output DGM

5.1 Motivation

In contrast to the continuous domain, where deep generative methods have delivered many results, they struggle to perform equally on tabular data. We propose several architecture extensions to train deep generative models on multivariate feature vectors representing multiple variables of different types. We evaluate our architectures' performance on datasets with different sparsity, number of features, ranges of categorical values, and dependencies among the features. We compute existing metrics, suggest alternative metrics, and show that our proposed architecture outperforms existing models.

5.2 Definitions

In this section, we provide a summary of concepts relevant to the topic of tabular data generation using deep learning models. We describe several architectures built progressively on top of each other until we reach the issue of generating discrete distributions, followed by existing proposals to solve the problem.

5.2.1 Tabular Data

From a statistical point of view, we can represent the outcome of a random phenomenon with a *random variable*. The literature defines a random variable with finite or countably infinite possible values as *discrete*. Note that this kind of variable may contain values that are not numerical, e.g., the nationality of a person participating in a poll could be French, German, or Spanish. On the other side, a random variable of the *continuous* type can take infinite real values. A probability distribution describes how likely a random variable or set of random variables takes each possible value. The variable type determines how to describe probability distributions. For example, the *Bernoulli* distribution describes the probabilities of a *random binary variable*, a discrete variable with only two possible values. The *Multinoulli* or *categorical* distribution covers a more general case, where discrete variables have a finite number of possible values greater than two. A sample is said to be *univariate* when it is composed of only one variable, and *multivariate* when it is composed of more than one variable. For this work, we are interested in *tabular data*, the general case when multivariate samples contain variables of heterogeneous types. This kind of data offers the possibility to represent and study complex variable interactions. Please refer to [49, Chapter 3 - Probability and Information Theory] for more information.

5.2.2 Encoding Variables as Numerical Features

In machine learning, computers improve from experience in some tasks without being explicitly programmed. In many cases (e.g., most of the supervised and unsupervised learning tasks), a *dataset* describes the experience, which is a collection of *samples*, which are in turn a collection of measurable properties called *features*. In practice, a dataset is usually represented with a matrix $X \in \mathbb{R}^{n \times m}$, where each row vector x_i corresponds to the sample number $1 \leq i \leq n$, and each column $x_{:,j}$ corresponds to the feature number $1 \leq j \leq m$. A numerical representation is needed because many machine learning algorithms (in particular deep learning models) can only work with this kind of data [49, Chapter 5 - Machine Learning Basics].

Tabular data, on the other hand, is usually represented with a structure $\mathbf{X}$ containing $\mathbf{n}$ rows and $\mathbf{m}$ columns. Note that we use a bold font to distinguish the tabular notation from the feature matrix notation, and we will describe variable indexing with superscripts between square brackets. We then define the rows of $\mathbf{X}$ as $\mathbf{x}^{[i]}$ for $1 \leq i \leq \mathbf{n}$ that correspond to multivariate samples, and the columns as $\mathbf{x}^{[:,k]}$ for $1 \leq k \leq \mathbf{m}$ that correspond to variables. Additionally, each variable k has a related variable type $\texttt{type}(k) \in \{\texttt{num}, \texttt{bin}, \texttt{cat}\}$ corresponding to numerical, binary, and categorical respectively. To work with tabular data in deep learning, we need to numerically encode all the variables, defining a mapping between each variable value and the feature space. Given that the number of rows does not change after the mapping, we define $n = \mathbf{n}$. For the same reason, we use i to refer to row numbers in both cases. However, the number of variables $\mathbf{m}$ and the number of features m might differ. We define $m^{[k]} \geq 1$ as the number of features the variable k is mapped into. Thus, the total amount of features of X is restricted to $m = \sum_{k=1}^{\mathbf{m}} m^{[k]}$. For each sample i and variable k, the definition of $m^{[k]}$ and the mapping $\texttt{map}(\mathbf{x}^{[i,k]})$ depends on $\texttt{type}(k)$.

The simplest case is when $\texttt{type}(k) = \texttt{num}$: we represent the value of the numerical variable with one feature with the same value, meaning that $m^{[k]} = 1$ and $\texttt{map}(\mathbf{x}^{[i,k]}) = \mathbf{x}^{[i,k]}$.

If $\texttt{type}(k) = \texttt{cat}$ then the variable k has a set of possible values $\texttt{values}(k)$. The set needs to be finite to define its size as $\texttt{size}(k)$. Otherwise, we cannot apply any of the following methods. Given that the set is countable, elements from $\texttt{values}(k)$ can be arbitrarily enumerated, by assigning to each one a different integer $1 \leq q \leq \texttt{size}(k)$. We then define $value(k,q)$ as the value of variable k that has the integer q assigned to it. One possible mapping is the *label encoding*, which uses the arbitrary enumeration to represent the variable value, hence $m^{[k]} = 1$ and $\texttt{map}(\mathbf{x}^{[i,k]}) = q$ when $\mathbf{x}^{[i,k]} = value(k,q)$. The problem with this technique is that there is no distance between the categorical values, but a distance emerges as a side effect of the representation choice, which could encourage machine learning models to find artificial patterns. This is not an issue when using *one-hot-encoding*, which maps the variable value into a binary vector with one position per possible value, where only the position related to the selected value is one, thus $m^{[k]} = \texttt{size}(k)$ and:

$$\texttt{map}(\mathbf{x}^{[i,k]}) = \left[x^{[i,k,1]}, \ldots, x^{[i,k,size(k)]}\right] \tag{5.1}$$

$$x^{[i,k,q]} = \begin{cases} 1 & \text{if } \mathbf{x}^{[i,k]} = value(k,q) \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

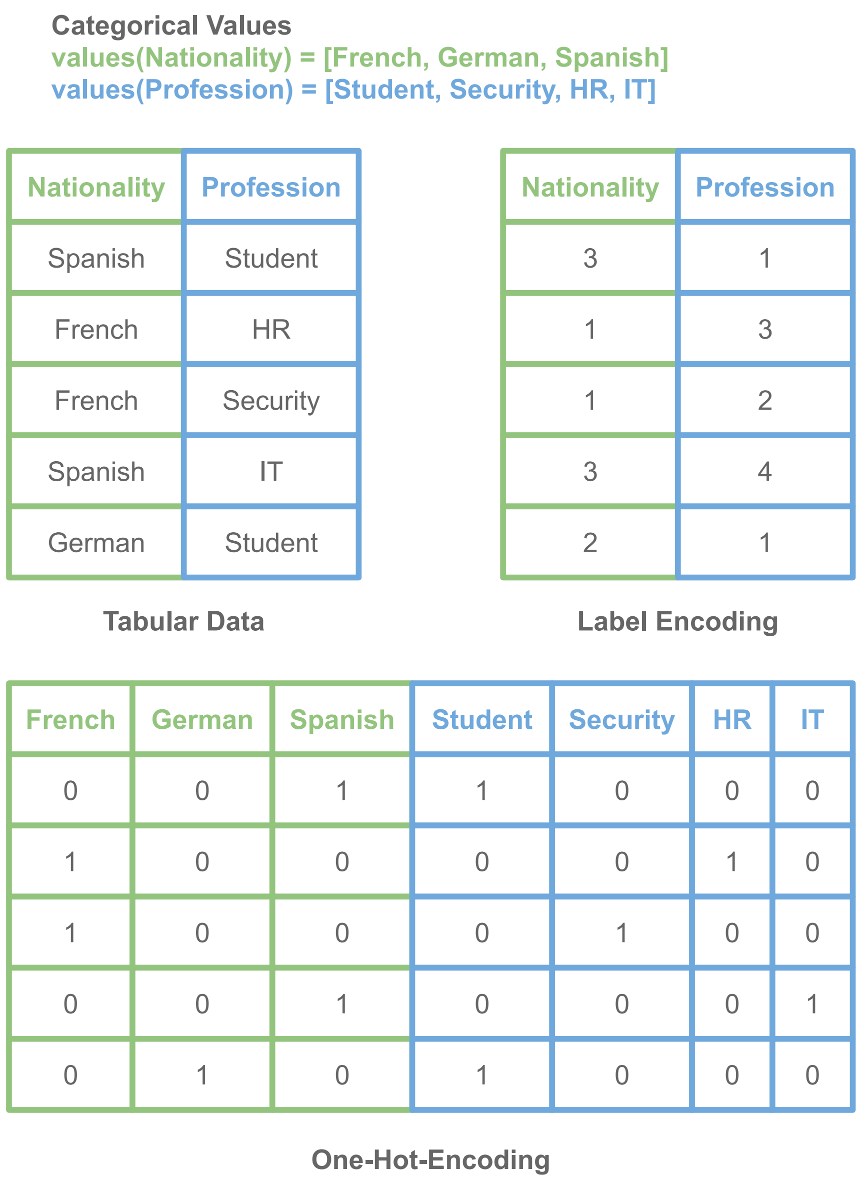
Finally, we define the mapping of an entire tabular row $\mathbf{x}_i$ into the feature vector row x_i as the vector concatenation over the mappings of each variable k (that we define as $x_i^{[k]}$):

$$x_i = \left[\texttt{map}(\mathbf{x}^{[i,1]}); \ldots; \texttt{map}(\mathbf{x}^{[i,\mathbf{m}]})\right] = \left[x_i^{[1]}; \ldots; x_i^{[\mathbf{m}]}\right] \tag{5.3}$$

In Figure 5.1 we show an encoding example with $\mathbf{m} = 2$ variables: nationality and profession. Both are categorical ($\texttt{type}(1) = \texttt{type}(2) = \texttt{cat}$). The values of the nationality are $\texttt{values}(1) = [\text{French}, \text{German}, \text{Spanish}]$, and the values of the profession are $\texttt{values}(2) = [\text{Student}, \text{Security}, \text{HR}, \text{IT}]$. The variables' order in the list defines the number assignation for both variables. We present the tabular data with $\mathbf{n} = 5$ rows, mapped into a matrix of $n = 5$ rows and $m = m^{[1]} + m^{[2]} = 1 + 1 = 2$ features with label encoding, and $m = m^{[1]} + m^{[2]} = 3 + 4 = 7$ features with one-hot-encoding.

5.2.3 The Problem of Backpropagation with Sampling

In deep learning, the task a model trains to solve is tightly related to the loss function, the output layer's activation, and the output representation. On one side, when the target variable is continuous, the related task is known as a regression. The mean squared error is usually



Categorical Values
values(Nationality) = [French, German, Spanish]
values(Profession) = [Student, Security, HR, IT]

Nationality	Profession
Spanish	Student
French	HR
French	Security
Spanish	IT
German	Student

Tabular Data

Nationality	Profession
3	1
1	3
1	2
3	4
2	1

Label Encoding

French	German	Spanish	Student	Security	HR	IT
0	0	1	1	0	0	0
1	0	0	0	0	1	0
1	0	0	0	1	0	0
0	0	1	0	0	0	1
0	1	0	1	0	0	0

One-Hot-Encoding

Fig. 5.1 Encoding Example.

the loss, which in this case, minimizing it is equivalent to maximizing the log-likelihood. The output layer does not need any non-linearity, which means the activation is linear. On the other side, for classification tasks, the target variable can be binary on two-class classification problems or categorical on multi-class classification problems. For binary classifications, the target variable belongs to a Bernoulli distribution conditioned on the features to compute the maximum-likelihood. The model then only needs to output a single number but must lie on the interval $[0, 1]$ to be a valid probability. For this purpose the *sigmoid* activation is used:

$$\sigma(h) = \frac{1}{1+exp(-h)} \tag{5.4}$$

This function provides the desired output range and, at the same time, is differentiable and continuous. This property facilitates the backpropagation of the errors, which we compute with the binary cross-entropy loss. The multi-class classification case is a generalization of the binary case. The *softmax* activation defines a Multinoulli distribution, by normalizing the input vector so each position falls in the range $[0,1]$ and they all add up to one:

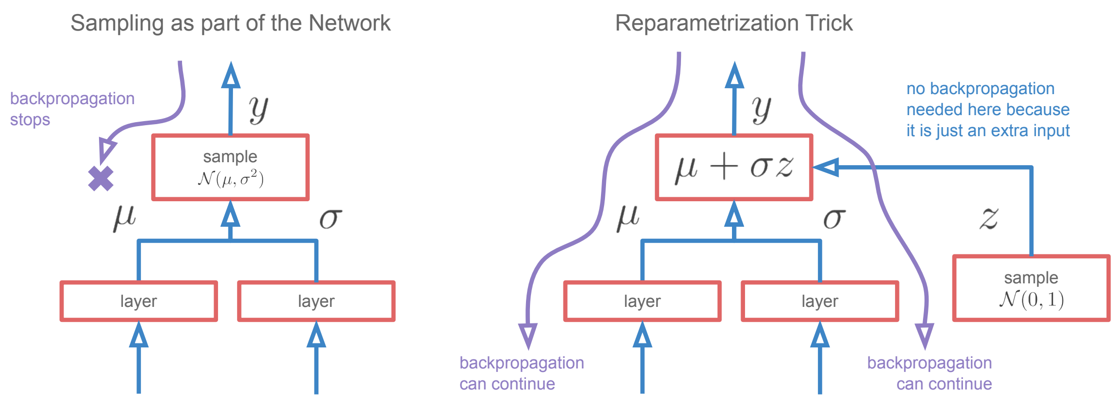
$$softmax([h_1,\ldots,h_m]) = \left[\frac{exp(h_1)}{\sum_{j=1}^{m} exp(h_j)},\ldots,\frac{exp(h_m)}{\sum_{j=1}^{m} exp(h_j)}\right] \tag{5.5}$$

This activation function also is continuous and differentiable, and the exponential component makes the cross-entropy loss computation easier because it gets canceled with the logarithms.

However, both the sigmoid and the softmax activation functions do not return an actual sample from the target variable. They return the parameters for the Bernoulli and Multinoulli distributions, respectively. These outputs are enough for the corresponding loss calculations during training, but they need an extra step in evaluation mode. One alternative consists of using the parameters to sample from the distributions (implementing popular libraries like PyTorch [114]). The most practical alternative is taking each distribution mode: rounding the sigmoid outputs, or taking the position with the maximum value of the softmax output vector (known as the *argmax*). The issue with all these operations (sampling, round, and argmax) is that they are not differentiable, which means that the backpropagation algorithm cannot transfer gradient information correctly, and the learning procedure is affected. The problem is that while generating samples, these troublesome operations may be needed not only in the output layers, meaning that we need to address the backpropagation blocking in some way. For more information on output activation functions and deep learning losses, please refer to [49, Chapter 6 - Deep Feedforward Networks].

5.2.4 The Reparametrization Trick

Neural networks typically perform deterministic operations, but we need stochastic transformations to implement generative models. A simple solution involves considering the sources of randomness as additional inputs to the network and letting the model continue behaving in a deterministic way. The function implemented by the network now appears to be stochastic for an observer that has no access to the random inputs. Additionally, while this function is continuous and differentiable, the gradients can be computed as usual using backpropagation.



Sampling as part of the Network
backpropagation stops
y
sample $\mathcal{N}(\mu, \sigma^2)$
μ
σ
layer
layer
Reparametrization Trick
no backpropagation needed here because it is just an extra input
y
$\mu + \sigma z$
μ
σ
z
sample $\mathcal{N}(0,1)$
layer
layer
backpropagation can continue
backpropagation can continue

Fig. 5.2 Reparametrization Trick.

However, it is also necessary that the random inputs do not depend on the rest of the inputs, so the gradients do not need to travel through the sampling operation.

Consider the example in Figure 5.2 when the random value y is drawn from a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, where the parameters μ and σ need to be part of the learning process. We cannot obtain the derivatives of y with respect to μ and σ because the sampling operation is not differentiable. However, given the known properties of the Gaussian distribution, the sampling operation can be replaced by the transformation $y = \mu + \sigma z$ while we draw z from $\mathcal{N}(0,1)$, which is, in this case, a random input that does not depend on any other input. This transformation is known as the *reparametrization trick*. For more information on this topic, please refer to [49, Chapter 20 - Deep Generative Models].

5.2.5 Autoencoders

An autoencoder is composed of two networks: an encoder Enc with parameters θ_{Enc} and a decoder Dec with parameters θ_{Dec}. The encoder receives as input a sample x and outputs a latent code z, and the decoder takes z to return $\tilde{x}$ which is considered a *reconstruction* of x:

$$z_i = Enc(x_i) \tag{5.6}$$

$$\tilde{x}_i = Dec(z_i) \tag{5.7}$$

The model's goal is to push $\tilde{x}$ as close as possible to x. We can achieve this by measuring the *reconstruction loss*, the difference between x and $\tilde{x}$, and back-propagating the gradients of that loss through the decoder and then to the encoder to adjust both θ_{Dec} and θ_{Enc}. How to measure the loss depends on the nature of the data. For example, in Equation 5.8 we present

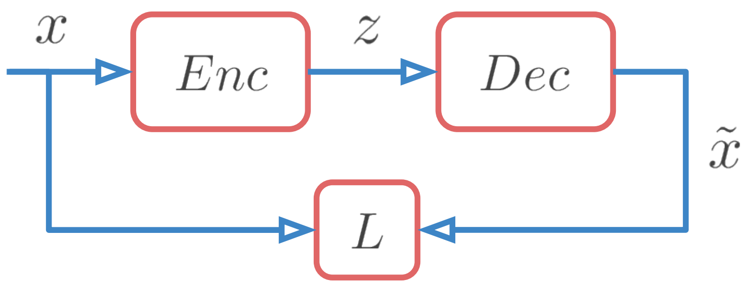


Fig. 5.3 Autoencoder Architecture.

the reconstruction loss for binary features using binary cross-entropy, and for numerical features using mean squared error in Equation 5.9:

$$L_{binrec}(x,\tilde{x}) = \sum_{i=1}^{n} x_i \log(\tilde{x}_i) + (1-x_i)\log(1-\tilde{x}_i) \tag{5.8}$$

$$L_{numrec}(x,\tilde{x}) = \sum_{i=1}^{n} (x_i - \tilde{x}_i)^2 \tag{5.9}$$

Feature scaling, standardization, or normalization is usually applied to numerical features for faster convergence during training. In the case of some losses like the mean squared error, we also need these techniques to avoid assigning more weight to features with a broader range of values. In particular, if numerical features are scaled to the range $[0,1]$, both binary and numerical features would share the same range, allowing the mean squared error to serve as reconstruction loss for both variable types at the same time.

The fact that autoencoders can learn to copy their inputs is not particularly useful. One possible application for them is to assign useful properties from x to z. We can achieve this with *undercomplete* autoencoders by restricting z to a latent space with fewer dimensions than the input space. Undercomplete autoencoders must extract salient properties from x to reconstruct the inputs losing the least possible amount of information. If the decoder employs a linear activation and the selected loss is a mean squared error, an undercomplete autoencoder can learn a PCA transformation. For more applications and variants of autoencoders, please refer to [49, Chapter 14 - Autoencoders].

5.2.6 Variational Autoencoders

The autoencoder's goal is to encode and decode with the smallest possible reconstruction error, without enforcing any structure or restriction over the latent space. Hence, sending any arbitrary data point from the latent space into the decoder might not result in a meaningful or

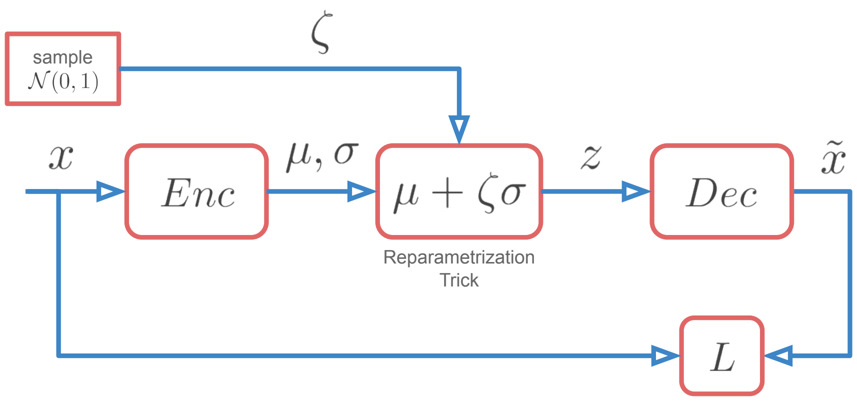


Fig. 5.4 VAE Architecture.

valid sample from the original feature space. In other words, an autoencoder is not well suited for data generation without additional treatment. A Variational Autoencoder (VAE) [66] can be defined as an autoencoder that is regularised during training to ensure that the latent space has properties that make sampling possible. Instead of encoding inputs as single points, VAE encodes inputs as distributions over the latent space. In practice, this means that the encoder returns two outputs representing the parameters μ and σ of a Gaussian distribution. The loss function minimized when training a VAE is composed of a reconstruction term and a regularisation term that affects the latent space's organization by making the distributions returned by the encoder close to a standard normal distribution. Note that during training, the latent codes need to be drawn from the distributions returned by the encoder, then passed to the decoder to obtain the outputs, and finally used to compute the reconstruction loss. To back-propagate to the encoder through the sampling operation, VAE implements the reparametrization trick mentioned in Section 5.2.4. The regularisation term is expressed as the Kulback-Leibler divergence between the returned distribution and a standard Gaussian:

$$\mu_i, \sigma_i = Enc(x_i) \tag{5.10}$$

$$z_i = \mu_i + \zeta_i \sigma_i \text{ for } \zeta_i \sim \mathcal{N}(0,1) \tag{5.11}$$

$$\tilde{x}_i = Dec(z_i) \tag{5.12}$$

$$L(x,\tilde{x}) = L_{rec}(x_i,\tilde{x}_i) + KL(\mathcal{N}(\mu_i,\sigma_i^2)||\mathcal{N}(0,I)) \tag{5.13}$$

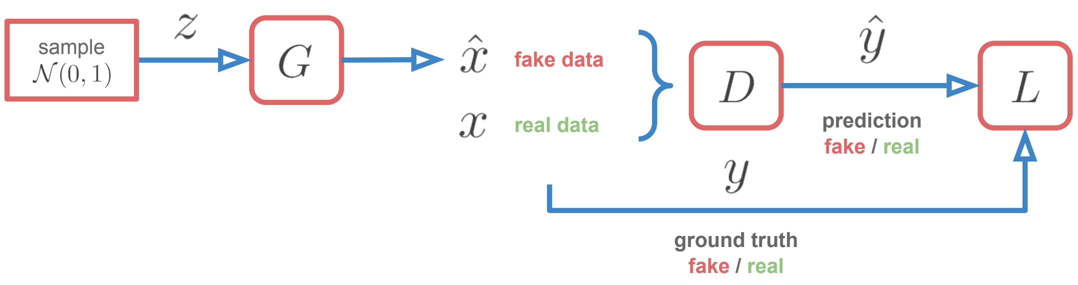


Fig. 5.5 GAN Architecture.

For more information about the theory behind the VAE, and concepts like Variational Inference or the Evidence Lower BOund (ELBO), please refer to [49] or [66].

5.2.7 Generative Adversarial Network

The architecture of a Generative Adversarial Network (GAN) [50] is composed of two networks: the generator and the discriminator. On one side, the generator G with parameters θ_G takes as input a random vector z drawn from a prior P_z and outputs a sample $\hat{x}$. By doing so, G can be considered as a deterministic mapping from the distribution P_z into a distribution defined as P_G. Thus, a sample drawn from P_z determines a latent representation of a sample from P_G. On the other side, the discriminator D with parameters θ_D takes either *real* samples x from a distribution P_{data} or *fake* samples $\hat{x}$ from the distribution P_G and tries to classify the inputs into *real* or *fake*. The goal is to approximate the distribution P_{data} with the distribution P_G by back-propagating the classification loss of D into G. This adversarial setting can be thought of as a two-player minimax game with the following value function:

$$\min_{\theta_G} \max_{\theta_D} \mathbb{E}_{x \sim \mathbb{P}_{data}}[D(x)] + \mathbb{E}_{z \sim \mathbb{P}_z}[1 - D(G(z))] \tag{5.14}$$

If the discriminator trains until convergence, then minimizing the value function is equivalent to minimizing the Jensen-Shannon divergence between $\mathbb{P}_{data}$ and $\mathbb{P}_g$, but that lead to vanishing gradients. In practice, the model is trained by alternating learning steps between the discriminator and the generator optimizing a separate loss for each case:

$$L_D(x,z) = \frac{1}{n} \sum_{i=1}^{n} \log(D(x_i)) + \log(1 - D(G(z_i))) \tag{5.15}$$

$$L_G(z) = \frac{1}{n} \sum_{i=1}^{n} \log(D(G(z_i))) \tag{5.16}$$

where L_D is the discriminator loss and L_G is the generator loss.

5.2.8 Wasserstein GAN

In the article introducing Wasserstein GAN (WGAN) [6], the authors claim that the divergences minimized in GANs are potentially not continuous, leading to training difficulties. They propose to use the Wasserstein-1 distance (also known as Earth-Mover) because it is continuous everywhere and differentiable almost everywhere. Also, they show that the loss they propose is more correlated with sample quality. In practice, the difference is that a critic replaces the discriminator, which returns real values instead of binary outputs. Most importantly, the network parameters are limited or clipped to the range $[-c; c]$ for a small constant c. The losses from Equations 5.15 and 5.16 are respectively modified:

$$L_D(x,z) = \frac{1}{n}\sum_{i=1}^{n} -D(x_i) + D(G(z_i)) \tag{5.17}$$

$$L_G(z) = \frac{1}{n}\sum_{i=1}^{n} -D(G(z_i)) \tag{5.18}$$

The WGAN with Gradient Penalty (WGAN-GP) [51] is an extension of the previous model where the clipping is discarded a new term is added to Equation 5.17:

$$\lambda_{GP}\mathbb{E}_{\hat{x}\sim\mathbb{P}_{\hat{x}}}\left[(||\nabla_{\hat{x}}D(\hat{x})||_2 - 1)^2\right] \tag{5.19}$$

where λ_{GP} is a new hyperparameter called penalty coefficient, and $\mathbb{P}_{\hat{x}}$ is a distribution defined by sampling uniformly along straight lines between pairs of samples from P_{data} and P_G.

5.2.9 Adversarially Regularized Autoencoders

The Adversarially Regularized Autoencoders (ARAE) [159] alternate training steps between an autoencoder, a generator that samples continuous latent codes, and a critic that distinguishes between real and fake codes. The generator loss is the same as in WGAN (Equation 5.18), and for the critic, the loss is similar to Equation 5.17 but includes the encoder:

$$L_D(x,z) = \frac{1}{n}\sum_{i=1}^{n} -D(Enc(x_i)) + D(G(z_i)) \tag{5.20}$$

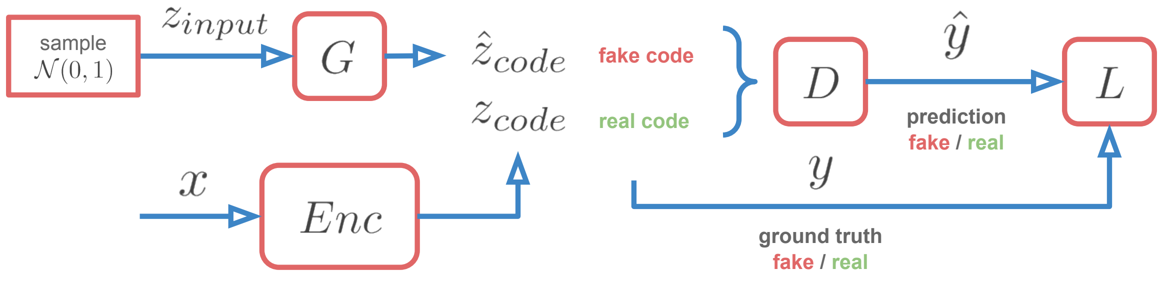


Fig. 5.6 ARAE Architecture.

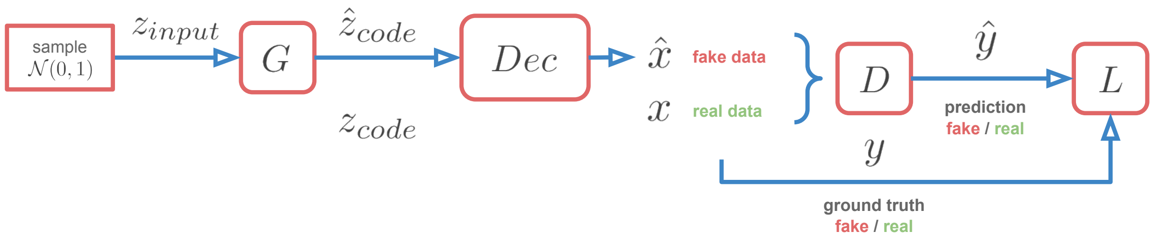


Fig. 5.7 MedGAN Architecture.

5.2.10 MedGAN

MedGAN [29] also implements an autoencoder between the generator and the discriminator, but in this case, the discriminator receives real data or fake data obtained from decoding fake codes. The autoencoder is first pre-trained separately using binary cross-entropy as reconstruction for datasets with binary features. During the main training phase, the gradients flow from the discriminator to the decoder and afterward to the generator. The losses from Equations 5.15 and 5.16 are respectively modified in the following way:

$$L_D(x,z) = \frac{1}{n}\sum_{i=1}^{n} \log(D(x_i)) + \log(1 - D(Dec(G(z_i)))) \tag{5.21}$$

$$L_G(z) = \frac{1}{n}\sum_{i=1}^{n} \log(D(Dec(G(z_i)))) \tag{5.22}$$

Note that the model was conceived only for binary variables. The study [29] mentions an alternative for continuous variables, but it is not clear how to adapt it for tabular data.

5.2.11 Gumbel-Softmax

The Gumbel-Softmax is an activation function that can generate vectors $x = [x_1, \dots, x_d] \in \mathbb{R}^d$ from inputs $h = [h_1, \dots, h_d] \in \mathbb{R}^d$, a temperature hyperparameter $\tau \in (0, \infty)$ and d samples $g = [g_1, \dots, g_d]$ drawn from $Gumbel(0,1) = -\log(-\log(u))$ with $u \sim U(0,1)$ given by:

$$gs(h, g, \tau) = [Softmax((\log(h_1) + g_1)/\tau), \dots, Softmax(\log(h_d) + g_d)/\tau)] \tag{5.23}$$

$$= \left[\frac{exp((\log(h_1) + g_1)/\tau)}{\sum_{j=1}^{d} exp((\log(h_j) + g_j)/\tau)}, \dots, \frac{exp((\log(h_d) + g_d)/\tau)}{\sum_{j=1}^{d} exp((\log(h_j) + g_j)/\tau)} \right] \tag{5.24}$$

As τ tends to zero, the Softmax smoothly approaches the argmax, and the resulting samples approach one-hot. As τ tends to infinity, the samples become uniform. The usual technique involves using continuous vectors during training and discretizing the sample vectors to one-hot vectors during evaluation. However, these two different dynamics lead to poor evaluation performance. The Straight Through Gumbel-Softmax technique ensures the training and the evaluation dynamics are the same using the discrete form in the forward pass and the continuous one during backpropagation. Nevertheless, note that this technique applies to one categorical variable at a time and not directly to tabular data. Please refer to [94, 57] for more information about the theory behind the Gumbel-Softmax activation.

5.2.12 Evaluating the Sample Quality

Evaluating the quality of GANs has proved to be complicated. In [15], the authors provide a discussion of commonly used metrics. Furthermore, not all measures provide a fair assessment of the sample quality, as noted in [136]. This section defines two metrics proposed in [29] that perform a quantitative evaluation of binary datasets.

During experiments, each dataset X is usually partitioned into X_{train} and X_{test}, where $train$ and $test$ indicate row indices following $train \cap test = \emptyset$ and $train \cup test = \{1, \dots, n\}$. The models are trained with X_{train} to generate another dataset called X_{synth}, while the remaining X_{test} is used in different ways to evaluate the sampling quality.

We start with the most straightforward metric, which only checks if the model has learned the independent distribution of ones per dimension correctly. For each of the m feature columns from the datasets X_{synth} and X_{test}, we count the number of ones, and we divide that by the number of samples n, obtaining two vectors of size m. The authors compare both vectors visually using scatter plots. We describe the procedure in Algorithm 5.1 extending it

Algorithm 5.1: Probabilities by feature

```
Function probByFeat(X):
    Data: $X \in \mathbb{R}^{n \times m}$ binary feature matrix
    Result: $v \in \mathbb{R}^{m_{bin}+m_{cat}}$ vector with probabilities by feature
    $j \leftarrow 1$ // output vector position
    $l \leftarrow 1$ // feature column position
    forall variable k do
        for $s \leftarrow 1$ to size(k) do
            if type(k) $\in$ {bin, cat} then
                $v_j \leftarrow \frac{1}{n} \sum_{i=1}^{n} X_{i,l}$
                $j \leftarrow j + 1$
            end
            $l \leftarrow l + 1$
        end
    end
```

for datasets of mixed variable types by counting the number of ones only for feature columns corresponding to binary and categorical variables (it makes no sense for numerical variables).

The last metric problem is that it only captures how well the model can reproduce the independent feature distributions, but it does not consider the dependencies between them. The second method creates one logistic regression model per feature to tackle this, trained with X_{train}, to predict the selected feature using the rest of the features. Then, calculating the f1-score of each feature using the trained model and the corresponding predictions over X_{test}, we obtain a vector with m prediction scores. Repeating the same procedure but training with X_{synth} and evaluating with X_{test} we obtain another vector of the same size. Both vectors are also compared visually by the authors using scatter plots. Algorithm 5.2 summarizes the procedure extending it for datasets of mixed variable types by training logistic regression models only for feature columns corresponding to binary and categorical variables (a binary classification does not work with a numerical target variable).

5.3 Contributions

5.3.1 Multi-Categorical Output Architecture

We propose to adapt the outputs of several architectures presented in the previous section to generate multi-categorical samples. In Figure 5.8 we depict the general idea of the modifications. Which model is affected depends on the architecture and will be explained in the following sections. Regardless of the case, when we need to generate a sample $\hat{x}$, we

Algorithm 5.2: Predictions by feature

```
1  Function predByFeat($X^{train}$, $X^{test}$):
       Data: $X^{train} \in \mathbb{R}^{n^{train} \times m}$ train feature matrix
       Data: $X^{test} \in \mathbb{R}^{n^{test} \times m}$ test feature matrix
       Result: $v \in \mathbb{R}^{m_{bin}+m_{cat}}$ vector with classification f1-scores by feature
2      $j \leftarrow 1$ // output vector position
3      $l \leftarrow 1$ // feature column position
4      forall variable k do
5          for $s \leftarrow 1$ to size($k$) do
6              if type($k$) $\in$ {bin, cat} then
7                  $y^{train} \leftarrow X^{train}_{:,l}$ // separate feature column l from train
8                  $y^{test} \leftarrow X^{test}_{:,l}$ // separate feature column l from test
9                  $X^{train}_{:,\sim l} \leftarrow X^{train}$ without feature column $l$
10                 $X^{test}_{:,\sim l} \leftarrow X^{test}$ without feature column $l$
11                 $M_j \leftarrow$ train a binary classification model with $X^{train}_{:,\sim l}$ as features and
                     the separated feature column $y^{train}$ as labels
12                 $\hat{y}_{test} \leftarrow$ classify features $X^{test}_{:,\sim l}$ with model $M_j$
13                 $v_j \leftarrow$ compute f1-score between predictions $\hat{y}_{test}$ and labels $y^{test}$
14                 $j \leftarrow j+1$
15             end
16             $l \leftarrow l+1$
17         end
18     end
```

connect the model output h to **m** different dense layers working in parallel. Each dense layer k takes h, which could have any arbitrary size, and returns a vector $h^{[k]}$ of size $m^{[k]}$. We then apply an activation function that depends on $\texttt{type}(k)$ over each $h^{[k]}$ to compute $\hat{x}^{[k]}$. When $\texttt{type}(k) = num$, a *ReLU* activation can be used, or no activation at all (a *linear activation*). If $\texttt{type}(k) = bin$, then we simply use a *sigmoid* activation. Finally for $\texttt{type}(k) = cat$ we can select a softmax or Gumbel-softmax as a categorical activation. Following all the activation functions, we concatenate every $\hat{x}^{[k]}$ in the right order to obtain the output sample $\hat{x}$.

5.3.2 Generators with Multi-Categorical Outputs

We adapt the GAN and WGAN-GP architectures by changing the generator's outputs described in the previous section. For the first model, we select the Gumbel-softmax activation function, and for the second, a simple softmax (following the text generation use case on

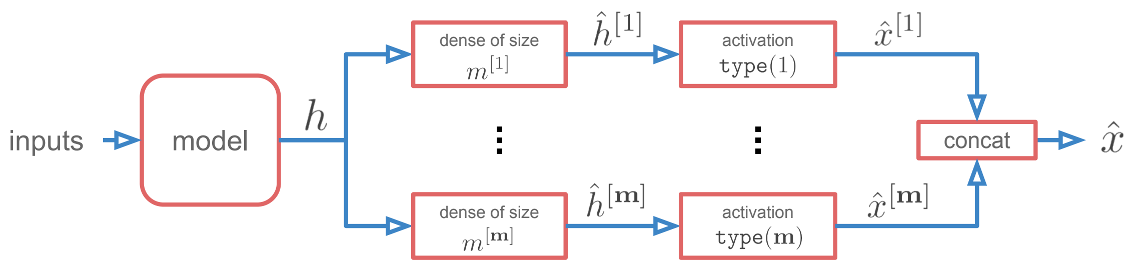


Fig. 5.8 Multi-Output Architecture.

[51]). For the rest of this chapter, we refer to these models as *Multi-Output Gumbel-GAN* (MO-Gumbel-GAN) and *Multi-Output WGAN-GP* (MO-WGAN-GP), respectively.

5.3.3 Autoencoders with Multi-Categorical Outputs

The ARAE and MedGAN architectures involve an autoencoder mixed with a GAN. For both models, the generator outputs latent codes, and the decoder is in charge of mapping those outputs to real samples. Hence, for these architectures, we propose modifying the decoder outputs, using in both cases a Gumbel-softmax activation. The output concatenation remains as proposed at evaluation time, but during training, we leave each $\hat{x}^{[k]}$ separated to calculate a reconstruction loss that takes into account the multi-categorical structure:

$$L_{rec} = \frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{\mathbf{m}} \sum_{q=1}^{\mathtt{size}(k)} -x_i^{[k,q]} \log \hat{x}_i^{[k,q]} \tag{5.25}$$

which is the average over the n samples, of the summation of cross-entropy losses over the features from **m** categorical variables, between every sample and its reconstruction.

5.3.4 Proposals for Evaluating the Sample Quality

In this section, we propose modifications and extensions to the evaluation metrics presented in Section 5.2.12. Regarding the dimension-wise prediction metric, we noticed that as the number of dimensions grows, the performance of the logistic regression models starts to deteriorate. This phenomenon can be misleading because if the f1 score is low, it is not clear if the cause is the dataset's properties (either real or synthetic) or the model's predictive power. For that reason, we decided to replace logistic regression with random forests.

Furthermore, the purpose of this metric is to evaluate binary samples. In our multi-output setting, when categorical variables are present, we represent them using one-hot-encoding. Hence, only one dimension by categorical variable has a value of one. Consequently, for

Algorithm 5.3: Predictions by categorical variable

```
1  Function $predByCatVar(X^{train}, X^{test})$:
       Data: $X^{train} \in \mathbb{R}^{n^{train} \times m}$ train feature matrix
       Data: $X^{test} \in \mathbb{R}^{n^{test} \times m}$ test feature matrix
       Result: $v \in \mathbb{R}^{m_{cat}}$ vector with classification accuracy by categorical variable
2      $j \leftarrow 1$ // output vector position
3      $l \leftarrow 1$ // feature column position
4      forall variable $k$ do
5          if $\texttt{type}(k) = \texttt{cat}$ then
6              $cols \leftarrow$ feature column indices in range $[l, \ldots, l + \texttt{size}(k)]$
7              $y^{train} \leftarrow$ label encoding of $X^{train}_{:,cols}$
8              $y^{test} \leftarrow$ label encoding of $X^{test}_{:,cols}$
9              $X^{train}_{:,\sim cols} \leftarrow X^{train}$ without feature columns in range $cols$
10             $X^{test}_{:,\sim cols} \leftarrow X^{test}$ without feature columns in range $cols$
11             $M_j \leftarrow$ train a multi-class classification model with $X^{train}_{:,\sim cols}$ as features and
                $y^{train}$ as categorical labels
12             $\hat{y}^{test} \leftarrow$ classify features $X^{test}_{:,\sim cols}$ with model $M_j$
13             $v_j \leftarrow$ compute accuracy between predictions $\hat{y}^{test}$ and labels $y^{test}$
14             $j \leftarrow j + 1$
15         end
16         $l \leftarrow l + \texttt{size}(k)$
17     end
```

features corresponding to categorical variables, the task of predicting one dimension based on the remaining ones is almost trivial. The prediction can only take two forms: if there is a one in the remaining features, the selected feature should be a zero; otherwise, it must be a one. Therefore, we propose a third evaluation metric based on the dimension-wise prediction, but predicting one categorical variable at a time. Each predictive model now handles a multi-class classification problem. Thus, we replace the f1-score with the accuracy score. We collect two vectors that we can visually analyze in the same way we mentioned for the previous metrics. In Algorithm 5.3 we describe the proposed third metric.

Following the third metric's logic, we propose to separate one by one each numerical variable as a target for a linear regression model trained with the rest of the variables. We calculate the mean squared error of the regression tasks for both X_{train} and X_{synth}, obtaining the respective regression score vectors. The fourth metric is described in Algorithm 5.4.

In addition to the visual inspection, we introduce a numerical analysis by measuring the mean squared error between real and synthetic vectors for each of the four metrics:

Algorithm 5.4: Regressions by feature

```
1  Function $predByNumVar(X^{train}, X^{test})$:
       Data: $X^{train} \in \mathbb{R}^{n^{train} \times m}$ train feature matrix
       Data: $X^{test} \in \mathbb{R}^{n^{test} \times m}$ test feature matrix
       Result: $v \in \mathbb{R}^{m_{num}}$ vector with regression MSE by feature
2      $j \leftarrow 1$ // output vector position
3      $l \leftarrow 1$ // feature column position
4      forall variable $k$ do
5          if type($k$) = num then
6              $y^{train} \leftarrow X^{train}_{:,l}$ // separate feature column l from train
7              $y^{test} \leftarrow X^{test}_{:,l}$ // separate feature column l from test
8              $X^{train}_{:,\sim l} \leftarrow X^{train}$ without feature column $l$
9              $X^{test}_{:,\sim l} \leftarrow X^{test}$ without feature column $l$
10             $M_j \leftarrow$ train a regression model with $X^{train}_{:,\sim l}$ as features and $y^{train}$ as
                 numerical labels
11             $\hat{y}^{test} \leftarrow$ make a regression from features $X^{test}_{:,\sim l}$ with model $M_j$
12             $v_j \leftarrow$ compute MSE between predictions $\hat{y}^{test}$ and labels $y^{test}$
13             $j \leftarrow j + 1$
14         end
15         $l \leftarrow l +$ size($k$)
16     end
```

$$MSE_p = \frac{1}{m^{bin} + m^{cat}} \left\| probByFeat(X^{test}) - probByFeat(X^{synth}) \right\|^2 \tag{5.26}$$

$$MSE_f = \frac{1}{m^{bin} + m^{cat}} \left\| predByFeat(X^{train}, X^{test}) - predByFeat(X^{synth}, X^{test}) \right\|^2 \tag{5.27}$$

$$MSE_a = \frac{1}{m^{cat}} \left\| predByCatVar(X^{train}, X^{test}) - predByCatVar(X^{synth}, X^{test}) \right\|^2 \tag{5.28}$$

$$MSE_r = \frac{1}{m^{num}} \left\| predByNumVar(X^{train}, X^{test}) - predByNumVar(X^{synth}, X^{test}) \right\|^2 \tag{5.29}$$

Table 5.1 Dataset statistics.

Name	n	$\mathbf{m}$	$\mathbf{m_{cat}}$	$\mathbf{m_{num}}$	m	m_{min}	m_{max}
Fixed 2	10K	10	10	0	20	2	2
Fixed 10	10K	10	10	0	100	10	10
Mix Small	10K	10	10	0	68	2	10
Mix Big	10K	100	100	0	635	2	10
Census	2.5M	68	68	0	396	2	18
Forest	39K	12	2	10	54	1	40

5.4 Experiments

To assess our proposal's quality, we conduct experiments with varying data properties: sparsity, dimensionality, number of categories, and sample size. We first experiment with datasets involving only categorical variables, and the second set adds numerical variables.

5.4.1 Datasets

To have better control of our experiments, we synthesize several datasets of one-hot encoded categorical variables. While we select a uniform distribution for the first categorical variable, we randomly distribute each remaining variable based on the previous one's value. With this procedure, we present four datasets of 10K samples: two with ten variables of fixed size 2 and 10 respectively, one with ten variables with random size between 2 and 10, and a bigger dataset of 100 variables with random size between 2 and 10. This setting allows us to evaluate the models with different configurations of sparsity and dimensionality.

Additionally, we selected from the UCI Machine Learning Repository [37] two datasets to get more real use cases of tabular data. The *US Census Data 1990 Data Set* (Census)[1] contains almost 2.5M samples with 68 variables, each of them with a size in the range from 2 to 18, transformed into 396 features. The *Forest Cover Type Data Set* (Forest) [2] contains around 39K samples with 10 numerical variables, plus 1 categorical variable of size 40 and another of size 4, transformed altogether into 54 features. This last dataset is used only for the experiments with mixed variable types. A summary is presented in Table 5.1.

[1]https://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990)
[2]https://archive.ics.uci.edu/ml/datasets/Covertype

5.4.2 Implementation details

We developed all our experiments using PyTorch 0.4.0[3]. Based on [49, Chapter 11 - Practical Methodology], we chose hyperparameters with random search followed by manual tuning. We trained using Adam [65] with a minibatch size of 100 for 2000 epochs with a learning rate of 10^{-3}. Compared to the other models, we had to experiment more to find the right setting for ARAE. We obtained better results by removing the original code normalization and noise annealing, training for more epochs with a lower learning rate of 10^{-5}.

Autoencoders were implemented with MLPs using TanH hidden activations. Generators and discriminators were implemented with MLPs with batch norm between layers (setting the decay to 0.99), using ReLU activations for the generators and LeakyReLU for the discriminators. For ARAE, we used the same amount of steps for the three training stages, and for the rest, we used two steps for the discriminator for every step of the other components.

In particular, we defined for ARAE the critic weights range to $[-0.1; 0.1]$. We used three hidden layers of size 100 for the generator and one for the discriminator, except for the MedGAN models where we used two hidden layers in the discriminator of sizes 256 and 128 and two hidden layers in the generator with the same size as the latent code because of the residual connections. For models with autoencoders, in the baseline cases, the decoder used sigmoid outputs, and for the multi-categorical versions, we selected Gumbel-Softmax outputs with temperature $\tau = 2/3$. Finally, for the simpler GAN with Gumbel-Softmax outputs, the temperature was set to a smaller value of $\tau = 1/3$.

5.4.3 Results

In Table 5.2 we summarize the experiments we executed for every model on each dataset. Because of the randomness during training, we need to ensure that we did not get satisfactory results just by chance. Thus, we trained every model three times with different random seeds. For each of the three proposed metrics, we present the mean and the standard deviation of the runs' values using different seeds. We make the following observations:

- The standard deviation for every case is usually smaller than 0.01, which indicates no significant difference in the metrics between runs and that the results are not a product of mere chance.
- For the FIXED 2 dataset, the multi-output models' improvement is not evident. However, it is reasonable because samples with only two possible values per categorical

[3]Our open-source implementation is available at https://github.com/rcamino/multi-categorical-gans

Table 5.2 The three proposed metrics for every dataset and every model.

Method	Dataset	MSE_p	MSE_f	MSE_a
ARAE	FIXED 2	0.00031 ± 0.00004	**0.00001** ± 0.00001	0.00059 ± 0.00022
MedGAN	FIXED 2	0.00036 ± 0.00031	0.00005 ± 0.00003	0.00056 ± 0.00033
MO-ARAE	FIXED 2	0.00046 ± 0.00028	**0.00001** ± 0.00000	0.00058 ± 0.00024
MO-MedGAN	FIXED 2	**0.00013** ± 0.00006	**0.00000** ± 0.00000	**0.00032** ± 0.00017
MO-Gumbel-GAN	FIXED 2	0.00337 ± 0.00188	0.00014 ± 0.00012	0.00050 ± 0.00012
MO-WGAN-GP	FIXED 2	0.00030 ± 0.00007	**0.00001** ± 0.00000	0.00068 ± 0.00012
ARAE	FIXED 10	0.00398 ± 0.00002	0.00274 ± 0.00021	0.02156 ± 0.00175
MedGAN	FIXED 10	0.00720 ± 0.00825	0.00463 ± 0.00404	0.01961 ± 0.00214
MO-ARAE	FIXED 10	0.00266 ± 0.00009	**0.00036** ± 0.00018	0.01086 ± 0.00159
MO-MedGAN	FIXED 10	**0.00022** ± 0.00003	0.00167 ± 0.00010	0.00062 ± 0.00044
MO-Gumbel-GAN	FIXED 10	0.00056 ± 0.00006	0.00110 ± 0.00013	0.00055 ± 0.00035
MO-WGAN-GP	FIXED 10	0.00026 ± 0.00001	0.00123 ± 0.00005	**0.00048** ± 0.00010
ARAE	MIX SMALL	0.00261 ± 0.00020	0.01303 ± 0.00146	0.01560 ± 0.00039
MedGAN	MIX SMALL	0.00083 ± 0.00039	0.01889 ± 0.00258	0.02070 ± 0.00170
MO-ARAE	MIX SMALL	0.00195 ± 0.00040	**0.00081** ± 0.00018	0.00759 ± 0.00100
MO-MedGAN	MIX SMALL	**0.00029** ± 0.00003	0.00133 ± 0.00012	0.00080 ± 0.00018
MO-Gumbel-GAN	MIX SMALL	0.00078 ± 0.00027	0.00104 ± 0.00013	0.00047 ± 0.00008
MO-WGAN-GP	MIX SMALL	0.00048 ± 0.00010	0.00140 ± 0.00014	**0.00037** ± 0.00016
ARAE	MIX BIG	0.04209 ± 0.00362	0.02075 ± 0.01144	0.00519 ± 0.00087
MedGAN	MIX BIG	0.01023 ± 0.00263	0.00211 ± 0.00033	0.00708 ± 0.00162
MO-ARAE	MIX BIG	0.00800 ± 0.00019	0.00249 ± 0.00035	0.00472 ± 0.00092
MO-MedGAN	MIX BIG	**0.00142** ± 0.00015	0.00491 ± 0.00055	0.01309 ± 0.00106
MO-Gumbel-GAN	MIX BIG	0.00312 ± 0.00032	**0.00194** ± 0.00017	**0.00430** ± 0.00021
MO-WGAN-GP	MIX BIG	0.00144 ± 0.00006	0.00536 ± 0.00030	0.01664 ± 0.00177
ARAE	CENSUS	0.00165 ± 0.00082	0.00206 ± 0.00030	0.00668 ± 0.00175
MedGAN	CENSUS	0.00871 ± 0.01078	0.00709 ± 0.00889	0.01723 ± 0.02177
MO-ARAE	CENSUS	0.00333 ± 0.00020	0.00129 ± 0.00019	0.00360 ± 0.00095
MO-MedGAN	CENSUS	**0.00012** ± 0.00004	0.00024 ± 0.00003	0.00013 ± 0.00003
MO-Gumbel-GAN	CENSUS	0.01866 ± 0.00040	0.00981 ± 0.00034	0.03930 ± 0.00469
MO-WGAN-GP	CENSUS	0.00019 ± 0.00004	**0.00017** ± 0.00002	**0.00008** ± 0.00002

variable are equivalent to binary samples, and the baseline models are well suited to that kind of data.

- Using variables with the same size or with different sizes does not affect the result.
- The size and sparsity of the samples seem to be the most critical factors. When the datasets have more variables and contain more possible values, the dependencies between dimensions become more complex, and also the independent proportion of ones per dimension is harder to learn. The experiments show that no model outperforms the rest for every case. It is reasonable to claim that a different model can be the best for different settings, but in general, we can observe that the multi-output models provide the best results.

Table 5.3 The four proposed metrics for every model on the FOREST dataset.

Method	Dataset	MSE_p	MSE_f	MSE_a	MSE_r
ARAE	FOREST	0.00042	0.00158	0.03779	0.00647
MedGAN	FOREST	0.00004	0.00010	0.02418	0.00272
MO-ARAE	FOREST	0.00007	0.00043	0.03874	0.00007
MO-MedGAN	FOREST	**0.00002**	**0.00007**	0.01143	**0.00001**
MO-Gumbel-GAN	FOREST	0.00104	0.00015	**0.00945**	0.00011
MO-WGAN-GP	FOREST	**0.00002**	0.00015	0.02651	0.00023

- We notice cases when, for example, Gumbel-GAN is the best method in the MIX BIG experiments, but the worst in the CENSUS dataset. Given that we deal with a considerable amount of hyperparameters, it might be valuable to do a more exhaustive hyperparameter search to identify the reason for the variance in performance.

As an example, we show in Figure 5.9 the scatter plots for the MIX SMALL dataset, that are associated with the values summarized in the third block of Table 5.2. Because we plot ground-truth versus model output, perfect results should fall onto the diagonal line. We can see how the first column is reasonably right for every model and how the second column is almost perfect for our approach (as explained in 5.3.4). Finally, for the last column, the last four rows corresponding to multi-output models present better performance than the rest.

5.4.4 Extension for Mixed Variable Types

In Table 5.3, we describe the four metric values using the Forest dataset for the two baselines and the four proposed models. Note that since the table has an additional column, we removed the standard deviation information for readability. The results are consistent with the results from the multi-output setting presented in Table 5.2, showing that proposed models outperform the baselines for most of the metrics.

5.5 Conclusion

We extended the autoencoder of MedGAN and ARAE and the generator of Gumbel-GAN and WGAN-GP with dense layers feeding into (Gumbel) softmax layers, each separated into the right dimensionality for their corresponding output variables. Additionally, we adapted a metric to our categorical setting to contrast all approaches.

Compared to unmodified MedGAN and ARAE, all approaches improve the performance across all datasets, even though we cannot identify the definite best model. The performance improvement comes at the cost of adding extra information, namely the dimensionality of the variables. In some cases, this information may not be available.

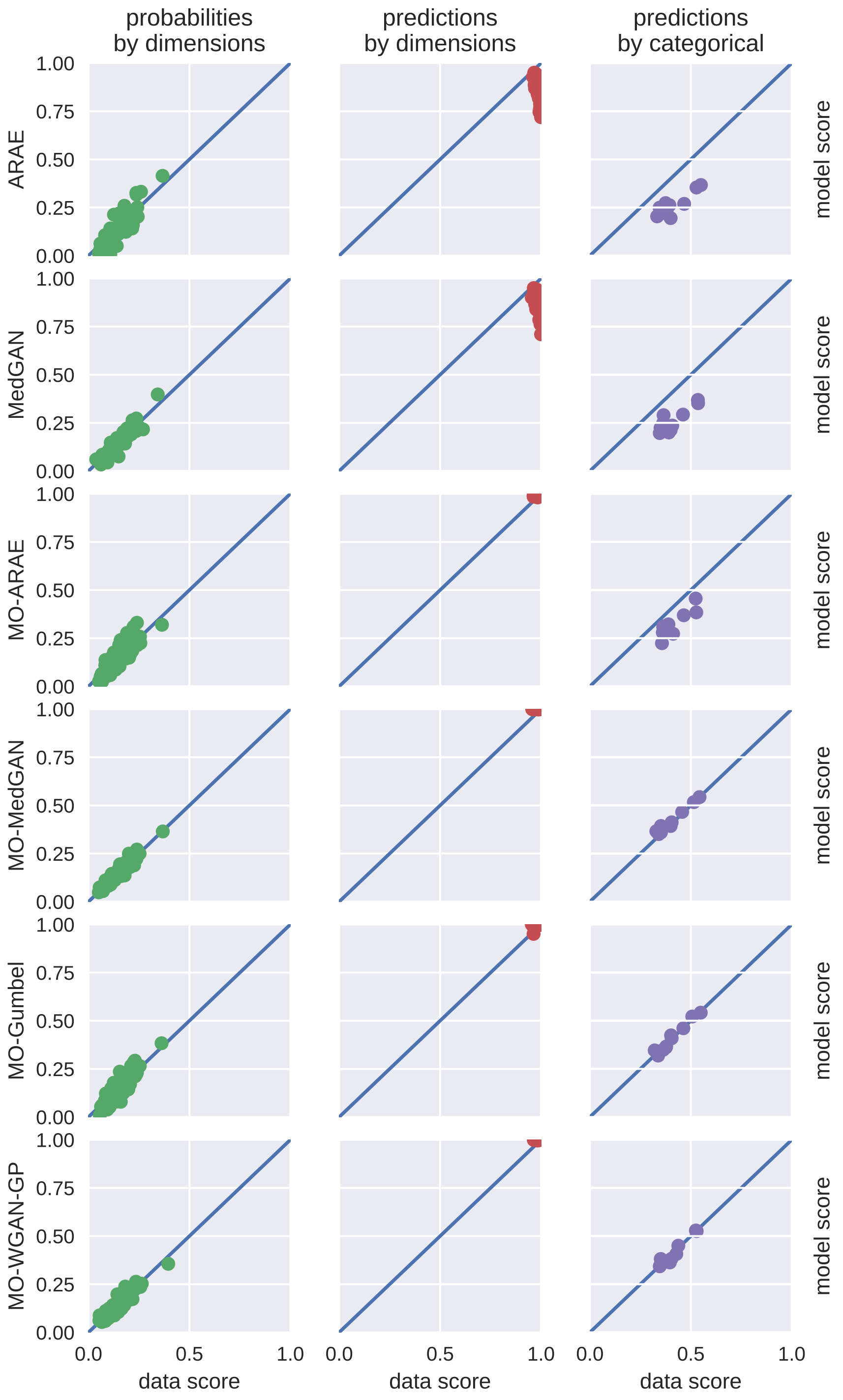


Fig. 5.9 Example scatter plots for a particular run on the MIX SMALL dataset.

Note that other applications might require different evaluation methods to determine the samples' quality generated by each approach. This requirement might also guide the search for the right hyperparameters and architecture options.

Chapter 6

Oversampling Tabular Data with DGM

6.1 Motivation

In practice, machine learning experts must often confront imbalanced data. Without accounting for the imbalance, common classifiers perform poorly, and standard evaluation metrics mislead the practitioners on the model's performance. A standard method to treat imbalanced datasets is under- and oversampling. The undersampling process removes samples from the majority class, while the oversampling process adds synthetic samples to the minority class. In this chapter, we follow up on recent developments in deep learning. We take proposals of deep generative models, including our own, and study the possibility of providing realistic samples that improve performance on imbalanced classification tasks via oversampling.

Across 160K+ experiments, we show that all of the new methods tend to perform better than simple baseline methods such as SMOTE but require different under- and oversampling ratios to do so. Our experiments show that the sampling method does not affect quality, but runtime varies widely. We also observe that the improvements in terms of performance metric, while shown to be significant when ranking the methods, often are minor in absolute terms, especially compared to the required effort. Furthermore, we notice that a large part of the improvement is due to undersampling, not oversampling.

6.2 Comparison

6.2.1 Datasets

Datasets like MNIST[1], ImageNet[2], and CIFAR[3] are very well known in the domain of computer vision. Thanks to this, countless studies could compare their findings against others using widely accepted benchmarks. On the other side, when applying deep learning to tabular data, no framework is well defined. Several papers opted to work with datasets from the UCI Repository [37] on tasks related to this domain: tabular data imputation [153, 108, 48, 98], imbalanced classification using a latent space [109], oversampling from deep generative models [35, 149] or all the previous tasks at the same time [55]. Nevertheless, we want to point out several interesting aspects of the datasets selected across these studies. First of all, the datasets are usually presented with short names or aliases, leading to confusion. For example, one dataset referred to as "breast" or "breast-cancer" presents four versions online [4][5][6][7] with different features, and sometimes it is not very clear which one the study is referring to. Second, some datasets contain less than a thousand samples or just a few features. While this can be reasonable for other machine learning models, deep learning models may not reach their full potential when training with datasets of such dimensions. Finally, state-of-the-art machine learning algorithms like XGBoost [25] can solve most of the classification tasks associated with imbalanced UCI datasets without much need for any additional treatment. For example, one study [109] showed that a simple Random Forest [79] could classify 14 datasets from the UCI Repository with $ROCAUC > 0.9$ (and for most of the cases very close to 1). Some examples are included in Section 6.3.

6.2.2 Deep Generative Models

In this chapter, we compare two well known deep generative architectures: GAN [50] and VAE [66]. We include as well two variants of GAN that involve autoencoders, ARAE [159] and MedGAN [29], adapting the reconstruction loss by separating the features per variable (e.g., all the features from a one-hot-encoded categorical variable). Each separated reconstruction loss depends on the variable type: cross-entropy for categorical variables, binary-cross entropy for binary variables, and mean squared error for numerical variables. Please refer to

[1]http://yann.lecun.com/exdb/mnist/
[2]http://www.image-net.org/
[3]https://www.cs.toronto.edu/ kriz/cifar.html
[4]https://archive.ics.uci.edu/ml/datasets/Breast+Cancer
[5]https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)
[6]https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)
[7]https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Prognostic)

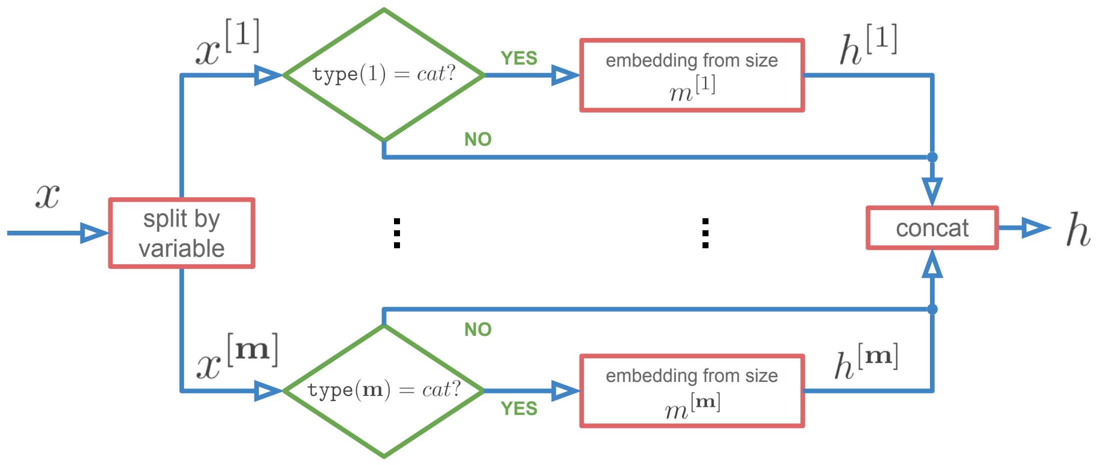

x
split by variable
$x^{[1]}$
$\mathtt{type}(1) = cat?$
YES
NO
embedding from size $m^{[1]}$
$h^{[1]}$
$x^{[\mathbf{m}]}$
$\mathtt{type}(\mathbf{m}) = cat?$
NO
YES
embedding from size $m^{[\mathbf{m}]}$
$h^{[\mathbf{m}]}$
concat
h

Fig. 6.1 Multi-Input Architecture.

Section 5.2 for more details about this architecture. Additionally, HI-VAE [108] contains details for a more complex version of the architecture.

For each of the models mentioned above, we include the Multi-Output variants presented in Section 5.3. We select Gumbel-softmax [57, 94] for the categorical activations. For all the Multi-Output versions of GAN, we use WGAN [6] instead, and we also add an alternative with WGAN-GP [51]. In addition to the output modifications, we add a Multi-Input layer on each model, as shown in Figure 6.1. This layer uses metadata to split the inputs x into variables $x^{[k]}$ and applies a different embedding to each categorical variable. Each embedding reduces by half the dimension $m^{[k]}$ of the corresponding categorical variable $x^{[k]}$ bounded to the range $[2, 50]$ as illustrated in the following equation:

$$\max(2, \min(50, \left\lceil \frac{m^{[k]}}{2} \right\rceil)) \tag{6.1}$$

This transformation allows us to work in a dense, continuous, and lower-dimensional space, which is better for training than the sparse, discrete, and higher dimensional one-hot-encoded input. The transformed variables are concatenated back with the other non-categorical variables. We can then connect the resulting modified inputs h to the next layer. We call Multi-Variable (MV) to the final architecture that includes the modified inputs and outputs.

6.2.3 Sampling from Deep Generative Models

Among the literature, we found two alternatives to generate synthetic samples with deep generative models for later use in imbalanced classification tasks. The authors in [39] train a GAN only using the minority class samples to improve the detection of credit card frauds.

Table 6.1 Comparison of sampling strategies.

	Minority	Conditional	Rejection
Training data	only features from minority class	features and labels from both classes	features with labels appended from both classes
Changes to all models	-	add label as input	label implicitly added as input by the concatenation
Changes to training loss	-	-	label implicitly part of the reconstruction loss (if present)
Sampling during evaluation	just input noise	noise and positive label as input	noise as input, separate label from output, discard samples from negative class, keep trying until obtaining desired sample size (can timeout)

We call this technique “minority”. The second method [35] uses GANs with a condition (the label) as an additional input for the generator and the discriminator. We will refer to this alternative as “conditional”. We propose an additional technique that we call “rejection”, that consists of training any deep generative model attaching the label to the rest of the features as an additional variable. Afterward, the method discards all the samples that do not belong to the desired class during the synthetic generation. Note that iterative draws are necessary for obtaining a synthetic sample of a specific size with this technique, and the procedure could never end. Therefore, the method needs a limit on the number of draws or the execution time. Table 6.1 presents a summary and comparison of these three sampling strategies.

6.3 Experiments

In this section, we provide the details and the analysis of our experiments. The goal is to provide an empirical comparison of how different under- and oversampling techniques affect

Table 6.2 Dataset statistics.

Name	n	**m**	$\mathbf{m_{bin}}$	$\mathbf{m_{cat}}$	$\mathbf{m_{num}}$	m	classes
Adult	45K	14	0	8	6	105	2
Breast Cancer Wisconsin (Diagnostic)	569	30	0	0	30	30	2
Credit Card Fraud	284K	29	0	0	29	29	2
Default of Credit Card Clients	30K	23	0	9	14	99	2
Forest Cover Type	581K	12	0	2	10	54	7
Letter Recognition	20K	16	0	0	16	16	26
Spambase	4.6K	57	0	0	57	57	2

imbalanced classification. The code for classification, undersampling, oversampling, and all the tasks related to deep generative models are publicly online [8].

6.3.1 Datasets

We initially select six datasets from the UCI Repository [37]: "Adult"[9], "Breast Cancer Wisconsin (Diagnostic)"[10], "Default of Credit Card Clients"[11], "Forest Cover Type"[12], "Letter Recognition"[13] and "Spambase"[14]. These specific datasets from the UCI repository are the most used among the related work. Some of them are multi-class problems, but we need to convert them into imbalanced binary classification problems to be useful for our study. The dataset "Letter Recognition" contains the same amount of samples per letter in the English alphabet. Thus we can implement 26 one-vs-all binary classification problems, where on each problem, one letter serves as the positive class and all the rest are part of the negative class. We also transformed "Forest Cover Type" into several one-vs-one binary classification problems by pairing two classes at a time. The next section describes more details about the classification experiments involving the UCI repository. We also include the "Credit card fraud" dataset presented in [31], which we obtained from the Kaggle repositories[15]. This dataset only contains numerical features (most of them from coming from a PCA transformation) and is highly imbalanced ($< 0.001\%$ of the cases are frauds). For any dataset, we apply the same preprocessing procedure. All the numerical variables are scaled to fit inside the range $[0;1]$. Additionally, we use one-hot-encoding for all the categorical variables, but we represent them with one binary feature for the particular case of binary variables. We

[8]https://github.com/rcamino/deep-generative-models
[9]http://archive.ics.uci.edu/ml/datasets/adult
[10]http://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(diagnostic)
[11]https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients
[12]http://archive.ics.uci.edu/ml/datasets/covertype
[13]http://archive.ics.uci.edu/ml/datasets/letter+recognition
[14]http://archive.ics.uci.edu/ml/datasets/spambase
[15]https://www.kaggle.com/mlg-ulb/creditcardfraud

generate metadata indicating for each variable the type (categorical, binary, or numerical) and the size (number of features). The SMOTE-NC oversampling method, the autoencoder reconstruction loss, and the multi-variable architectures use this information. The code for the preprocessing is available online [16]. In Table 6.2 we describe the datasets, where n is the number of samples, $\mathbf{m}$ is the number of variables, $\mathbf{m_{bin}}$, $\mathbf{m_{cat}}$, and $\mathbf{m_{num}}$ are the number of variables of binary, categorical and numerical types respectively, and m is the number of features after encoding the variables.

6.3.2 Classification

All of our experiments involve binary classification tasks implemented with XGBoost [25]. We compute the mean and standard deviation of the f1 score for the train and test sets over ten folds. For each dataset, we run a grid search over several XGBoost hyperparameters (the max depth and number of estimators). In Table 6.3, we present the results for the best hyperparameters for the binary classification experiments based on XGBoost. The multi-class datasets were transformed into several binary classification problems by pairing two classes at a time or one class versus the others. Note that most of the cases have very high test scores without any under- or oversampling. We found only three cases where the test f1 score was less than 0.95: “Adult”, “Credit Card Fraud” and “Default of Credit Card Clients”. Furthermore, we present in Figure 6.2 for these three datasets how train and test f1 score changes while the amount of XGBoost estimators are incremented, for a fixed maximum depth. We can see that the test curve converges in the early steps while the training curve keeps growing, which indicates that the models start to overfit with a small number of estimators. It sounds reasonable given that the imbalance is very high for these cases, which confirms that they are hard to solve. The selected XGBoost hyperparameters shown in Table 6.3 remain fixed for the rest of the under- and oversampling experiments.

6.3.3 Undersampling and Oversampling

The imbalanced-learn library [77] implements all the undersampling and oversampling algorithms presented in the experiments that do not involve deep generative models. The three selected datasets for the following experiments are imbalanced, which means that the imbalance ratio (IR) is less than one, where IR is defined as:

$$IR = \frac{|\{\text{minority class samples}\}|}{|\{\text{majority class samples}\}|} \tag{6.2}$$

[16]https://github.com/rcamino/dataset-pre-processing

Table 6.3 Best hyperparameters for baseline binary classification experiments.

Dataset	Positive Class	Negative Classes	Max Depth	Estimators	Train f1	Test f1
Adult	1	0	3	225	0.738	0.732
Breast Cancer Wisconsin (Diagnostic)	1	0	2	205	1.000	0.958
Credit Card Fraud	1	0	4	18	0.898	0.896
Forest Cover Type	0	1	5	1000	0.962	0.941
	2	0	3	280	1.000	1.000
	3	0	2	1	1.000	1.000
	4	0	4	891	1.000	0.999
	5	0	5	610	1.000	1.000
	6	0	5	984	1.000	0.998
	2	1	5	940	1.000	0.999
	3	1	2	143	1.000	1.000
	4	1	5	995	1.000	0.997
	5	1	5	569	1.000	0.999
	6	1	5	528	1.000	1.000
	3	2	4	267	1.000	0.994
	4	2	5	113	1.000	0.997
	5	2	5	972	1.000	0.979
	6	2	2	23	1.000	1.000
	3	4	2	1	1.000	1.000
	3	5	5	245	1.000	0.965
	3	6	2	1	1.000	1.000
	4	5	4	738	1.000	0.997
	4	6	2	23	1.000	1.000
	5	6	2	22	1.000	1.000
Default of Credit Card Clients	1	0	3	3	0.494	0.496
Letter Recognition	0	others	5	137	1.000	0.995
	1	others	5	262	1.000	0.937
	2	others	4	235	1.000	0.972
	3	others	4	316	1.000	0.950
	4	others	4	504	1.000	0.941
	5	others	3	774	1.000	0.952
	6	others	5	311	1.000	0.966
	7	others	5	688	1.000	0.939
	8	others	3	195	0.996	0.966
	9	others	2	830	0.999	0.957
	10	others	2	817	0.993	0.945
	11	others	2	672	1.000	0.963
	12	others	2	251	0.995	0.990
	13	others	3	337	1.000	0.979
	14	others	5	600	1.000	0.951
	15	others	5	456	1.000	0.980
	16	others	5	87	0.999	0.975
	17	others	5	370	1.000	0.950
	18	others	3	538	1.000	0.969
	19	others	4	171	1.000	0.987
	20	others	4	115	0.999	0.977
	21	others	3	227	0.998	0.975
	22	others	2	379	0.999	0.987
	23	others	4	357	1.000	0.990
	24	others	5	350	1.000	0.974
	25	others	4	129	1.000	0.985
Spambase	1	0	2	303	0.985	0.954

We undersample the majority class on the train set, applying a random under sampler, which removes samples from the majority class with a uniform probability until reaching the desired sample size smaller than the original. The result contains a larger IR that we call the “undersampling ratio” (USR). Afterward, we train and evaluate the classification model and

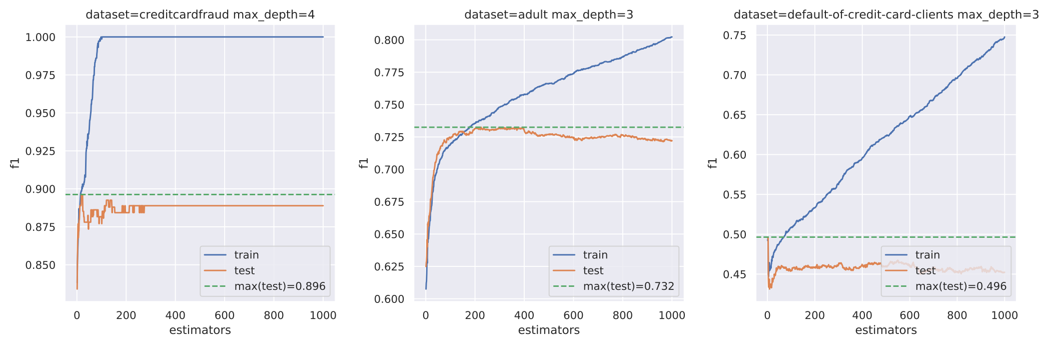


Fig. 6.2 Classification baseline train and test f1 scores by number of estimators.

compute the respective metrics. Furthermore, for each USR, we oversample the minority class on the train set that was previously undersampled. The result contains an even larger IR that we call the “oversampling ratio” (OSR). In Algorithm 6.1 we describe this whole process. We call this algorithm for different oversampling algorithms and incrementally larger values of USR and OSR.

6.3.4 Oversampling with Deep Generative Models

For the final experiments, we combine four deep generative models (or nine when we employ multi-variable alternatives) with three different strategies to draw samples from them (presented in Section 6.2). We implemented all the models with PyTorch [114]. The models’ training involves the training set, and when an autoencoder is present in the architecture, we separate a validation set from the training set to evaluate the quality of the reconstruction. After the training, the resulting model generates synthetic samples, and we append them to the training set until reaching the desired OSR. We report the experiments using 10-fold cross-validation. Algorithm 6.2 describes the whole procedure.

To ensure a fair comparison between models, we arbitrarily fix the amount and size of hidden layers for each model, so the resulting amount of parameters for every model is reasonably close to the other models given a specific dataset. The common hyperparameters also remain fixed across models: the number of epochs is 1000, the learning rate is $1e^{-3}$ and the batch size is 1000. All the models that require alternating training steps proceed one step at a time for every involved model. When we employ WGAN, we clamp the parameters to the range $[-0.01, 0.01]$ and set the gradient penalty’s weight to 0.1.

Algorithm 6.1: Imbalanced classification

```
1  Function $imbalancedClassification(X, y, folds, IR, US, USR, OS, OSR)$:
       Data: $X \in \mathbb{R}^{n \times m}$ feature matrix
       Data: $y \in \{0,1\} R^n$ binary label vector
       Data: $folds \in \mathbb{R}$ number of folds
       Data: $IR \in \mathbb{R}$, $0 < IR$ imbalance ratio
       Data: $US$ undersampling algorithm
       Data: $USR \in \mathbb{R}$, $IR \leq USR$ desired imbalance ratio after undersampling
       Data: $OS$ oversampling algorithm
       Data: $OSR \in \mathbb{R}$, $USR \leq OSR$ desired imbalance ratio after oversampling
       Result: $\mu^{train}$ classification train score mean
       Result: $\sigma^{train}$ classification test score standard deviation
       Result: $\mu^{test}$ classification test score mean
       Result: $\sigma^{test}$ classification train score standard deviation
2      $scores^{train} \leftarrow$ empty vector of size $folds$
3      $scores^{test} \leftarrow$ empty vector of size $folds$
4      for $fold \leftarrow 1$ to $folds$ do
5          $X^{train}, y^{train}, X^{test}, y^{test} \leftarrow foldSplit(X, y, fold, folds)$
6          if $IR < USR$ then
7              $X^{train}, y^{train} \leftarrow US(X^{train}, y^{train}, IR, USR)$
8          end
9          if $USR < OSR$ then
10             $X^{train}, y^{train} \leftarrow OS(X^{train}, y^{train}, USR, OSR)$
11         end
12         $M \leftarrow$ train classification model with features $X^{train}$ and labels $y^{train}$
13         $\hat{y}^{train} \leftarrow$ classify features $X^{train}$ with model $M$
14         $scores^{train}_{fold} \leftarrow$ score between predictions $\hat{y}^{train}$ and labels $y^{train}$
15         $\hat{y}^{test} \leftarrow$ classify features $X^{test}$ with model $M$
16         $scores^{test}_{fold} \leftarrow$ score between predictions $\hat{y}^{test}$ and labels $y^{test}$
17     end
18     $\mu^{train} \leftarrow mean(scores^{train})$
19     $\sigma^{train} \leftarrow std(scores^{train})$
20     $\mu^{test} \leftarrow mean(scores^{test})$
21     $\sigma^{test} \leftarrow std(scores^{test})$
```

6.3.5 Results

We present in Table 6.5, Table 6.6, and Table 6.7 the results for the classification experiments for the “Adult”, “Credit Card Fraud” and “Default of Credit Card Clients” datasets respectively. Each table presents the mean and standard deviation of the train and test f1-score for different experiments divided into four parts. In the first part, we indicate the initial IR of the

Algorithm 6.2: Oversampling with DGM

```
1 Function oversamplingDGM($X^{train}, y^{train}, OSR|S$):
      Data: $X^{train} \in \mathbb{R}^{n^{train} \times m}$ train feature matrix
      Data: $y^{train} \in \{0,1\} R^{n^{train}}$ train binary label vector
      Data: $IR \in \mathbb{R}$, $0 < IR$ imbalance ratio
      Data: $OSR \in \mathbb{R}$, $IR \leq OSR$ desired imbalance ratio after oversampling
      Data: $S$ sampling strategy
      Result: $X^{train}$ with imbalance ratio equal to $OSR$
      Result: $y^{train}$ with imbalance ratio equal to $OSR$
2     $M \leftarrow$ train deep generative model from features $X^{train}$ and labels $y^{train}$
        implemented by sampling strategy $S$
3     while $IR < OSR$ do
4         $x^{synth} \leftarrow$ draw samples from $M$ implemented by sampling strategy $S$
5         $X^{train} \leftarrow X^{train} \cup \{x^{synth}\}$
6         $y^{train} \leftarrow y^{train} \cup \{1\}$ // synthetic label is always positive
7         $IR \leftarrow$ compute imbalance ratio from updated $y^{train}$
8     end
```

Table 6.4 Range of DGM parameters for each dataset and sampling strategy.

Dataset	Sampling Strategy	Model Count	Model Parameters	
			min	max
Adult	Minority	90	11887	15755
	Rejection	90	11988	15856
	Conditional	90	12093	15910
Default of Credit Card Clients	Minority	90	11015	15149
	Rejection	90	11116	15250
	Conditional	90	11215	15298
Credit Card Fraud	Minority	40	1170	1394
	Rejection	40	1201	1425
	Conditional	40	1200	1438

dataset and the results without any under- or oversampling. In the second part, we present the USR that provides the best results using only undersampling. The third part shows each imbalanced-learn algorithm, with the USR-OSR combination providing the best scores with under- and oversampling. Finally, the last part describes the USR-OSR combination that provides the best scores with under- and oversampling for each deep generative model and sampling strategy pair. Note that the rejection sampling can “timeout” if it fails to obtain the desired samples after 10,000 iterations.

Table 6.5 Classification experiments for “Credit card fraud”.

Part I: Classifier		
IR	**Train f1**	**Test f1**
0.001	0.904 ± 0.005	0.814 ± 0.046

Part II: Undersampling → Classifier		
USR	**Train f1**	**Test f1**
0.007	0.861 ± 0.009	0.813 ± 0.057

Part III: Undersampling → SMOTE Oversampling → Classifier				
Oversampling	**USR**	**OSR**	**Train f1**	**Test f1**
BorderlineSMOTE	0.004	0.005	0.877 ± 0.010	0.811 ± 0.055
RandomOverSampler	0.002	0.007	0.895 ± 0.006	0.822 ± 0.052
SMOTE	0.002	0.007	0.891 ± 0.006	0.831 ± 0.043
SVMSMOTE	0.002	0.003	0.899 ± 0.006	0.816 ± 0.062

Part IV: Undersampling → DGM Oversampling → Classifier					
DGM	**Sampling**	**USR**	**OSR**	**Train f1**	**Test f1**
vae	Minority	0.002	0.009	0.905 ± 0.006	0.820 ± 0.039
arae	Minority	0.004	0.005	0.873 ± 0.014	0.814 ± 0.052
medgan	Minority	0.002	0.010	0.896 ± 0.006	0.822 ± 0.042
gan	Minority	0.006	0.008	0.860 ± 0.013	0.820 ± 0.069
vae	Conditional	0.006	0.009	0.871 ± 0.008	0.817 ± 0.062
arae	Conditional	0.002	0.007	0.902 ± 0.004	0.816 ± 0.048
medgan	Conditional	0.003	0.007	0.886 ± 0.007	0.810 ± 0.056
gan	Conditional	0.004	0.010	0.877 ± 0.011	0.815 ± 0.051
vae	Rejection	*Timeout*			
arae	Rejection	*Timeout*			
medgan	Rejection	*Timeout*			
gan	Rejection	*Timeout*			

We start analyzing the results shown in Table 6.5 for the dataset “Credit card fraud”. Given that the dataset contains no categorical or binary features, we do not implement the multi-variable models for this scenario. The class imbalance is very high, and our experiments involving large changes to the imbalance ratio after under- and oversampling resulted in a considerable deterioration of the classification scores. Small changes on the imbalance ratio after undersampling do not offer classification improvements, but some small oversampling cases seem useful. In our experiments with this dataset, SMOTE presents the best classification score, while most of the deep generative models offer a slight improvement over the classification without oversampling. Experiments with SMOTE and “minority” GAN

Table 6.6 Classification experiments for “Adult”.

Part I: Classifier

IR	Train f1	Test f1
0.33	0.743 ± 0.002	0.716 ± 0.005

Part II: Undersampling → Classifier

USR	Train f1	Test f1
0.50	0.756 ± 0.002	0.731 ± 0.010

Part III: Undersampling → SMOTE Oversampling → Classifier

Oversampling	USR	OSR	Train f1	Test f1
ADASYN	0.4	0.6	0.749 ± 0.001	0.727 ± 0.010
BorderlineSMOTE	0.5	0.6	0.751 ± 0.001	0.730 ± 0.008
KMeansSMOTE	0.5	0.6	0.752 ± 0.002	0.731 ± 0.009
RandomOverSampler	0.5	0.6	0.756 ± 0.002	0.732 ± 0.007
SMOTE	0.5	0.6	0.752 ± 0.002	0.732 ± 0.008
SMOTENC	0.6	0.7	0.662 ± 0.001	0.642 ± 0.010
SVMSMOTE	0.5	0.6	0.750 ± 0.001	0.730 ± 0.009

Part IV: Undersampling → DGM Oversampling → Classifier

DGM	Sampling	USR	OSR	Train f1	Test f1
vae	Minority	0.5	1.0	0.756 ± 0.001	0.732 ± 0.010
mv-vae	Minority	0.5	1.0	0.755 ± 0.002	0.733 ± 0.010
arae	Minority	0.5	1.0	0.755 ± 0.001	0.733 ± 0.010
mv-arae	Minority	0.5	1.0	0.752 ± 0.001	0.732 ± 0.009
medgan	Minority	0.5	0.6	0.755 ± 0.002	0.732 ± 0.007
mv-medgan	Minority	0.6	0.8	0.752 ± 0.001	0.733 ± 0.010
gan	Minority	0.6	0.7	0.754 ± 0.001	0.734 ± 0.010
mv-wgan	Minority	0.5	1.0	0.752 ± 0.002	0.734 ± 0.008
mv-wgan-gp	Minority	0.6	0.7	0.754 ± 0.001	0.731 ± 0.009
vae	Conditional	0.6	1.0	0.754 ± 0.002	0.732 ± 0.008
mv-vae	Conditional	0.5	0.6	0.755 ± 0.001	0.733 ± 0.007
arae	Conditional	0.5	0.9	0.755 ± 0.002	0.734 ± 0.009
mv-arae	Conditional	0.6	0.9	0.752 ± 0.002	0.732 ± 0.010
medgan	Conditional	0.5	1.0	0.754 ± 0.002	0.733 ± 0.008
mv-medgan	Conditional	0.5	0.8	0.753 ± 0.002	0.732 ± 0.008
gan	Conditional	0.6	0.8	0.755 ± 0.002	0.733 ± 0.011
mv-wgan	Conditional	0.5	0.6	0.752 ± 0.002	0.733 ± 0.005
mv-wgan-gp	Conditional	0.6	0.8	0.752 ± 0.003	0.733 ± 0.008
vae	Rejection	0.5	1.0	0.756 ± 0.002	0.732 ± 0.008
mv-vae	Rejection	0.7	1.0	0.751 ± 0.002	0.732 ± 0.009
arae	Rejection	0.5	1.0	0.755 ± 0.002	0.733 ± 0.007
mv-arae	Rejection	0.6	0.9	0.753 ± 0.002	0.732 ± 0.011
medgan	Rejection	0.6	0.9	0.753 ± 0.002	0.733 ± 0.010
mv-medgan	Rejection	0.5	1.0	0.753 ± 0.001	0.732 ± 0.008
gan	Rejection	*Timeout*			
mv-wgan	Rejection	0.5	0.7	0.754 ± 0.002	0.732 ± 0.007
mv-wgan-gp	Rejection	0.5	0.9	0.754 ± 0.001	0.732 ± 0.008

were carried out with the same dataset in [39], where both models presented very close results. Besides the scores for SMOTE, our results are comparable to this study. We believe that the

Table 6.7 Classification experiments for “Default of credit card clients”.

Part I: Classifier

IR	Train f1	Test f1
0.28	0.479 ± 0.006	0.457 ± 0.036

Part II: Undersampling → Classifier

USR	Train f1	Test f1
0.80	0.552 ± 0.004	0.534 ± 0.031

Part III: Undersampling → SMOTE Oversampling → Classifier

Oversampling	USR	OSR	Train f1	Test f1
ADASYN	0.8	1.0	0.552 ± 0.004	0.537 ± 0.028
BorderlineSMOTE	0.6	0.7	0.550 ± 0.003	0.535 ± 0.027
KMeansSMOTE	0.7	1.0	0.549 ± 0.004	0.537 ± 0.029
RandomOverSampler	0.8	0.9	0.553 ± 0.005	0.538 ± 0.031
SMOTE	0.6	0.8	0.550 ± 0.004	0.535 ± 0.031
SMOTENC	0.9	1.0	0.488 ± 0.003	0.467 ± 0.029
SVMSMOTE	0.8	0.9	0.549 ± 0.004	0.534 ± 0.029

Part IV: Undersampling → DGM Oversampling → Classifier

DGM	Sampling	USR	OSR	Train f1	Test f1
vae	Minority	0.8	0.9	0.550 ± 0.004	0.535 ± 0.030
mv-vae	Minority	0.8	0.9	0.550 ± 0.004	0.533 ± 0.031
arae	Minority	0.7	0.8	0.546 ± 0.004	0.533 ± 0.030
mv-arae	Minority	0.8	0.9	0.548 ± 0.005	0.535 ± 0.030
medgan	Minority	0.8	0.9	0.546 ± 0.005	0.534 ± 0.032
mv-medgan	Minority	0.7	0.8	0.543 ± 0.003	0.534 ± 0.032
gan	Minority	0.8	0.9	0.549 ± 0.004	0.535 ± 0.029
mv-wgan	Minority	0.8	0.9	0.545 ± 0.004	0.534 ± 0.030
mv-wgan-gp	Minority	0.8	0.9	0.548 ± 0.004	0.534 ± 0.031
vae	Conditional	0.7	0.9	0.539 ± 0.004	0.531 ± 0.030
mv-vae	Conditional	0.7	0.8	0.546 ± 0.005	0.532 ± 0.032
arae	Conditional	0.8	0.9	0.549 ± 0.006	0.535 ± 0.030
mv-arae	Conditional	0.8	0.9	0.547 ± 0.004	0.533 ± 0.030
medgan	Conditional	0.8	0.9	0.548 ± 0.003	0.535 ± 0.030
mv-medgan	Conditional	0.8	0.9	0.546 ± 0.004	0.535 ± 0.032
gan	Conditional	0.8	0.9	0.550 ± 0.003	0.535 ± 0.029
mv-wgan	Conditional	0.8	0.9	0.546 ± 0.004	0.536 ± 0.032
mv-wgan-gp	Conditional	0.9	1.0	0.545 ± 0.006	0.534 ± 0.028
vae	Rejection	0.7	0.8	0.543 ± 0.004	0.532 ± 0.031
mv-vae	Rejection	0.7	0.8	0.545 ± 0.004	0.534 ± 0.031
arae	Rejection	0.9	1.0	0.546 ± 0.004	0.532 ± 0.032
mv-arae	Rejection	0.8	0.9	0.547 ± 0.003	0.534 ± 0.032
medgan	Rejection	*Timeout*			
mv-medgan	Rejection	*Timeout*			
gan	Rejection	*Timeout*			
mv-wgan	Rejection	0.8	0.9	0.547 ± 0.004	0.535 ± 0.032
mv-wgan-gp	Rejection	0.9	1.0	0.547 ± 0.004	0.535 ± 0.033

small discrepancies are related to the stochastic nature of the experiments. Additionally, we can see that the classification tends to overfit every case, which we anticipated considering the

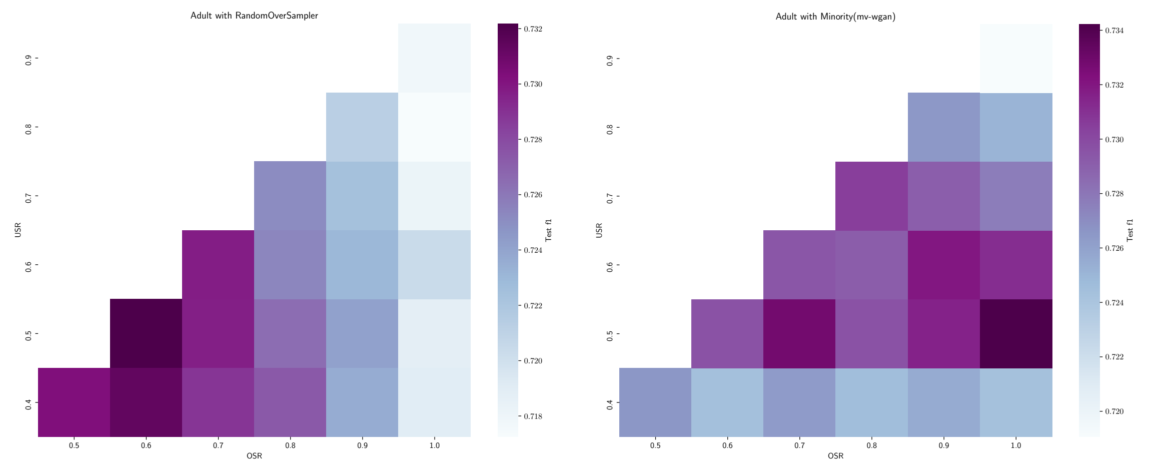


Fig. 6.3 Two examples of test f1 score versus USR and OSR for “Adult”.

small number of positive samples. Finally, note that all the models implementing rejection sampling reached a timeout. This phenomenon is reasonable for this dataset, given that the probability of drawing a sample from the minority class is very low.

Now we compare all the results from Table 6.6 and Table 6.7. Regarding the classification baseline without oversampling or undersampling, we can see that both problems are quite challenging since the mean test f1 score is considerably far from 1. Furthermore, adding only random undersampling to the pipeline shows some improvement in Table 6.6 but presents a considerable improvement in Table 6.7. Nevertheless, most of the oversampling techniques (both with and without deep generative models) only show a slight improvement on the third decimal of the mean test f1 score on top of the undersampling. The only oversampling algorithm that seems to worsen the classification quality is SMOTE-NC. Note also that in Table 6.7 the RandomOverSampler —which is the most straightforward oversampling technique— performs slightly better than the rest.

In Fig. 6.3, we present two example distributions of the f1 score across different ratios. We can see that there is some difference between the oversampling with and without deep generative models. The RandomOverSampler deteriorates progressively by incrementing the oversampling ratio, while the oversampling using a Multi-Variable WGAN does the opposite. However, note that the color bar scale (representing the mean test f1 score) is small.

In Fig. 6.4, we intend to explore some of the differences between sampling techniques visually. We present three alternatives: two feature space transformations into two dimensions based on PCA and t-SNE [93], and Self Organizing Maps (SOM) [67]. Scatter plots display the transformations where negative, positive, or synthetic samples present different colors. These plots provide us with the tools to examine how close to each other are samples from different types. On the other side, the SOM assigns samples to different cells on a grid,

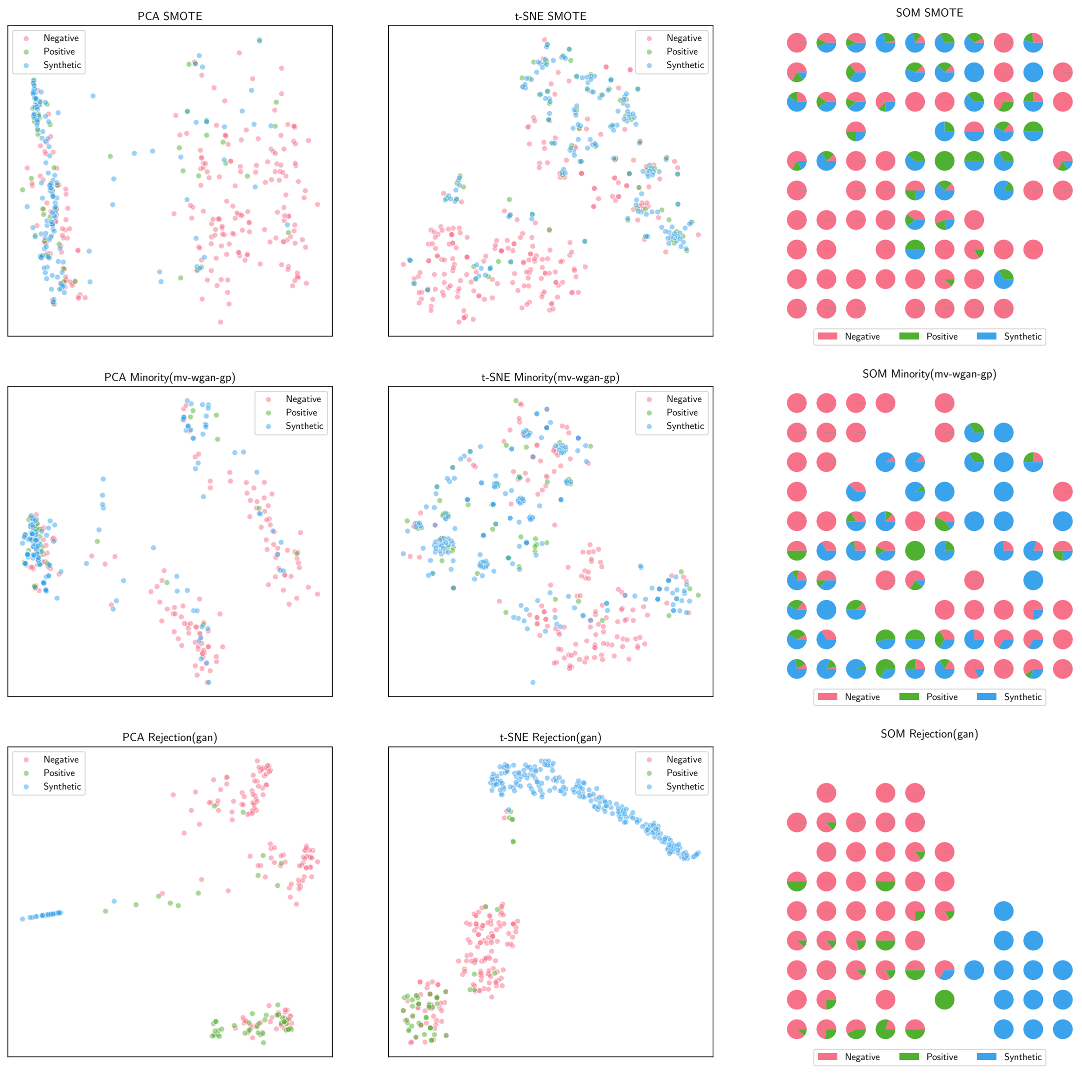


Fig. 6.4 PCA, t-SNE, and SOM examples for “Adult” with oversampling.

comparing each sample type’s frequency with a pie chart. Given that these methods do not scale well with the number of samples, we only visualize a subset in each plot. We are losing information by dropping samples and dimensions simultaneously, but these approximations can still explain the data structure. In the first two rows, we show SMOTE and one of the DGM models, and in both cases, we can see that synthetic samples tend to be grouped with positive cases as desired. Also, many of the negative classes are far away from the other two types of samples. Furthermore, in the third row, we show one example of a DGM model that generates samples that are very separated from the rest of the sample types. Nevertheless, regardless of these two different situations, we see in the classification results of the corresponding Table 6.6 that all oversampling techniques achieve practically the same performance. This finding could mean that the classification boundaries computed by XGBoost do not change much, neither by adding synthetic samples resembling positive samples nor with synthetic samples that fill empty portions of the feature space.

6.4 Discussion and Conclusion

Our experiments show several trends that persist across different generative methods and datasets: First, undersampling the majority class without any oversampling improves the classifier. Adding oversampling via a simple baseline such as SMOTE only leads to marginal improvements. Second, all generative models provide very close results to the SMOTE baselines. Finally, the sampling strategy does not substantially impact the results' quality, but rejection sampling is the slowest approach. It seems that we need to contextualize the performance gain given by deep generative models for oversampling. They possess a considerably more complicated setup and longer training time than the best-performing baseline, a simple random under- and oversampling approach.

It is noteworthy that generative models require different under- and oversampling settings to archive the best performance. However, regardless of the method used, the absolute improvement on the classification metric (F1 score) is often minimal. This phenomenon occurs despite results showing that in a ranking of methods, deep generative methods' improvement is statistically significantly better [35]. Even if the f1 score is prevalent in this domain, we could argue that it is not fair to compare results with this metric between experiments with different imbalance ratios. We also computed the area under the Precision-Recall curve for all the experiments, obtaining similar results. There is not a clear alternative in the literature, which implies a possible research opportunity.

Regarding the sampling strategies, it is reasonable to expect a lousy sample quality when training deep learning models with a small amount of data from the minority class. This situation directly affects the minority and the rejection strategies, but it is even worse for the latter because many draws are needed to obtain the desired output. On the other side, it is not very clear for the conditional strategy if learning the majority class's distribution would help define the distribution of the minority class better. Finally, besides the samples' quality, it is unclear whether powerful and complex machine learning algorithms like XGBoost can benefit by injecting synthetic samples into the minority class. There might be a combination of simpler models and oversampling methods that could achieve better results, but finding a useful combination might require many experiments. Brute forcing through combinations of tools, models, or techniques should not be the way of practicing science, but there is a lack of general understanding about what these generative models produce.

Chapter 7

Conclusions

7.1 Summary

Throughout this dissertation, we compared existing studies involving machine learning solutions to detect and analyze suspicious transactions. We synthesized methodologies for tackling different use cases in an organized manner and assessed the applicability of deep generative models for enhancing existing solutions.

In Chapter 2, we started with a literature review on machine learning methods to deal with the detection of fraud, money laundering, and different kinds of suspicious behavior related to financial activities. Most importantly, we identified several common challenges and flaws in the existing solutions:

- lack of labels, ground truth, and proper validation;
- highly imbalanced classes and high false positives;
- detection patterns based on complete information when in reality only partial information is available;
- studies based on private data with under specified distributions, pre-processing and feature engineering;
- studies based on synthetic data with strong baseless assumptions, simplistic or small scenarios and under specified distributions;
- and weak validation practices.

Additionally, we summarized and compared the different approaches involving deep generative models applied to tabular data or related types. We found a lack of benchmarks, standard baselines, and evaluation metrics like in other domains. Moreover, we described some factors that affect the reproducibility of existing studies:

- misidentified datasets (mostly from the UCI repository);
- under specified pre-processing practices;
- and obscure hyperparameter selection.

In Chapter 3, we presented a methodology to analyze financial transactions from different sources using unsupervised learning methods. We presented two case studies: one involving private transactions from a money remittance network provided by our industrial partner, and the other involving open transactions from the Ripple platform. In the first use case, we were able to validate our findings with experts from the private company. In particular, our tool was able to spot exciting cases in addition to the suspicious samples that traditional rule-based systems could detect. In the second use case, we validated our findings based on information from community forums. We detected not only expected anomalies but also cases that were not known to us beforehand.

In Chapter 4, we presented a step-by-step methodology to obtain, process, and analyze Ethereum contracts for the task of honeypot detection. We showed how to contrast assumptions and hypotheses about honeypot behavior against real data and derived features for supervised machine learning models. Classification models proved to generalize well, even when simulating that honeypots belonged to new techniques. Most importantly, our approach detected honeypots from two new techniques, which would be impossible to achieve using byte code analysis without manually crafting new detection rules.

In Chapter 5 we analyzed the impact of tabular data on deep learning models, a type of structure that is very common in fraud detection tasks as shown in Chapters 2, 3, and 4. We extended state-of-the-art deep generative models for tabular data with our multi-output architecture. We adapted an existing metric to capture the variable correlations better when using one-hot-encoding. Compared to the unmodified models, all the enhanced approaches improved the performance across all datasets, even though we could not identify an exact best model. The performance improvement came at the cost of requiring to add extra information.

Finally, in Chapter 6 we extended the work from Chapter 5 by applying the presented models to enhance classification tasks with imbalanced classes, the kind of task that is common for supervised fraud detection scenarios like the one presented in Chapter 4. Additionally, we introduced the multi-input architecture to expand models alongside our previously proposed multi-output architecture. Our experiments showed several trends that persist across different generative methods and datasets. First, undersampling the majority class without any oversampling improves the classifier. Second, adding oversampling via simple baseline techniques only leads to marginal improvements. Nevertheless, all generative models perform better than baselines. Finally, the sampling strategy does not seem to impact the quality of the results substantially, but it affects the execution time.

7.2 Contributions

Besides the particular findings on each presented use case, this thesis provides the means to address the many challenges for the general fraud detection domain. We proposed experimental protocols to test the generalization capabilities of supervised models when labels are present. For scenarios without ground truth, we proposed unsupervised alternatives using an ensemble of anomaly detection models to obtain robust anomaly rankings and several manual examination techniques for a robust validation. Additionally, we showed that both supervised and unsupervised scenarios could benefit from similar sets of features: aggregated values related to recency, frequency, and monetary aspects of the transactions, as well as metrics derived from the transaction graph and derived from sequences of discrete events triggered by the transactions. Most importantly, this thesis provides a compilation of machine learning techniques related to fraud detection and provides rigorous pre-processing and implementation details usually omitted or poorly defined in the literature.

We further compiled studies related to the application of deep learning on tabular data, and in particular, deep generative models. Our analysis can provide guidance to data science practitioners or statisticians who are looking to expand their sets of tools. We proposed several modern algorithms for generating synthetic data to avoid using arbitrary random distributions on simulations. Additionally, we presented several powerful models for oversampling data in the presence of class imbalance. Finally, we contributed to the deep learning community by presenting the multi-variable architecture, proposing new data generation evaluation techniques, and designing a large-scale evaluation of models defining a transparent and fair experimental protocol employing comparable model capacities.

7.3 Limitations

Our proposed multi-variable architecture requires a detailed description of each variable to work. However, in some cases, this information may not be available (e.g., when data is anonymized or obfuscated in some way). Furthermore, generative models require different under- and oversampling settings to archive the best performance. Nevertheless, regardless of the method used, the absolute improvement on the classification metric is often minimal.

It seems that we need to contextualize the use of deep generative models for oversampling. They possess a considerably more complicated set-up and longer training time than the best-performing baseline, a simple random under- and oversampling approach. Consider that traditional oversampling methods have been around for more than a decade, pushing the performance of imbalanced classification tasks. However, novel deep generative models can achieve comparable results, even when conceived for a different purpose.

7.4 Future Work

One way of extending this dissertation's work would be to focus on detecting unexpected changes in user behavior and extending the analysis methods to fit into *streaming and big data environments*. For example, unsupervised machine learning models could be re-trained and replaced either periodically or when the number of anomalies exceeds a certain predefined threshold (indicating the presence of a *concept drift*). Another alternative would be to implement *online learning*, where the models are updated directly with each predicted data point. Note that not only *when to update the models* is a crucial point to study, but also *how much historical data is needed to train the models*. In the case of supervised models, the adaptation is not straightforward, given that the presence of new samples does not come alongside new labels (or at least, not at the same time, and possibly not for free). One possible solution would be to experiment with *semi-supervised* alternatives, a domain in which deep generative models also have potential.

Counts and frequencies of discrete events have been used as features for the different use cases throughout this dissertation. We experimented with the possibility of capturing a more extended pattern of discrete events, represented as traces or strings of symbols. However, we discovered that the sequences of events related to the users or accounts we studied were not long enough to present exciting dependencies. Therefore, working with n-grams instead of only uni-grams, using string similarities for clustering, and employing Markov models or recurrent neural networks for detection remain ideas for future use cases.

In the domain of deep generative models, we have ongoing work in missing data imputation, compiling, comparing, and extending state-of-the-art models, and proposing novel iterative imputation methods. We are also interested in applying the power of deep generative models to different types of transfer learning. We could achieve this either by sharing synthetic data or trained generative models preserving privacy constraints. Finally, we are looking forward to applying the novel transformer architectures on tabular data.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467.*

[2] Aggarwal, C. C. (2016). *Outlier analysis*. Springer, 2nd edition.

[3] Alexandre, C. and Balsa, J. (2015a). Client profiling for an anti-money laundering system. *arXiv preprint arXiv:1510.00878.*

[4] Alexandre, C. and Balsa, J. (2015b). A multiagent based approach to money laundering detection and prevention. In *ICAART (1)*, pages 230–235.

[5] Alexandre, C. and Balsa, J. (2016). Integrating client profiling in an anti-money laundering multi-agent based system. In *New Advances in Information Systems and Technologies*, pages 931–941. Springer.

[6] Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein gan. *arXiv preprint arXiv:1701.07875.*

[7] Armknecht, F., Karame, G. O., Mandal, A., Youssef, F., and Zenner, E. (2015). Ripple: Overview and outlook. In *International Conference on Trust and Trustworthy Computing*, pages 163–180. Springer.

[8] Atzei, N., Bartoletti, M., and Cimoli, T. (2017). A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186. Springer-Verlag New York, Inc.

[9] Baesens, B., Van Vlasselaer, V., and Verbeke, W. (2015). *Fraud analytics using descriptive, predictive, and social network techniques: a guide to data science for fraud detection.* John Wiley & Sons.

[10] Bartoletti, M., Carta, S., Cimoli, T., and Saia, R. (2020). Dissecting ponzi schemes on ethereum: identification, analysis, and impact. *Future Generation Computer Systems*, 102:259–277.

[11] Bershtein, L. S. and Tselykh, A. (2013). A clique-based method for mining fuzzy graph patterns in anti-money laundering systems. In *Proceedings of the 6th International Conference on Security of Information and Networks*, pages 384–387.

[12] Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

[13] Bolton, R. J. and Hand, D. J. (2002). Statistical fraud detection: A review. *Statistical science*, pages 235–249.

[14] Bolton, R. J., Hand, D. J., et al. (2001). Unsupervised profiling methods for fraud detection. *Credit scoring and credit control VII*, pages 235–255.

[15] Borji, A. (2019). Pros and cons of gan evaluation measures. *Computer Vision and Image Understanding*, 179:41–65.

[16] Bowman, S. R., Vilnis, L., Vinyals, O., Dai, A. M., Jozefowicz, R., and Bengio, S. (2015). Generating sentences from a continuous space. *arXiv preprint arXiv:1511.06349*.

[17] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

[18] Burda, Y., Grosse, R., and Salakhutdinov, R. (2015). Importance weighted autoencoders. *arXiv preprint arXiv:1509.00519*.

[19] Cao, D. K. and Do, P. (2012). Applying data mining in money laundering detection for the vietnamese banking industry. In *Asian Conference on Intelligent Information and Database Systems*, pages 207–216. Springer.

[20] Cao, Y., Li, Y., Coleman, S., Belatreche, A., and McGinnity, T. M. (2014a). Adaptive hidden markov model with anomaly states for price manipulation detection. *IEEE transactions on neural networks and learning systems*, 26(2):318–330.

[21] Cao, Y., Li, Y., Coleman, S., Belatreche, A., and McGinnity, T. M. (2014b). Detecting price manipulation in the financial market. In *2014 IEEE Conference on Computational Intelligence for Financial Engineering & Economics (CIFEr)*, pages 77–84. IEEE.

[22] Cao, Y., Li, Y., Coleman, S., Belatreche, A., and McGinnity, T. M. (2015). Detecting wash trade in financial market using digraphs and dynamic programming. *IEEE transactions on neural networks and learning systems*, 27(11):2351–2363.

[23] Chang, W.-H. and Chang, J.-S. (2010). Using clustering techniques to analyze fraudulent behavior changes in online auctions. In *2010 International Conference on Networking and Information Technology*, pages 34–38. IEEE.

[24] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357.

[25] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794.

[26] Chen, W., Zheng, Z., Cui, J., Ngai, E., Zheng, P., and Zhou, Y. (2018). Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In *Proceedings of the 2018 World Wide Web Conference*, pages 1409–1418.

[27] Chen, Z., Nazir, A., Teoh, E. N., Karupiah, E. K., et al. (2014). Exploration of the effectiveness of expectation maximization algorithm for suspicious transaction detection in anti-money laundering. In *2014 IEEE Conference on Open Systems (ICOS)*, pages 145–149. IEEE.

[28] Chiang, A. and Yeh, Y.-R. (2015). Anomaly detection ensembles: In defense of the average. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, volume 3, pages 207–210. IEEE.

[29] Choi, E., Biswal, S., Malin, B., Duke, J., Stewart, W. F., and Sun, J. (2017). Generating multi-label discrete patient records using generative adversarial networks. *arXiv preprint arXiv:1703.06490*.

[30] Colladon, A. F. and Remondi, E. (2017). Using social network analysis to prevent money laundering. *Expert Systems with Applications*, 67:49–58.

[31] Dal Pozzolo, A., Caelen, O., Johnson, R. A., and Bontempi, G. (2015). Calibrating probability with undersampling for unbalanced classification. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 159–166. IEEE.

[32] Delort, J.-Y., Arunasalam, B., Milosavljevic, M., and Leung, H. (2009). The impact of manipulation in internet stock message boards. *International Journal of Banking and Finance, Forthcoming*.

[33] Diaz, D., Theodoulidis, B., and Sampaio, P. (2011). Analysis of stock market manipulations using knowledge discovery techniques applied to intraday trade prices. *Expert Systems with Applications*, 38(10):12757–12771.

[34] Donoho, S. (2004). Early detection of insider trading in option markets. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 420–429.

[35] Douzas, G. and Bacao, F. (2018). Effective data generation for imbalanced learning using conditional generative adversarial networks. *Expert Systems with applications*, 91:464–471.

[36] Dreżewski, R., Sepielak, J., and Filipkowski, W. (2015). The application of social network analysis algorithms in a system supporting money laundering detection. *Information Sciences*, 295:18–32.

[37] Dua, D. and Graff, C. (2017). Uci machine learning repository.

[38] Ferreira Torres, C., Steichen, M., et al. (2019). The art of the scam: Demystifying honeypots in ethereum smart contracts. In *USENIX Security Symposium, Santa Clara, 14-16 August 2019*.

[39] Fiore, U., De Santis, A., Perla, F., Zanetti, P., and Palmieri, F. (2019). Using generative adversarial networks for improving classification effectiveness in credit card fraud detection. *Information Sciences*, 479:448–455.

[40] Franke, M., Hoser, B., and Schröder, J. (2008). On the analysis of irregular stock market trading behavior. In *Data Analysis, Machine Learning and Applications*, pages 355–362. Springer.

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

[42] Gao, S. and Xu, D. (2009). Conceptual modeling and development of an intelligent agent-assisted decision support system for anti-money laundering. *Expert Systems with Applications*, 36(2):1493–1504.

[43] Gao, S. and Xu, D. (2010). Real-time exception management decision model (rtemdm): Applications in intelligent agent-assisted decision support in logistics and anti-money laundering domains. In *2010 43rd Hawaii International Conference on System Sciences*, pages 1–10. IEEE.

[44] Gao, S., Xu, D., Wang, H., and Wang, Y. (2006). Intelligent anti-money laundering system. In *2006 IEEE International Conference on Service Operations and Logistics, and Informatics*, pages 851–856. IEEE.

[45] Gao, Z. (2009). Application of cluster-based local outlier factor algorithm in anti-money laundering. In *2009 International Conference on Management and Service Science*, pages 1–4. IEEE.

[46] Gao, Z. and Ye, M. (2007). A framework for data mining-based anti-money laundering research. *Journal of Money Laundering Control*.

[47] Golmohammadi, K., Zaiane, O. R., and Díaz, D. (2014). Detecting stock market manipulation using supervised learning algorithms. In *2014 International Conference on Data Science and Advanced Analytics (DSAA)*, pages 435–441. IEEE.

[48] Gondara, L. and Wang, K. (2018). Mida: Multiple imputation using denoising autoencoders. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 260–272. Springer.

[49] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

[50] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680.

[51] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. C. (2017). Improved training of wasserstein gans. In *Advances in neural information processing systems*, pages 5767–5777.

[52] Helmy, T. H., Zaki, M., Salah, T., and Badran, K. (2016). Design of a monitor for detecting money laundering and terrorist financing. *Journal of Theoretical and Applied Information Technology*, 85(3):425.

[53] Herzog, T. N., Scheuren, F. J., and Winkler, W. E. (2007). *Data quality and record linkage techniques*. Springer Science & Business Media.

[54] Hilas, C. S. and Sahalos, J. N. (2005). User profiling for fraud detection in telecommunication networks. In *5th International conference on technology and automation*, pages 382–387.

[55] Hwang, U., Jung, D., and Yoon, S. (2019). Hexagan: Generative adversarial nets for real world classification. *arXiv preprint arXiv:1902.09913*.

[56] Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134.

[57] Jang, E., Gu, S., and Poole, B. (2016). Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*.

[58] Jank, W. and Shmueli, G. (2003). Dynamic profiling of online auctions using curve clustering. *submitted for publication*.

[59] Jurafsky, D. and Martin, J. H. (2000). *Speech and Language Processing: An introduction to natural language processing, computational linguistics, and speech recognition*. Prentice Hall.

[60] Kannan, S. and Somasundaram, K. (2017). Autoregressive-based outlier algorithm to detect money laundering activities. *Journal of Money Laundering Control*.

[61] Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., and Aila, T. (2020). Analyzing and improving the image quality of stylegan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8110–8119.

[62] Keyan, L. and Tingting, Y. (2011). An improved support-vector network model for anti-money laundering. In *2011 Fifth International Conference on Management of e-Commerce and e-Government*, pages 193–196. IEEE.

[63] Kharote, M. and Kshirsagar, V. (2014). Data mining model for money laundering detection in financial domain. *International Journal of Computer Applications*, 85(16).

[64] Kim, T., Cha, M., Kim, H., Lee, J. K., and Kim, J. (2017). Learning to discover cross-domain relations with generative adversarial networks. *arXiv preprint arXiv:1703.05192*.

[65] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[66] Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

[67] Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480.

[68] Krupp, J. and Rossow, C. (2018). teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1317–1333.

[69] Kusner, M. J. and Hernández-Lobato, J. M. (2016). Gans for sequences of discrete elements with the gumbel-softmax distribution. *arXiv preprint arXiv:1611.04051*.

[70] Le Khac, N. A. and Kechadi, M.-T. (2010). Application of data mining for anti-money laundering detection: A case study. In *2010 IEEE International Conference on Data Mining Workshops*, pages 577–584. IEEE.

[71] Le-Khac, N.-A., Markos, S., and Kechadi, M.-T. (2009a). A heuristics approach for fast detecting suspicious money laundering cases in an investment bank. *World Academy of Science, Engineering and Technology*, 60(2009):76–80.

[72] Le-Khac, N.-A., Markos, S., and Kechadi, M.-T. (2009b). Towards a new data mining-based approach for anti-money laundering in an international investment bank. In *International Conference on Digital Forensics and Cyber Crime*, pages 77–84. Springer.

[73] Le Khac, N. A., Markos, S., and Kechadi, M.-T. (2010). A data mining-based solution for detecting suspicious money laundering cases in an investment bank. In *2010 Second International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 235–240. IEEE.

[74] Le Khac, N. A., Markos, S., O'Neill, M., Brabazon, A., and Kechadi, M.-T. (2009). An investigation into data mining approaches for anti money laundering. In *Proceedings of International Conference on Computer Engineering and Applications (ICCEA 2009).*

[75] Le-Khac, N.-A., Markos, S., O'Neill, M., Brabazon, A., and Kechadi, T. (2016). An efficient search tool for an anti-money laundering application of an multi-national bank's dataset. *arXiv preprint arXiv:1609.02031.*

[76] Leangarun, T., Tangamchit, P., and Thajchayapong, S. (2016). Stock price manipulation detection based on mathematical models. *International Journal of Trade, Economics and Finance*, 7(3):81–88.

[77] Lemaître, G., Nogueira, F., and Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *The Journal of Machine Learning Research*, 18(1):559–563.

[78] Li, Z., Xiong, H., and Liu, Y. (2010). Detecting blackholes and volcanoes in directed networks. *arXiv preprint arXiv:1005.2179.*

[79] Liaw, A., Wiener, M., et al. (2002). Classification and regression by randomforest. *R news*, 2(3):18–22.

[80] Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2008a). Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE.

[81] Liu, R., Qian, X.-l., Mao, S., and Zhu, S.-z. (2011). Research on anti-money laundering based on core decision tree algorithm. In *2011 Chinese Control and Decision Conference (CCDC)*, pages 4322–4325. IEEE.

[82] Liu, X. and Zhang, P. (2007). An agent based anti-money laundering system architecture for financial supervision. In *2007 International Conference on Wireless Communications, Networking and Mobile Computing*, pages 5472–5475. IEEE.

[83] Liu, X. and Zhang, P. (2008). Research on constraints in anti-money laundering (aml) business process in china based on theory of constraints. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, pages 213–213. IEEE.

[84] Liu, X. and Zhang, P. (2010). A scan statistics based suspicious transactions detection model for anti-money laundering (aml) in financial institutions. In *2010 International Conference on Multimedia Communications*, pages 210–213. IEEE.

[85] Liu, X., Zhang, P., and Zeng, D. (2008b). Sequence matching for suspicious activity detection in anti-money laundering. In *International Conference on Intelligence and Security Informatics*, pages 50–61. Springer.

[86] Lopez-Rojas, E. A. and Axelsson, S. (2012a). Money laundering detection using synthetic data. In *Annual workshop of the Swedish Artificial Intelligence Society (SAIS)*. Linköping University Electronic Press, Linköpings universitet.

[87] Lopez-Rojas, E. A. and Axelsson, S. (2012b). Multi agent based simulation (mabs) of financial transactions for anti money laundering (aml). In *Nordic Conference on Secure IT Systems*. Blekinge Institute of Technology.

[88] Lopez-Rojas, E. A. and Axelsson, S. (2016). A review of computer simulation for fraud detection research in financial datasets. In *2016 Future Technologies Conference (FTC)*, pages 932–935. IEEE.

[89] Luca, D. and Mueller, B. (2019). The Ether Wars: Exploits, counter-exploits and honeypots on Ethereum. https://infocondb.org/con/def-con/def-con-27/the-ether-wars-exploits-counter-exploits-and-honeypots-on-ethereum.

[90] Lucic, M., Kurach, K., Michalski, M., Gelly, S., and Bousquet, O. (2018). Are gans created equal? a large-scale study. In *Advances in neural information processing systems*, pages 700–709.

[91] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA. ACM.

[92] Lv, L.-T., Ji, N., and Zhang, J.-L. (2008). A rbf neural network model for anti-money laundering. In *2008 International Conference on Wavelet Analysis and Pattern Recognition*, volume 1, pages 209–215. IEEE.

[93] Maaten, L. v. d. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605.

[94] Maddison, C. J., Mnih, A., and Teh, Y. W. (2016). The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*.

[95] Mann, H. B. and Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60.

[96] Mansinghka, V., Tibbetts, R., Baxter, J., Shafto, P., and Eaves, B. (2015). Bayesdb: A probabilistic programming system for querying the probable implications of data. *arXiv preprint arXiv:1512.05006*.

[97] Martínez-Miranda, E., McBurney, P., and Howard, M. J. (2016). Learning unfair trading: A market manipulation analysis from the reinforcement learning perspective. In *2016 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS)*, pages 103–109. IEEE.

[98] Mattei, P.-A. and Frellsen, J. (2018). Miwae: Deep generative modelling and imputation of incomplete data. *arXiv preprint arXiv:1812.02633*.

[99] MCA, G. K. and Prabakaran, M. (2014). An multi-variant relational model for money laundering identification using time series data set. *Int. J. Eng. Sci*, 3:43–47.

[100] McNally, S., Roche, J., and Caton, S. (2018). Predicting the price of bitcoin using machine learning. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 339–343. IEEE.

[101] Mehmet, M. and Wijesekera, D. (2013a). Data analytics to detect evolving money laundering. In *STIDS*, pages 71–78.

[102] Mehmet, M. and Wijesekera, D. (2013b). Using dynamic risk estimation & social network analysis to detect money laundering evolution. In *2013 IEEE International Conference on Technologies for Homeland Security (HST)*, pages 310–315. IEEE.

[103] Mirza, M. and Osindero, S. (2014). Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*.

[104] Mitchell, T. M. et al. (1997). *Machine learning*. McGraw Hill.

[105] Mottini, A., Lheritier, A., and Acuna-Agost, R. (2018). Airline passenger name record generation using generative adversarial networks. *arXiv preprint arXiv:1807.06657*.

[106] Musgrave, K., Belongie, S., and Lim, S.-N. (2020). A metric learning reality check. *arXiv preprint arXiv:2003.08505*.

[107] Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*.

[108] Nazabal, A., Olmos, P. M., Ghahramani, Z., and Valera, I. (2018). Handling incomplete heterogeneous data using vaes. *arXiv preprint arXiv:1807.03653*.

[109] Ng, W. W., Zeng, G., Zhang, J., Yeung, D. S., and Pedrycz, W. (2016). Dual autoencoders features for imbalance classification problem. *Pattern Recognition*, 60:875–889.

[110] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., and Hobor, A. (2018). Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663.

[111] Norvill, R., Pontiveros, B. B. F., State, R., Awan, I., and Cullen, A. (2017). Automated labeling of unknown contracts in ethereum. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6. IEEE.

[112] Öğüt, H., Doğanay, M. M., and Aktaş, R. (2009). Detecting stock-price manipulation in an emerging market: The case of turkey. *Expert Systems with Applications*, 36(9):11944–11949.

[113] Palshikar, G. K. and Apte, M. M. (2008). Collusion set detection using graph clustering. *Data mining and knowledge Discovery*, 16(2):135–164.

[114] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

[115] Paula, E. L., Ladeira, M., Carvalho, R. N., and Marzagao, T. (2016). Deep learning anomaly detection as support fraud investigation in brazilian exports and anti-money laundering. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 954–960. IEEE.

[116] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830.

[117] Petrov, S. (2017). Another parity wallet hack explained. https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c.

[118] Phua, C., Lee, V., Smith, K., and Gayler, R. (2010). A comprehensive survey of data mining-based fraud detection research. *arXiv preprint arXiv:1009.6119*.

[119] Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

[120] Rieke, R., Zhdanova, M., Repp, J., Giot, R., and Gaber, C. (2013). Fraud detection in mobile payments utilizing process behavior analysis. In *2013 International Conference on Availability, Reliability and Security*, pages 662–669. IEEE.

[121] Rohit, K. D. and Patel, D. B. (2015). Review on detection of suspicious transaction in anti-money laundering using data mining framework. *International Journal for Innovative Research in Science & Technology*, 1(8):129–133.

[122] Russel, S., Norvig, P., et al. (2013). *Artificial intelligence: a modern approach*. Pearson Education Limited.

[123] Salehi, A., Ghazanfari, M., and Fathian, M. (2017). Data mining techniques for anti money laundering. *International Journal of Applied Engineering Research*, 12(20):10084–10094.

[124] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016). Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242.

[125] Sanjuas, J. (2018). An analysis of a couple ethereum honeypot contracts. https://medium.com/coinmonks/an-analysis-of-a-couple-ethereum-honeypot-contracts-5c07c95b0a8d.

[126] Schmidt, A. B. (2011). *Financial markets and trading: an introduction to market microstructure and trading strategies*, volume 637. John Wiley & Sons.

[127] Schölkopf, B., Platt, J. C., Shawe-Taylor, J., Smola, A. J., and Williamson, R. C. (2001). Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471.

[128] Schott, P. A. (2006). *Reference guide to anti-money laundering and combating the financing of terrorism*. The World Bank.

[129] Sherbachev, A. (2018). Hacking the hackers: Honeypots on ethereum network. https://hackernoon.com/hacking-the-hackers-honeypots-on-ethereum-network-5baa35a13577.

[130] Siegel, D. (2016). Understanding the dao attack. https://www.coindesk.com/understanding-dao-hack-journalists/.

[131] Sudjianto, A., Nair, S., Yuan, M., Zhang, A., Kern, D., and Cela-Díaz, F. (2010). Statistical methods for fighting financial crimes. *Technometrics*, 52(1):5–19.

[132] Suresh, C., Reddy, K. T., and Sweta, N. (2016). A hybrid approach for detecting suspicious accounts in money laundering using data mining techniques. *International Journal of Information Technology and Computer Science (IJITCS)*, 8(5):37.

[133] Tamersoy, A., Xie, B., Lenkey, S. L., Routledge, B. R., Chau, D. H., and Navathe, S. B. (2013). Inside insider trading: Patterns & discoveries from a large scale exploratory analysis. In *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 797–804.

[134] Tang, J. and Yin, J. (2005). Developing an intelligent data discriminating system of anti-money laundering based on svm. In *2005 International conference on machine learning and cybernetics*, volume 6, pages 3453–3457. IEEE.

[135] Tann, A., Han, X. J., Gupta, S. S., and Ong, Y.-S. (2018). Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities. *arXiv preprint arXiv:1811.06632*.

[136] Theis, L., Oord, A. v. d., and Bethge, M. (2015). A note on the evaluation of generative models. *arXiv preprint arXiv:1511.01844*.

[137] Torres, C. F., Schütte, J., and State, R. (2018). Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 664–676, New York, NY, USA. ACM.

[138] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., and Vechev, M. (2018). Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82. ACM.

[139] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM.

[140] Wang, G., Zhang, X., Tang, S., Zheng, H., and Zhao, B. Y. (2016). Unsupervised clickstream clustering for user behavior analysis. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 225–236.

[141] Wang, J., Zhou, S., and Guan, J. (2012). Detecting potential collusive cliques in futures markets based on trading behaviors from real data. *Neurocomputing*, 92:44–53.

[142] Wang, S.-N. and Yang, J.-G. (2007). A money laundering risk evaluation method based on decision tree. In *2007 International Conference on Machine Learning and Cybernetics*, volume 1, pages 283–286. IEEE.

[143] Wang, X. and Dong, G. (2009). Research on money laundering detection based on improved minimum spanning tree clustering and its application. In *2009 Second international symposium on knowledge acquisition and modeling*, volume 2, pages 62–64. IEEE.

[144] Wang, Y., Xu, D., Wang, H., Ye, K., and Gao, S. (2007). Agent-oriented ontology for monitoring and detecting money laundering process. In *Proceedings of the 2nd international conference on Scalable information systems*, pages 1–4.

[145] Watkins, R. C., Reynolds, K. M., Demara, R., Georgiopoulos, M., Gonzalez, A., and Eaglin, R. (2003). Tracking dirty proceeds: exploring data mining technologies as tools to investigate money laundering. *Police Practice and Research*, 4(2):163–178.

[146] Westphal, C. (2008). *Data Mining for Intelligence, Fraud & Criminal Detection: Advanced Analytics & Information Sharing Technologies*. CRC Press.

[147] Witten, I. H. and Frank, E. (2002). Data mining: practical machine learning tools and techniques with java implementations. *Acm Sigmod Record*, 31(1):76–77.

[148] Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32.

[149] Xu, L., Skoularidou, M., Cuesta-Infante, A., and Veeramachaneni, K. (2019). Modeling tabular data using conditional gan. In *Advances in Neural Information Processing Systems*, pages 7333–7343.

[150] Xu, L. and Veeramachaneni, K. (2018). Synthesizing tabular data using generative adversarial networks. *arXiv preprint arXiv:1811.11264*.

[151] Xu, X. (2010). Sequential anomaly detection based on temporal-difference learning: Principles, models and case studies. *Applied Soft Computing*, 10(3):859–867.

[152] Yang, Q., Feng, B., and Song, P. (2007). Study on anti-money laundering service system of online payment based on union-bank mode. In *2007 International Conference on Wireless Communications, Networking and Mobile Computing*, pages 4991–4994. IEEE.

[153] Yoon, J., Jordon, J., and Van Der Schaar, M. (2018). Gain: Missing data imputation using generative adversarial nets. *arXiv preprint arXiv:1806.02920*.

[154] Yu, L., Zhang, W., Wang, J., and Yu, Y. (2017). Seqgan: Sequence generative adversarial nets with policy gradient. In *Thirty-first AAAI conference on artificial intelligence*.

[155] Yunkai, C., Quanwen, M., and Zhengding, L. (2006). Using link analysis technique with a modified shortest-path algorithm to fight money laundering. *Wuhan University Journal of Natural Sciences*, 11(5):1352–1356.

[156] Zaki, M., Theodoulidis, B., and Solis, D. D. (2011). A data mining approach for the analysis of "stock-touting" spam emails. *Journal of Manufacturing Technology Management*, 22(6):70–79.

[157] Zhai, J., Cao, Y., Yao, Y., Ding, X., and Li, Y. (2017). Computational intelligent hybrid model for detecting disruptive trading activity. *Decision Support Systems*, 93:26–41.

[158] Zhang, C.-w. and Wang, Y.-b. (2010). Research on application of distributed data mining in anti-money laundering monitoring system. In *2010 2nd International Conference on Advanced Computer Control*, volume 5, pages 133–135. IEEE.

[159] Zhao, J., Kim, Y., Zhang, K., Rush, A. M., and LeCun, Y. (2017). Adversarially regularized autoencoders. *arXiv preprint arXiv:1706.04223*.

[160] Zhdanova, M., Repp, J., Rieke, R., Gaber, C., and Hemery, B. (2014). No smurfs: Revealing fraud chains in mobile money transfers. In *2014 Ninth International Conference on Availability, Reliability and Security*, pages 11–20. IEEE.

[161] Zhou, Y., Wang, X., Zhang, J., Zhang, P., Liu, L., Jin, H., and Jin, H. (2017). Analyzing and detecting money-laundering accounts in online social networks. *IEEE Network*, 32(3):115–121.