POLITECNICO DI MILANO
School of Industrial and Information Engineering
Department of Electronics, Information and Bioengineering
Master of Science Degree in Computer Science and Engineering

Oversampling Techniques to Improve Fraud Detection

Supervisor: Giacomo Boracchi, Ph.D.

Master's Thesis by:
Vincenzo Visco, 893974

Academic Year 2018-2019

To my family

Contents

Abstract

We live in a world where almost everyone has access to internet, with the number of users growing every year; more and more of them are using online banking services actively every day, because these are easy and fast to use, enabling payments in few seconds. In this context there are a lot of individuals trying to steal money from people with different kinds of techniques. Fraud Detection has become a primary need for banks and financial institutions, that can prevent the losses up to billions of Euros every year, both for costumers and providers.

Aim of the thesis is to present a framework for fraud detection in the context of credit card transactions, with a specific focus on how oversampling can improve performance. The framework is able to address all the major problems coming from fraud detection: class imbalance, verification latency, and concept drift. In particular we focused on the problem of class imbalance, testing and comparing different oversampling techniques in order to balance the dataset, including a new technique developed by us, that is based on genetic algorithms. The dataset used is available on the Kaggle repository [30], that contains 284.807 transactions, of which 492 frauds, spanning on a period of 48 hours, characterized by 31 features, of which 28 of them were anonymized to preserve the privacy of the cardholders, while the remaining were *Time*, *Amount*, and *Class*.

The work was carried out during my internship at the company Technology Reply, where I learned to use new tools that allowed me to conduct this thesis. In our experiments we found out that in our settings, the proposed

oversampling solutions helped fraud detection, leading to better performance with respect to the baseline. The results are limited to the dataset we used, but they are promising and they should be tested on different ones.

Sommario

Viviamo in un mondo dove quasi tutti hanno accesso ad internet, con il numero di utenti in crescendo ogni anno; sempre più di loro usano attivamente servizi di online banking ogni giorno, per via della loro facilità e velocità di utilizzo, permettendo di effettuare pagamenti in pochi secondi. In questo contesto molti individui cercano di approfittarsene per rubare soldi con diversi tipi di tecniche. La Fraud Detection è diventata una priorità per banche e istituti finanziari, che grazie ad essa può prevenire perdite di miliardi di Euro ogni anno, sia per loro che per i loro clienti.
Lo scopo della tesi è lo sviluppo di un sistema per la fraud detection nel contesto delle transazioni relative alle carte di credito, con attenzione particolare a come le tecniche di sovracampionamento possono migliorare le prestazioni. Il sistema è capace di affrontare e risolvere i principali problemi relativi alla fraud detection: sbilanciamento delle classi, latenza di verifica, e non stazionarità dei dati. In particolare ci siamo concentrati sul problema dello sbilanciamento delle classi, testando e comparando diverse tecniche di sovracampionamento per bilanciare il dataset, compresa una nuova tecnica sviluppata da noi basata su algoritmi genetici. Il dataset che abbiamo usato, dispoibile sul sito di Kaggle [30], contiene 284807 transazioni che coprono un periodo di 48 ore e che sono caratterizzate da 31 attributi, di cui 28 anonimizzate per preservare la privacy degli utenti, mentre le restanti erano *Tempo*, *Ammontare della transazione*, e *Classe*.
Il lavoro è stato svolto durante il mio periodo di stage nell'azienda Technology Reply, dove sono venuto a consocenza e ho imparato ad usare nuovi

strumenti che mi hanno permesso di portare avanti questa tesi. Durante i nostri esperimenti abbiamo scoperto che per le nostre impostazioni il sovra-campionamento ha reso migliore la detezione di frodi, portando a prestazioni migliori rispetto all'algoritmo di base. I risultati sono limitati al dataset in nostro possesso, ma sono molto promettenti e dovrebbero essere testati su altri dataset.

Chapter 1

Introduction

We live in a world where almost everyone has access to internet, with the number of users growing every year; more and more of them are using online banking services actively every day, because they are easy and fast to use, allowing payments to be completed in few seconds. In this context there are a lot of individuals trying to steal money from people with different kinds of techniques. The huge amount of credit card transactions performed every day, both in person and online, makes the users exposed to these fraudsters. This kind of activities leads to loss of billions of Euros every year, both for banks and for users.

Analysis of fraudulent attacks can be performed manually by investigators, but this task is really hard since the strategies of fraudsters are becoming more and more sophisticated, because they learn from genuine activities and imitate these. For these reasons it is necessity for banks and financial institutions to develop automated systems to detect and to prevent frauds, in order to solve this problem. Classical fraud detection systems are based on if-then-else rules, but they are static and can be easily tricked by fraudsters, hence they are obsolete. The newest techniques in this field are based on machine learning algorithms, that have some benefits. Firstly, using machine learning makes the task automated, saving money and resources. Then, it can spot patterns that are invisible to humans, leveraging the huge amount of data at

our disposal, finding relationships between data that can make no sense at first sight. At last, this kind of applications can adapt easily to changes in behaviour without a lot of human intervention.

When machine learning is used for fraud detection, there are two main problems that have to be addressed. Concept drift, that is the behavioural changes of fraudsters, that can mislead the detector, and that have to be solved in order to have a reliable system, and class imbalance, that is when in a dataset we have more sample belonging to a class with respect to the others. In our case we have two classes, genuine and fraud, and the sample belonging to the fraud class are way outnumbered by those belonging to the genuine class. The last problem is a very delicate one in machine learning, because most of the algorithms perform at their best when the classes are balanced, and they get worse performance along with the rising of the imbalance.

The objective of the thesis is to develop a fraud detection system and to test and compare different oversampling techniques, in order to tackle the problem of class imbalance and to understand which technique is most suited for fraud detection's problems. To meet this objective we developed a framework for fraud detection based on machine learning, testing different classification and oversampling techniques, that is able to tackle the most critic limitations of this problem. For classification we tested random forest, extreme gradient boosting and neural networks; for oversampling we tested random oversampling, an oversampling algorithm based on generative adversarial networks, a new algorithm called generative adversarial minority oversampling [39], and we developed a customized algorithm based on genetic algorithms.

As already said, thanks to online banking services, there is a huge amount of transactions performed everyday, hence for a fraud detection system there is a huge amount of data to analyze everyday. In cases like that we talk about *Big Data*, that are difficult to manage with classical techniques. For this reason we also implemented a version of the framework in PySpark [45] in order to give a solution able to scale on a Big Data environment.

For the implementation of the framework we followed the one described in [12]. This paper developes a fraud detection system able to address class imbalance, concept drift, and the intrinsic latency between the time when a fraud is spotted and the labeling of the corresponding transaction. Following their work, we considered two types of transactions: the ones that are already labeled, that in a real setting are usually not available immediately, and the feedbacks coming from the investigators, usually available in short times. The first set of transactions was divided by hour and stored in datasets called delayed datasets, while the feedbacks were stored in a separated dataset called feedback dataset. The learning strategy consists in training everyday two different classifiers for these two kinds of transactions, and then aggregate their predictions to have the final predictions. For what concerns the feedback dataset, we trained a random forest on the feedbacks coming from the 7 hours prior to the training. For what concerns the delayed datasets, we trained an ensemble of classifiers. The ensemble was formed by 8 classifiers relative to the 8 hours before the first hour of feedbacks. So, in total, the aggregation was considering transactions coming from 15 hours before the training. In [12] the dataset was divided by day and not by hour. For us this was not possible, because the dataset we used was taken from kaggle [30], and it was relative to transactions covering 48 consecutive hours. Hence to imitate the functioning of their framework we had to work with datasets diveded by hour.

As said before the datasets were highly unbalanced. To solve this problem we performed oversampling on every hour dataset before using it for the training, that means that we created synthetic samples as similar as possible to the original transactions, until the dataset became balanced. The techniques that we used were random oversampling [35], generative adversarial networks [20], generative adversarial minority oversampling [39], and an algorithm based on genetic algorithms [25] developed by us for this specific problem.

After the oversampling phase, the model was trained. For the classifiers relative to the delayed datasets, we tried different solutions. At first we used random forest, as depicted in [12], then we used also extreme gradient boosting and neural network. We tried different configurations for every algorithm in order to find the best setting for our problem.
In our experiments we found out that the genetic algorithm was the oversampling method that achieved the best performance. For what concern the classification technique, none clearly overperfomed the others, but the best results were achieved using extreme gradient boosting. In an earlier phase of our experiments we also conducted some tests on the whole dataset, without dividing it in hours. From these experiments we found out that the model using neural network was clearly the best, being the one that spotted the highest number of frauds. These results, as already said weren't confirmed when we divided the dataset by hour and implemented the sequential learning strategy. One possible reason could have been that the dataset at our disposal was relatively small, and neural networks' performances increase with the size of data available.
The main limitation of our work, as just said, is the dataset used. We used a small dataset available on the repository of Kaggle, that is not as big as a stream of transactions we would have in a real environment. So the natural continuation of this work would be to test the framework on real setting, with the implementation in PySpark.

The thesis is structured as follows:

- In Chapter 2 we give a proper background on machine learning, on the techniques used for classification and for oversampling, as well as an overview of generative adversarial models and of genetic algorithms.

- In Chapter 3 we formulate our research problem, we present the workflow schema of our experiments, and we describe our solution based on genetic algorithms.

- In Chapter 4 we describe the tools and software used, the architecture of our system, and the implementation in Python and in PySpark.
- In Chapter 5 we present our dataset and the results obtained in our experiments with the relative evaluation metrics.
- In Chapter 6 we express the main limitation of our system, the conclusions, and we expose some possible paths for future research.

Chapter 2

State of the art

In this chapter we will explore the state of the art approaches in the fields relevant to the thesis. The chapter is organized in the following way: Section 2.1 will give an overview of Machine Learning, with focus on some of the most used techniques and algorithms. In Section 2.2 the architecture of neural networks and of their history will be presented. In Section 2.3 will formulate fraud detection problem in the contest of credit cards transactions and illustrate some of the approaches used in the literature for the solution. Section 2.4 will give necessary background about oversampling techniques, and will present the ones we used in our experiments as well. In Section 2.4.3, we will go in details about WGANs and GAMO networks, starting from the basic GAN architecture. Finally, in Section 2.5 we will present and discuss the family of Genetic Algorithms.

2.1 Machine Learning Background

Machine Learning is a subfield of artificial intelligence where systems try to learn and improve from experience to solve a specific task, without being

explicity programmed for that. The learning is obtained through data observation, trying to find patterns in them in order to make better decision in the future based on what they already saw. The primary goal is to avoid human intervention, allowing computers to learn autonomously.
Machine learning algorithms are usually categorized as follow:

- *Supervised Learning*, that is the group of algorithms that exploits the knowledge coming from labeled examples to predict future samples. More specifically, given a training data set (x) including desired outputs(t) $\mathcal{D} = \{\langle x, t\rangle\}$ from some unknown function f, we try to find a good approximation of f that generalizes well on data never seen before. Examples of supervised techniques are regression or classification.

- *Unsupervised Learning*, that is the set of algorithms used when we don't have at disposal labels for our training data. More specifically, given a training set without labels $\mathcal{D} = \{\langle x\rangle\}$ we try to find previously undetected patterns in the dataset. Examples of unsupervised techniques are clustering or compression.

- *Semi-supervised Learning*, that is a group of algorithms that combines supervised and unsupervised techniques. These algorithms are used when labeled and unlabeled samples are available, trying to take advantage from both. Examples of semi-supervised learning are generative models, graph-based methods or low-density separation.

- *Reinforcement Learning*, that is the set of algorithms that try to learn which action (u) to perform, interacting with the agents around, going to the current state (x) to another one (x'), in order to maximize a cumulative reward (r). These algorithms search for the best action by trial and error, learning automatically the ideal behavior (called *optimal policy*) to maximize their performance. More specifically, given a training set $\mathcal{D} = \{\langle x, u, x', r\rangle\}$ we want to find the optimal policy $\pi^*(x)$, that is a sequence of actions to perform. Examples of reinforcement learning are Markov Decision Processes or Stochastic Games.

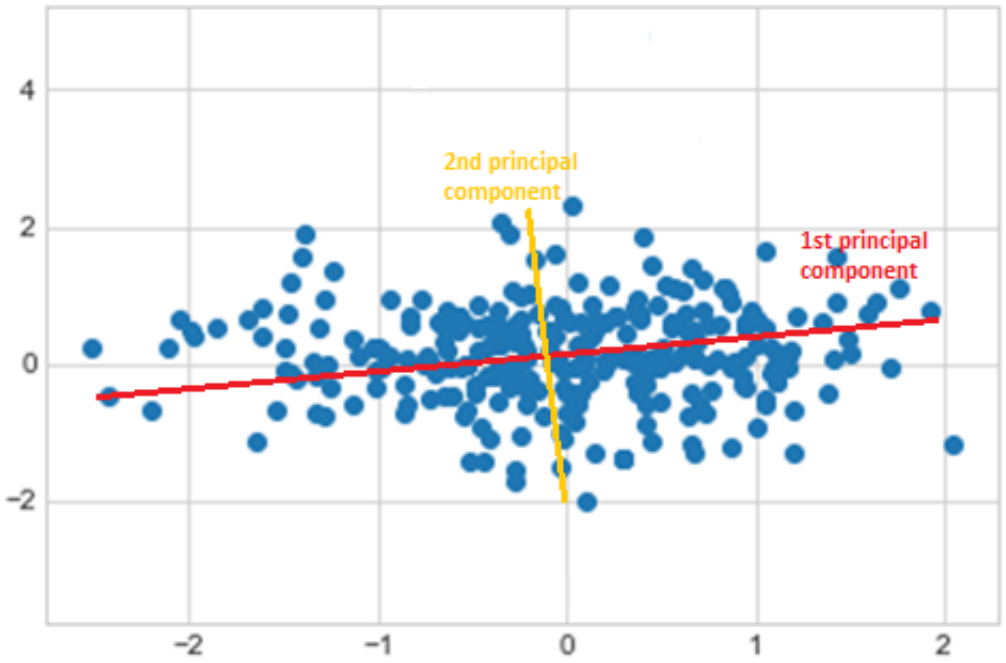

Figure 2.1: In here a simplified picture of the PCA result is depicted. The first principal component is the direction that explains the largest amount of variance of the dataset, while the second principal component is the one that explains the largest amount of variance between the components that are horthoganal to the first one.

In the context of our thesis we used Principal component analysis to perform feature selection in a part of our experiments, and also our dataset was the result of a PCA transformation. *Principal Component Analysis* (PCA) [52] is an unsupervised learning approach for *dimension reduction*, that is a group of techniques that convert the original feature space in a different one, allowing to make experiments on the new transformed space. This can be useful because the new representation can be more informative with respect to the old one on the task that has to be solved.

PCA performs an orthogonal transformation, converting a set of correlated features into a set of linearly uncorrelated ones, called *principal components*. The principal components are defined such that the first one is the one with largest possible variance, then all the succeeding ones are chosen as the one

with the largest variance between the orthogonal to the previous component (Figure 2.1). The resulting features are linear combination of the original ones, and are all uncorrelated with each other. Now we describe the exact procedure of the algorithm:

- Compute the mean of the data:

$$\overline{x} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{1}$$

- Bring the data to zero-mean (by subtracting $\overline{x}$)

- Compute the covariance matrix:

$$S = X^T X = \frac{1}{N-1}\sum_{n=1}^{N}(x_n - \overline{x})(x_n - \overline{x})^T \tag{2}$$

  - Eigenvector e_1 with largest eigenvalue λ_1 is the first principal component.
  - Eigenvector e_k with k^{th} largest eigenvalue λ_k is the k^{th} principal component.
  - $\frac{\lambda_k}{\sum_i \lambda_i}$ is the proportion of variance captured by the k^{th} principal component.

The projection of the original data onto the first k principal components ($E_k = (e_1, ..., e_k)$) create a new representation of the data with less dimensions:

$$X' = XE_k$$

In the context of our thesis we used as baseline classifier Random Forest, that is a machine learning method based on decision trees. A *Decision Tree* is a supervised algorithm used both for supervised tasks, as classification or regression, and unsupervised tasks, as clustering. It has a tree structure, and

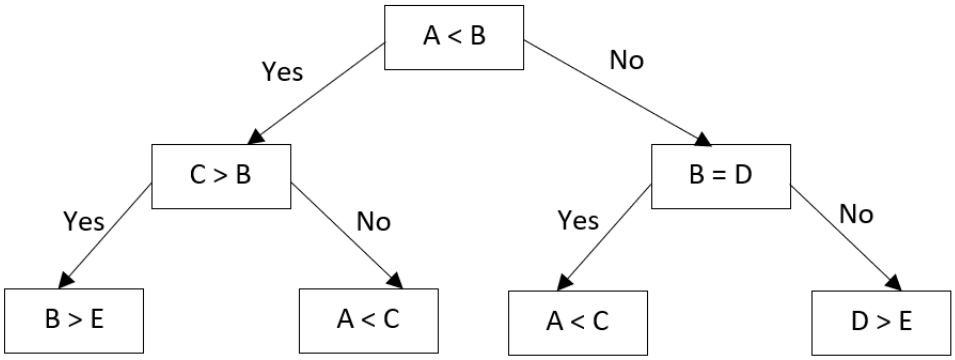

Figure 2.2: Basic schema of a Decision Tree.

each node of the tree represent a subset of the population we have. We start from the *root node*, that represents the entire population. The root node has to be split on one of the features of the dataset, generating two or more child nodes, called *decision nodes*, that have to be split in turn. Each split can be seen a test on an attribute, and it generates one or more branches that are the outcome of the test. They are performed on the most informative feature for the problem we're facing and to do that a *purity measure* is used (the most famous ones are *Entropy* and *Gini Index*).
The procedure continue until we cannot split nodes anymore on a branch, or a termination condition is met. The final nodes that cannot be split anymore are called *leaves*, and they represent a probability distribution on the class labels. Figure 2.2 shows the basic schema of a decision tree.

Algorithm 1 Random Forest Algorithm

for *b=1 to B* **do**

Draw a bootstrap sample Z^* of size N from the training data;

Grow a random forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached:

- select m attributes at random from the p attributes
- pick the best split-point from the m attributes
- split the node into the child nodes

end

Output the ensemble of tree $\{T_b\}_1^B$;

To make a prediction at a new point x let $\hat{C}_b(x)$ be the class prediction of the *b-th* random forest tree. Then $\hat{C}_{rf}^B(x) = majority\_vote\{\hat{C}_b(x)\}_1^B$

Ensemble methods are methods that generate a group of classifiers, and then predict the label of new data by aggregating the predictions of the whole set. *Random Forest* [6] is an ensemble method that combines decision trees predictors, that at each node perform the split using only a random subset of the features. The main advantages of a random forest over a decision tree is the reduction of overfitting and the much higher accuracy, at the cost of a more complex algorithm and less interpretable results.

In Algorithm 1 we depict the Random Forest algorithm. Random Forest is a *bagging* predictor [5]. A bagging predictor is an ensemble method that takes predictions from different predictors and aggregates them to have the final prediction. The aggregation is done by average in case of predicting a numerical outcome (e.g. regression), and by majority voting in case of predicting a class (e.g. classification). The single predictors are trained only on a subset of the dataset, that is different for each one of them. Bagging methods are very useful when aggregating learners for which a small change in the training set can cause significant changes in the predictions, like it

happens for decision trees.

Another ensemble method is *boosting* [48]. Boosting is a strategy where multiple simple models (called *weak learners*) are sequentially combined in a single composite model. Each weak learners is weighted with respect to the samples that have been misclassified more frequently so far. The idea is that the more weak models we add, the stronger the composite model becomes.
Gradient Boosting [14] is a boosting techniques that at first train a regression predictor, and then trains each one of the following models on the residual error of the previous ensemble on an error function. The aim is to minimize this error function, that can be of different types and depends on the algorithm we want to use. In order to minimize the function, the algorithms makes use of *gradient descent*, from which it takes the name. Decision trees of fixed size are usually used as base learners, and in this case we talk of *Gradient Tree Boosting* (GTB).
In [9] the authors proposed a scalable implementation for the GTB called *eXtreme Gradient Boosting* (XGBoost). XGBoost improves the basic framework through some enhancements; the most important are parallelization of the process, hardware optimization, and the use of regularization to prevent overfitting. XGBoost has been widely used with very good performances in data science solutions, being the approach used in most of the winning solutions in data science competitions.

2.2 Artificial Neural Networks

An artificial Neural Network (ANN) is a classifier that tries to mimic the functioning of the human brain. Human brain is composed by nerve cells and neurons in very big amount, that are connected to each other creating a network of signal transmission. Every cell sends signals to the cells they are connected to, and when one of them reaches a certain threshold because of the input received, it releases a signal itself to all the cells it is connected to.

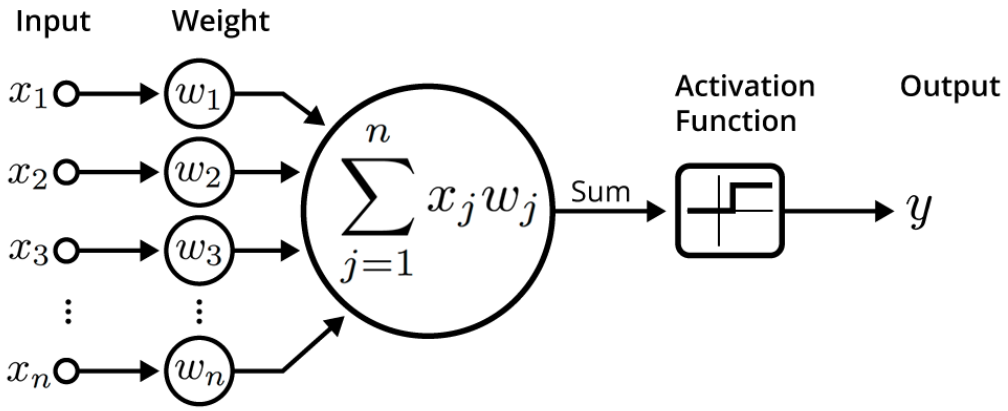

Figure 2.3: Graphical representation of a perceptron. Picture taken from [27].

An ANN works with the same principle, but instead of the neuron, it uses the "perceptron" as basic unit (also referred as neuron). The perceptron has a number of weighted inputs and an activation function, and when these combined inputs exceed a certain threshold it sends a signal as output. The output sent is defined by the *activation function*, and it is usually in the range $(0, 1)$ and $(-1, 1)$. An example of function in the range $(0, 1)$ is the *sigmoid* activation function, that is used when we need to predict a probability as outcome. An example of function in the range $(-1, 1)$ is the *hyperbolic tangent* activation function, that is used for classification with two classes. The equation for a perceptron can be written as:

$$y = g(\sum_{i=1}^{n} w_{\mathrm{i}} x_{\mathrm{i}} + b) \tag{3}$$

where y is the output signal, g is the activation function, n is the number of connections to the perceptron, w_{i} is the weight associated with the ith connection and x_{i} is the value of the *i-th* connection, b represents the threshold, that is always set to a constant value of -1. A graphical representation can be found in Figure 2.3. The strength of this model can be seen clearly when more perceptrons are combined and put together. The perceptrons are disposed in layers, that are neurons at the same distance from the input neu-

rons. All the elements of a layer takes in input the signals coming from the elements of the previous one, they apply the weights and then they send a signal to the next layer depending from the activation function. An artificial neural network is a non-linear model that is composed by number of neurons, activation functions, and the values of weights and biases, that are arranged in a particular topology.

The layers can be of three types: *input layer*, that is composed by the neurons that receives the input from the data; *output layer*, that is composed by the neurons that gives the final result of the network; *hidden layer*, that is a layer of neurons that process data coming from neurons from the previous layer and send information to neurons in the next one.

The most simple ANN is composed by the input layer, one hidden layer, and the output layer composed by a single neuron. In Figure 2.4 we can see a multi-layer perceptron with four inputs, one hidden layer of five neurons, and one output.

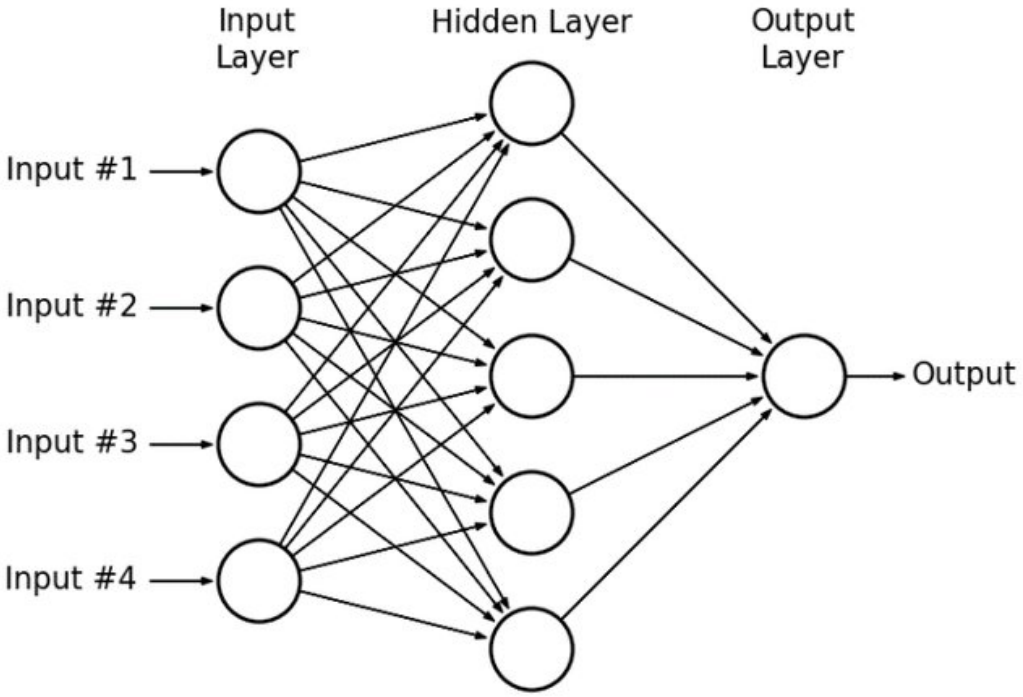

Figure 2.4: Multi-layer perceptron with four inputs, one hidden layer with five neurons and one output. Picture taken from [23].

In an ANN the learning procedure consists in updating the weights associated with the connections between the layers. This is usually achieved by "back-propagation". Back-propagation consists in feeding backward the error that the network makes at the output on new examples. By iteratively repeating this process the network can learn to discriminate trough different classes. The learning equations are the following:

$$w^{k+1} = w^k + \Delta w \tag{4}$$

$$\Delta w = -\eta \cdot \frac{\partial E}{\partial w} \tag{5}$$

with:

$$E = \sum_{n}^{N} (t_{\mathrm{n}} - y_{\mathrm{n}})^2 \tag{6}$$

$$y = g(\sum_{j}^{J} W_{\mathrm{j}} \cdot h(\sum_{i}^{I} w_{\mathrm{ji}} \cdot x_{\mathrm{i}})) \tag{7}$$

and with t being the desired output of the network, η being the learning rate, that is a parameter that determines the step size at each iteration of the learning procedure, and $h(\cdot)$ being the activation function of the hidden layer.

2.3 Fraud Detection

Fraud detection for credit cards transactions aims at distinguishing between genuine transactions and fraudulent ones, and a lot of methods have been developed in the literature, using supervised [12], unsupervised, semi-supervised [7], and reinforcement learning [49] methods.
When the problem is addressed, there are some specific challenges that have to be faced. The main issues are class imbalance, meaning that frauds transactions are way fewer than genuine ones, and concept drift, meaning that there could be changes in the distribution of the transactions over time.
As depicted in [12], a *Fraud Detection System*(FDS) is typically composed of five layers of control, that are also shown in Figure 2.5:

1. *Terminal:* it represent the first control layer and it performs security checks as controlling PIN code, card status, number of attempts, and so on. The latency of these checks must be really small, because the response should be real time. The requests that passed all the checks become transactions and enter the second layer, while all the others are denied.

2. *Transaction-Blocking Rules:* these are if-then(-else) statements designed by physical investigators, hence they are the expert-driven component of the FDS. The rules analyze only the information available at the

time of the request, without analyzing historical records. These rules must be quick, and they should raise very few false alarms, so they have to be very specific. An example of rule is *"IF internet transaction AND unsecured website THEN deny the transaction"*.

3. *Scoring Rules:* these are of the same form of the previous ones, but they operate on the vector of features of the transactions, and they assign to each of them a score measuring how risky it is. An example of rule is *"IF internet transaction AND cardholder age > 60 AND first internet transaction THEN fraud score = 0.95"*. Also these rules are designed by investigators, so they are limited to detect only those frauds whose patterns are already been identified.

4. *Data Driven Model (DDM):* this layer estimates for each transaction the probability of being a fraud, that will become the associated fraud score. This is usually done by mean of a classifier trained from a set of labelled transactions. Then the transactions with a high fraud score generates alerts, but only some of them are reported to the investigators, usually the first k alerted, with k being a paremeter that varies along with the avilability of the investigators. Physical investigators are the final layer of control.
   The investigators cannot interfere with the DDM, because the DDM aims at finding frauds patterns that are beyond their experience and that are not necessarily explicable.

5. *Investigators:* this is the last layer of the system and it's composed by professionals that are in charge of checking if an alert raised by the previous layers is a fraud or not. Their main job is to contact cardholders, to label alerts as "fraud" or "genuine" and insert them in the FDS. These labeled transactions are called feedbacks. In practice investigators can only check few alerts per day, so the main goal of the DDM is to raise very precise alerts and to avoid false positives.

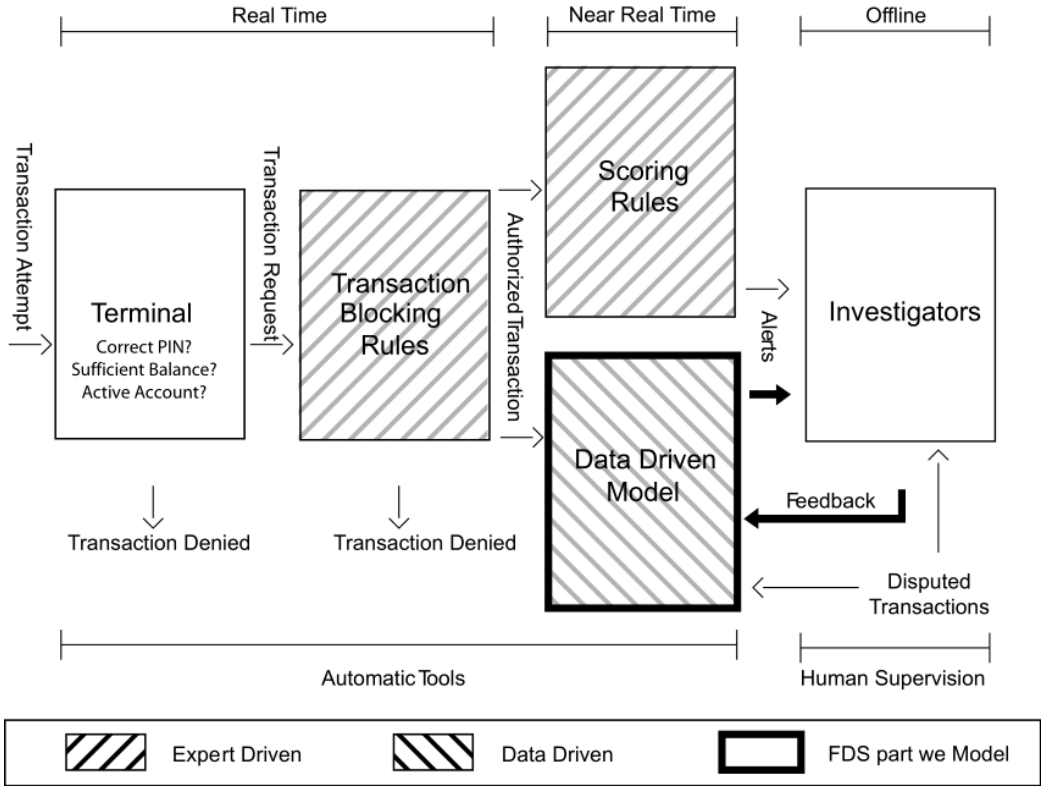

Figure 2.5: The picture illustrates the layers of control in a FDS. Our role is to design the data driven model and the alert-feedback interaction. Picture taken from [12].

In [12] Dal Pozzolo et al. address all the main problems relative to Fraud Detection, presenting a realistic learning strategy for the detection of frauds. The framework used in this paper is the same that we adopted in our experiments, hence a detailed description of it will be provided in the corresponding section. They assume that in a real setting not all the available transactions are labeled at time t, because there is a latency due to the *alert feedback interaction*, the process from a transaction creation to its labeling and inserting in the dataset. Hence, two types of data are available: *delayed samples*, that are the labeled samples that have been inserted in the dataset after a certain amount of time and that are labeled according to investigations of the experts or reports of the costumers; and *feedbacks*, that are the few transactions that can be investigated by the professionals, that are available every day.

The main innovation in their work is that they trained a classifier $\mathcal{K}$, composed by the aggregation of the posterior probability of two different classifiers: one for the delayed samples, and another for the feedbacks. This is important because it exploits the fact that the two sets of data have different distributions, and that investigated separately they can give additional information for the solution.

Another important contribution of the paper is the introduction of performance measures that are more suitable for fraud detection. Given the nature of the problem, classic metrics as accuracy, precision or recall are not very informative for the performance of the system. Because of the class imbalance, a classifier could have really high score for these metrics, but then result in poor performance when it comes to new samples because it focused only on the majority class, ignoring the minority one. They argue also that the limited time that the investigators have should be taken into account, so the FDS should provide a small number of precise alerts, allowing them not to waste time. For these reasons they introduced two specific measures: *alert precision* $P_k(t)$ and *card precision* $CP_k(t)$. The first one is defined as the precision on the k most risky transactions; the second one address the fact that multiple fraudulent transactions coming from the same card, should be counted as a single correct detection, so it's defined as the proportion of fraudulent cards detected in the k cards controlled by the investigators.

In the work a costant verification latency of δ is considered, and to process transactions authorized at day $t+1$ are considered Q days of feedbacks, $\{F_t, ..., F_{t-(Q-1)}\}$, and M days of delayed samples $\{D_{t-\delta}, ..., D_{t-(\delta+Q-1)}\}$, where $M = Q - \delta$.

The proposed learning strategy consist in separetely training a classifier $\mathcal{F}_t$ on feedbacks:

$$\mathcal{F}_t = TRAIN(\{F_t, ..., F_{t-(Q-1)}\}) \tag{8}$$

and a classifier on delayed samples:

$$\mathcal{D}_t = TRAIN(\{D_{t-\delta}, ..., F_{t-(\delta+Q-1)}\}) \tag{9}$$

and then the frauds are detected by the aggregation classifier $\mathcal{A}_t$, that has as posterior probability:

$$\mathcal{P}_{\mathcal{A}_t}(+|x) = \alpha\mathcal{P}_{\mathcal{F}_t}(+|x) + (1-\alpha)\mathcal{P}_{\mathcal{D}_t}(+|x) \tag{10}$$

where $\alpha = 0.5$ is the weight parameter that balanced the two contributions. For all the classifiers used a Random Forest of 100 trees has been used. Each tree has been trained on a balanced bootstrap sample obtained with Random Undersampling of the majority class, in order to solve class imbalance.

In [49] they make use of reinforcement learning to tackle the problem. The authors claim that in fraud detection they need to face a trade-off between exploration and exploitation, where exploration consists in the investigation of transactions with the purpose of improving predictive models, and exploitation consists in investigating transactions predicted to be risky. To deal with this trade-off they use a *Contextual Multi-Armed Bandit*(CMAB) [55].
Multi-Armed Bandit is a reinforcement learning technique where there is a set of N arms, and to each arm is associated a reward distribution that is unknown. The reward can be obtained in a deterministic way, meaning that we have a single value for the reward for each arm (trivial solution); in a stochastic way, meaning that the reward of an arm is drawn from a stationary distribution; in an adversarial way, meaning that an adversary chooses the reward for an arm at a given time step. In this case $r_t = amount(x_{i_t}) \times y_{i_t}$ is the reward obtained if arm i is investigated at time step t, with $amount(x_{i_t})$ being a feature of transaction i at time t that denotes its amount. They want to maximize the cumulative reward $R(\tau) = \sum_{t=1}^{\tau} r_t$ collected over a large number of steps τ. At each time step t an arm is selected and pulled,

and a reward from its reward distribution is given to the system, that has the goal of maximizing the cumulative reward. In the CMAB context vectors are available at each time step, that are related to the reward obtained by the arms. A context vector is a vector related to an element of the dataset that contains additional information related to the reward. In this paper the problem can be seen as a CMAB where each transaction corresponds to an arm in the bandit problem. The problem with this formulation is that there are too many transactions, hence the computation can be very expensive. In order to solve the problem, the number of arms can be reduced by clustering them. The clustering is performed by means of a Regression Tree $\mathcal{T}$, whose goal is to predict the reward r_i associated with each transaction i, and in which each leaf with the corresponding transactions is a cluster. This is done because the tree creates branches, that can be seen as arms for our bandit problem.

Every transaction is represented with a vector (x_i, y_i), where $x_i \in \mathbb{R}^d$ is the feature vector and $y_i \in \{0, 1\}$ is the class label (0 for genuine and 1 for fraud). Various CMAB algorithm were evaluated in the paper, but the best performing one was found to be Bootstrap Thompson Sampling (BTS), of which we provide the pseudocode, as depicted in [49]:

Algorithm 2 BTS

Initialize J models, $R(\tau) \leftarrow 0$

for $t = 1, 2, ..., \tau$ **do**

  $x_a \leftarrow$ feature vector for every arm a

  Randomly select one model j_t out of J

  Play arm with highest reward predicted by j_t

  Observe reward $r(t)$

  for *j=1, 2, ..., J* **do**

    Update j^{th} model using x_{a_t} and $r(t)$ w.p. 0.5

  end

  $R(\tau) \leftarrow R(\tau) + r(t)$

end

In [7] the authors combine supervised and unsupervised techniques in a semi-supervised learning strategy to detect frauds. The main idea here is to use unsupervised learning to compute different kind of outlier score for every transaction, that is the probability for that transaction to be different from the vast majority of the data. These scores are then aggregated with the feature vectors of the corresponding transactions, and then supervised learning based on Balanced Random Forest is performed on the augmented dataset.
The types of score used are: *Z-score*, *PC-1*, *PCA-RE-1*, *IF*, and *GM-1*. *Z-score* is defined as follows:

$$\sum_{i=1}^{f} \left(\frac{x_i - \hat{\mu}_i}{\hat{\sigma}_i} \right) \tag{11}$$

with $x \in \mathbb{R}^d$.
PC-1 and *PCA-RE-1* are both based on *Principal Component Analysis* (PCA), that transforms the original dataset X in a new representation $T = XW$ of d linearly uncorrelated features, called principal components. PCA is widely used for outlier detection, and the two scores are defined as:

$$PC\text{-}1 = W_1^T x \tag{12}$$

that is the value of the first principal component, and:

$$PCA\text{-}RE\text{-}1 = \|x - WW^T x\| \tag{13}$$

that is the reconstruction error obtained by using the first component.
The score *IF* is based on Isolation Forest [36], and gives to each sample an outlier score depending on the distance between the leaf node and root of a random forest. Finally, *GM-m* is the density in x of a Gaussian mixture (GM) model fit to the dataset, and m is the number of mixtures.

The outlier score is computed with different levels of granularity: *global*, where they assume that all the transactions are taken from the same distribution, *local*, where they assume that every cardholder has a different behaviour, hence transactions from different cards have different distributions, and *cluster*, that tries to overcome both the previous method's limit, by clustering transaction on the card's level, and then assuming that there is a distribution for every cluster. The aggregation is performed with the k-means algorithm, and the number of clusters was varying from 10 to 5000. After the computation of the outlier scores, they were added to the other features, and they were taken into account for the classification algorithm.

2.4 Oversampling

Class imbalance is a problem that we have when the number of data of a specific class is really small with respect to the number of data of the other classes. Most of machine learning algorithms are developed to give best results when the number of samples in each class is roughly equal, but when this doesn't happen problems may arise. There are many situations in wich we have imbalance, such as rare diseases in medical diagnosis, oil spills in satellite radar images, spotting unreliable telecommunication costumers, risk management, information retrieval, intrusion detection, shuttle system failures, earthquakes and nuclear explosions, and of course credit card frauds detection [21].

One direct solution to tackle this problem is to adopt techniques that change the distribution of the classes. This can be done in two ways: undersamping the majority class or oversampling the minority class. In [28] they came with the conclusion that undersampling and oversampling techniques are very effective methods to overcame class imbalance.

In our work we focused on different oversampling techniques and how they

perform in the classification of credit card transactions. The most used ones are ROS, SMOTE and ADASYN.

2.4.1 ROS, SMOTE and ADASYN

With Random Oversampling (ROS) the class distribution is balanced by random replicating examples belonging to the minority class. The examples to be replicated are chosen randomly from the original training set, and not from the new training set, otherwise the randomness of the selection would be biased. Furthermore, the oversampling is done with replacement, otherwise the members of the minority class would be over before reaching the desired balance between classes [35]. These technique is widely used because it is very simple and it doesn't require too much knowledge about the dataset.
However, there are two main drawbacks in ROS: overfitting would be more likely to occur, and it makes the learning process more time consuming when the dataset is both large and umbalanced [28].

SMOTE [8] is an oversampling technique that creates "synthetic" samples instead of duplicating existing ones. Each minority class example is taken and the new samples are produced along the line segments that join the selected point with the k nearest neighbors belonging to the same class. The synthetic samples are generated by taking the difference between the selected sample and the neighbor and multiplying this difference by a random number between 0 and 1, causing the selection of a point randomly in the line segment just described. This procedure makes the decision region of the minority class more general. In Figure 2.6 we can see a graphical representation of the generation.
If we consider a sample x_1 with coordinates (x_{1a}, x_{1b}), it is the sample for which k nearest neighbors are identified. Assume that the nearest neighbor

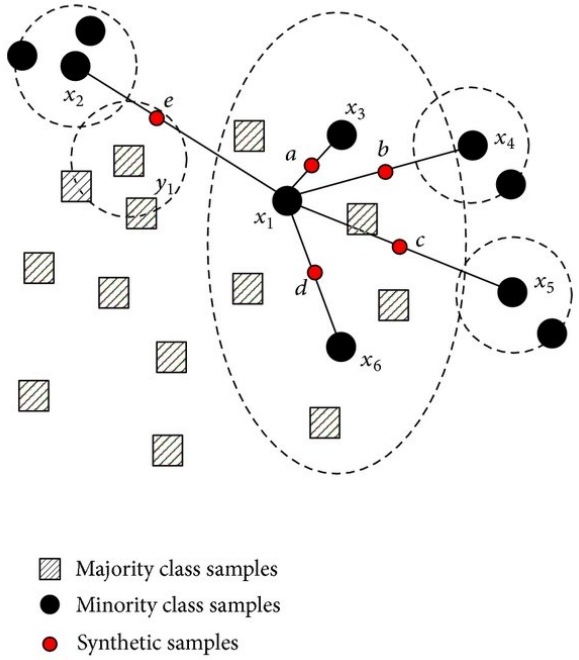

Figure 2.6: In here we can see how the oversampling is performed taking into account the point x_1. The difference vectors with the k-nearest neighbors(x_2, x_3, x_4, x_5, x_6) is computed, and from those vectors, k new samples are generated(a, b, c, d, e). Picture taken from [8].

is $(x_{2\mathrm{a}}, x_{2\mathrm{b}})$. Then the new sample will be generated as:

$$(x'_a, x'_b) = (x_{1\mathrm{a}}, x_{1\mathrm{b}}) + rand(0,1) \times (x_{2\mathrm{a}} - x_{1\mathrm{a}}, x_{2\mathrm{b}} - x_{1\mathrm{b}}) \tag{14}$$

The SMOTE algorithm, as described in details in [8], is reported in Algorithm 3.

Algorithm 3 SMOTE(T, N, k)

Input : Number of minority class samples T ; Amount of SMOTE $N\%$; Number of nearest neighbors k

Output: $(N/100) \times T$ synthetic minority class samples

if $N < 100$ **then**
    Randomize the T minority class samples;
    $T = (N/100) \times T$;
    $N = 100$;
end

- $N = (\text{int})(N/100)$ (*The amount of SMOTE is assumed to be in integral multiples of 100*)
- $k =$ Number of nearest neighbors
- $numattrs =$ Number of attributes
- $Sample[][]$: array for original minority class samples
- $newindex$: keeps a count of number of synthetic samples generated, initialized to 0
- $Synthetic[][]$: array for synthetic samples (* Compute k nearest neighbors for each minority class sample only. *)

for $i = 0$ *to* T **do**
    Compute k nearest neighbors for x_i, and save the indices in the $nnarray$;
    Populate$(N, i, nnarray)$;
end
while $N \neq 0$ **do**
    Choose a random number between 1 and k, call it nn. This step chooses one of the k nearest neighbors of i;
    for $attr = 1$ *to* $numattrs$ **do**
        Compute: $dif = Sample[nnarray[nn]][attr] - Sample[i][attr]$;
        Compute: $gap =$ random number between 0 and 1;
        $Synthetic[newindex][attr] = Sample[i][attr] + gap \times dif$;
    end
    $newindex++$;
    $N = N - 1$;
end

ADASYN [24] is an oversampling technique derived by SMOTE. A density distribution $\hat{r}_{\mathrm{i}}$ is used to decide the number of synthetic samples to be generated for each sample in the minority class. $\hat{r}_{\mathrm{i}}$ measure the distribution giving more weight to minority class examples that are more difficult to learn, forcing the learning algorithm to focus on these data points. That's the major difference with SMOTE, in which the number of generated samples for each point is equal.
The algorithm takes in input the training dataset D_{tr} with m samples $\{x_i, y_i\}$, and with $i = 1, ..., m$. $x_i \in X$, where X is the n-dimensional feature space, and $y_i \in Y$, where $Y = \{1, -1\}$ is the class identity label. Then m_s is defined as the number of minority class examples and m_l as the number of majority class examples, such that $m_s \leq m_l$ and $m_s + m_l = m$.
The procedure, as described in [24], is the following:

1. Calculate the degree of class imbalance

$$d = m_{\mathrm{s}}/m_{\mathrm{l}} \tag{15}$$

   with $d \in (0, 1]$

2. If $d < d_{\mathrm{th}}$ then ($\mathrm{d_{th}}$ is a preset threshold for the maximum tolerated degree of class imbalance ratio):

   (a) Calculate the number of synthetic data examples that need to be generated for the minority class:

$$G = (m_{\mathrm{l}} - m_{\mathrm{s}}) \times \beta \tag{16}$$

   where $\beta \in [0, 1]$ is a parameter used to specify the desired balance level after generation of the synthetic data. $\beta = 1$ means a fully balanced dataset is created after generalization process.

   (b) For each example $x_i \in S$, S being the set of samples belonging to the minority class, find K nearest neighbors based on the Euclidean distance in n dimensional space, and calculate the ratio r_i defined as:

$$r_{\mathrm{i}} = \Delta_{\mathrm{i}}/K, i = 1, ..., m_{\mathrm{s}} \tag{17}$$

where Δ_{i} is the number of examples in the K nearest neighbors of x_i that belong to the majority class, therefore $r_{\mathrm{i}} \in [0, 1]$;

(c) Normalize r_{i} according to

$$\hat{r}_{\mathrm{i}} = r_{\mathrm{i}} / \sum_{i=1}^{m_{\mathrm{S}}} r_{\mathrm{i}},$$

so that $\hat{r}_{\mathrm{i}}$ is a density distribution ($\sum_i r_{\mathrm{i}} = 1$).

(d) Calculate the number of synthetic data examples that need to be generated for each minority example x_i:

$$g_{\mathrm{i}} = \hat{r}_{\mathrm{i}} \times G \tag{18}$$

where G is the total number of synthetic data examples that need to be generated for the minority class as defined in Equation (3).

(e) For each minority class data example x_i, generate g_i synthetic data examples according to the following steps:
Do the **Loop** from 1 to g_{i}:

i. Randomly choose one minority data example, x_{zi}, from the K nearest neighbors for data x_{i}.

ii. Generate the synthetic data example:

$$s_{\mathrm{i}} = x_{\mathrm{i}} + (x_{\mathrm{zi}} - x_{\mathrm{i}}) \times \lambda \tag{19}$$

where $(x_{\mathrm{zi}} - x_{\mathrm{i}})$ is the difference vector in n dimensional spaces, and λ is a random number: $\lambda \in [0, 1]$.

End **Loop**.

2.4.2 GAN and GAMO

Generative Adversarial Networks (GANs) are neural networks composed by two components: a generator and a discriminator. The objective of the generator is to generate fake samples and try to fool its adversary. The objective

of the discriminator is to distinguish between fake and real samples. The network is determined by a minmax game, that ends when the discriminator is not able to distinguish anymore samples produced by the generator from samples belonging to the original dataset.
In [3] the author makes use of GANs as oversampling techniques to generate artificial samples on a dataset of credit cards transactions, improving the discriminatory power of the classifier. In the experiments the use of GANs as oversampling technique is compared with ROS, SMOTE and ADASYN, showing that GAN architectures(especially the variant called Wasserstein GAN) produce more realistic fraudolent transactions, leading to more stable results.
In [39] a new generative model based on GANs is developed, called Generative Adversarial Minority Oversampling.

2.4.3 Generative Adversarial Networks

In machine learning there are two main approaches for building statistical classifiers: *discriminative* and *generative*. Discriminative models map the samples to a class label to perform their task, and they do it by learning the conditional probability distribution of the input data using its own features. Generative models' goal is to generate synthetic samples the most similar to the original input, estimating and exploiting the joint probability distribution of the high dimensional input data.
The GANs are a particular type of neural networks, in which we estimate generative models trough an adversarial process [20]. Two models are trained simultaneously in the framework: a *generative model G*, that tries to generate synthetic samples capturing the data distribution, and a *discriminative model D*, that tries to distinguish fake samples from data coming from the original distribution. As stated in [20], the clearest modeling framework is when the models are both multi-layer perceptrons.

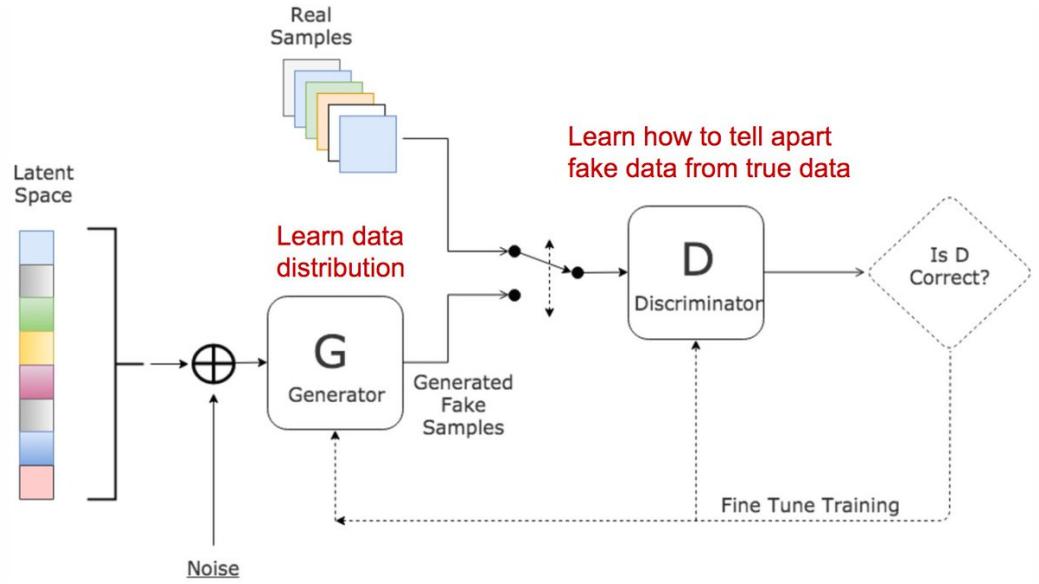

Figure 2.7: Generative Adversarial Network Model. Picture taken from [19].

We define as p_{g} the *generator's distribution* over data $\mathbf{x}$. We define $p_{\mathrm{z}}(z)$ as a prior on input noise, in order to learn p_{g}, and then we represent a mapping to the data space as $G(z;\theta_{\mathrm{g}})$, where G is the multi-layer perceptron equation with parameters θ_{g}. Finally, we define $D(x;\theta_{\mathrm{d}})$ as the equation of the discriminator, that outputs a single scalar corresponding to the probability that x came from the data rather than p_{g}. In Figure 2.7 the architecture of a GAN is showed.

The learning procedure consists in training D in order to maximize the probability of assigning the correct label to synthetic and original samples (the labels in this case are "fake" or "original"). Simoultaneously we train G to minimize $log(1 - D(G(Z)))$.

This consists in practice in the following two players minimax game:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\mathrm{data}}(x)}[logD(x)] + \mathbb{E}_{z\sim p_{\mathrm{z}}(z)}[log(1 - D(G(Z)))] \quad (20)$$

The game is implemented using an alternated approach, but optimizing D completely in the inner loop is prohibitive in terms of computational power,

and it can lead to overfitting in finite datasets [20]. The solution is to train D for k steps, and then training G for one. In this way D is maintained near its optimal solution if G changes slowly enough.

Algorithm 4 Minibatch stochastic gradient descent training of generative adversarial nets.

for *number of training iteration* **do**

  for *k steps* **do**

    - Sample minibatch of m noise samples $(z^{(1)}, ..., z^{(m)})$ from noise prior $p_g(z)$.

    - Sample minibatch of m examples $(x^{(1)}, ..., x^{(m)})$ from data generating distribution $p_{data}(x)$

    - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[logD\left[x^{(i)}\right] + log\left(1 - D(G(z^{(i)}))\right)\right] \tag{21}$$

  end

  - Sample minibatch of m noise samples $(z^{(1)}, ..., z^{(m)})$ from noise prior $p_g(z)$.

  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} log\left(1 - D(G(z^{(i)}))\right) \tag{22}$$

end

The gradient-based updates can use any standard gradient-based learning rule. The authors of [20] used momentum.

At a certain point in the training, a global optimum will be reached, that corresponds to the point in which $p_{\mathrm{g}} = p_{\mathrm{data}}$. This happens when $D(x) = \frac{1}{2}$ and the discriminator is not able to distinguish anymore between the two

distributions. This point in practice is hardly reached because of computational constraints.
Now we will prove what we just said, following the procedure explained in [20].

Proposition 1. *For G fixed, the optimal discriminator D is:*

$$D_G^*(\mathbf{x}) = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})} \tag{23}$$

Proof. The training criterion for the discriminator D, given any generator G, is to maximize the quantity $V(G, D)$

$$\begin{aligned} V(G,D) &= \int_x p_{\text{data}}(\mathbf{x}) log(D(\mathbf{x}))dx + \int_z p_{\text{z}}(\mathbf{z}) log(1 - D(G(\mathbf{z})))dz \\ &= \int_x p_{\text{data}}(\mathbf{x}) log(D(\mathbf{x})) + p_{\text{g}}(\mathbf{x}) log(1 - D(\mathbf{x}))dx \end{aligned} \tag{24}$$

For any $(a, b) \in \mathbb{R}^2 \backslash \{0, 0\}$, the function $y \to a \cdot log(y) + b \cdot log(1-y)$ achieves its maximum in $[0, 1]$ at $\frac{a}{a+b}$. The discriminator does not need to be defined outside of $Supp(p_{\text{data}}) \cup Supp(p_{\text{g}})$, concluding the proof.

□

The minimax game now can be expressed as:

$$\begin{aligned} C(G) &= \max_D V(G, D) \\ &= \mathbb{E}_{x \sim p_{\text{data}}}[log D_G^*(\mathbf{x})] + \mathbb{E}_{z \sim p_{\text{z}}}[log(1 - D_G^*(G(\mathbf{z})))] \\ &= \mathbb{E}_{x \sim p_{\text{data}}}[log D_G^*(\mathbf{x})] + \mathbb{E}_{x \sim p_{\text{g}}}[log(1 - D_G^*(\mathbf{x}))] \\ &= \mathbb{E}_{x \sim p_{\text{data}}}\left[log \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_{\text{g}}(\mathbf{x})}\right] + \mathbb{E}_{x \sim p_{\text{g}}}\left[log \frac{p_{\text{g}}(\mathbf{x})}{p_{\text{data}}(\mathbf{x}) + p_{\text{g}}(\mathbf{x})}\right] \end{aligned} \tag{25}$$

Theorem 2.4.1. *The global minimum of the virtual training criterion $C(G)$ is achieved if and only if $p_g = p_{data}$. At that point, $C(G)$ achieves the value $-log4$.*

Proof. For $p_{\text{g}} = p_{\text{data}}$, $D_G*(\mathbf{x}) = \frac{1}{2}$. Hence, as proved before, at this value, we have $C(G) = log\frac{1}{2} + log\frac{1}{2} = -log4$. To see that this is the best possible value of C(G), reached only for $p_{\text{g}} = p_{\text{data}}$, observe that:

$$E_{x\sim p_{\text{data}}}[-log2] + E_{x\sim p_{\text{g}}}[-log2] = -log4 \tag{26}$$

and that by subtracting this expression from $C(G) = V(D^*_G, G)$, we obtain:

$$C(G) = -log(4) + KL\left(p_{\text{data}}\middle\|\frac{p_{\text{data}}}{2+p_{\text{g}}}\right) + KL\left(p_{\text{data}}\middle\|\frac{p_{\text{g}}}{2+p_{\text{g}}}\right) \tag{27}$$

where KL is the *Kullback-Leibler divergence*(that we will expain in details in the next section). The previous expression is known as the *Jensen-Shannon divergence* between the model's distribution and the data generating process:

$$C(G) = -log(4) + 2\cdot JSD(p_{\text{data}}\|p_{\text{g}}) \tag{28}$$

The Jensen-Shannon divergence is always non-negative, and it is equal to zero only when the two distributions taken into account are equal. So we can say that $C* = -log(4)$ is the global minimum of $C(G)$ and that $p_{\text{g}} = p_{\text{data}}$ is the only solution, meaning that the generative model has copied the distribution of the original data. □

Now that we have described the GAN model, we will introduce the Wasserstein GAN, that will be one of the architectures we will use in our experiments.

2.4.4 Wasserstein GAN

The Wasserstein Generative Adversarial Network (WGAN) is an extension of the classical GAN that uses a loss function that is correlated to the quality of the generated samples, and that improves the stability of the learning procedure. Let's define elementary distances and divergences between two

distributions $\mathbb{P}_r, \mathbb{P}_g \in Prob(\chi)$, with χ being a compact metric set (such as the space of images $[0,1]^d$):

- The *Total Variation* (TV) distance:

$$\delta(\mathbb{P}_r, \mathbb{P}_g) = \sup_{A \in \Sigma} |\mathbb{P}_r(A) + \mathbb{P}_g(A)| \tag{29}$$

  with Σ being the set of all the *Borel* subsets of χ.
  A Borel set is any set in a topological space that can be formed from open sets (or, equivalently, from closed sets) through the operations of countable union, countable intersection, and relative complement.

- The *Kullback-Leibler* (KL) divergence:

$$KL(\mathbb{P}_r \| \mathbb{P}_g) = \int log\left(\frac{P_r(x)}{P_g(x)}\right) P_r(x) d\mu(x) \tag{30}$$

  where $\mathbb{P}_r$ and $\mathbb{P}_g$ both admit densities because absolutely continuous, with respect to the same measure μ defined on χ. The KL divergence is asymmetric and infinite when $P_g(x) = 0$ and $P_r(x) > 0$

- The *Jensen-Shannon* (JS) divergence

$$JS(\mathbb{P}_r, \mathbb{P}_g) = KL(\mathbb{P}_r \| \mathbb{P}_m) + KL(\mathbb{P}_g \| \mathbb{P}_m) \tag{31}$$

  where $\mathbb{P}_m = \frac{\mathbb{P}_r + \mathbb{P}_g}{2}$.

- The *Earth-Mover* (EM) distance or *Wasserstein-1*:

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|] \tag{32}$$

  where $\Pi(\mathbb{P}_r, \mathbb{P}_g)$ is the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively $\mathbb{P}_r$ and $\mathbb{P}_g$.
  In other words, γ is how much mass should be moved from x to y in order to make the two distributions coincide, while the EM distance is the cost we need to pay to do that.

Now we can present two theorems, formulated in [1].

Assumption 2.4.1. *Let $g : \mathcal{Z} \times \mathbb{R}_d \to \chi$ be locally Lipschitz between finite dimensional vector spaces. We will denote with $g_\theta(z)$ its evaluation on coordinates (z, θ). We say that g satisfies assumption 2.4.1 for a certain probability distribution p over $\mathcal{Z}$ if there are local Lipschitz constants $L(\theta, z)$ such that:*

$$\mathbb{E}_{z \sim p_z}[L(\theta, z)] < +\infty \tag{33}$$

Theorem 2.4.2. *Let $\mathbb{P}_r$ be a fixed distribution over χ. Let Z be a random variable (e.g Gaussian) over another space $\mathcal{Z}$. Let $g : \mathcal{Z} \times \mathbb{R}^d \to \chi$ be a function, that will be denoted $g_\theta(z)$ with z the first coordinate and θ the second. Let $\mathbb{P}_\theta$ denote the distribution of $g_\theta(Z)$. Then:*

1. *If g is continuous in θ, so is $W(\mathbb{P}_r, \mathbb{P}_\theta)$.*
2. *If g is locally Lipschitz and satisfies regularity Assumption 2.4.1, then $W(\mathbb{P}_r, \mathbb{P}_\theta)$ is continuous everywhere, and differentiable almost everywhere.*
3. *Statements 1-2 are false for the Jensen-Shannon divergence $JS(\mathbb{P}_r, \mathbb{P}_\theta)$ and all the KLs.*

Corollary 2.4.2.1. *Let g_θ be any feedforward neural network parameterized by θ, and $p_z(z)$ a prior over z such that $\mathbb{E}_{z \sim p_z(z)}[\|z\|] < \infty$ (e.g. Gaussian, uniform, etc.). Then Assumption 2.4.1 is satisfied and therefore $W(\mathbb{P}_r, \mathbb{P}_\theta)$ is continuous everywhere and differentiable almost everywhere.*

The corollary tells us that EM distance in theory is a very good choice of distance to minimize with neural networks.

Theorem 2.4.3. *Let $\mathbb{P}$ be a distribution on a compact space χ and $(\mathbb{P}_n)_{n \in \mathbb{N}}$ be a sequence of distributions on χ. Then, considering all limits as $n \to \infty$,*

1. *The following statements are equivalent*

- $\delta(\mathbb{P}_n, \mathbb{P}) \to 0$ *with* δ *the total variation distance.*
- $JS(\mathbb{P}_n, \mathbb{P}) \to 0$ *with JS the Jensen-Shannon divergence.*

2. *The following statemets are equivalent*
   - $W(\mathbb{P}_n, \mathbb{P}) \to 0$.
   - $\mathbb{P}_n \xrightarrow{D} \mathbb{P}$ *where* $\xrightarrow{D}$ *represents convergence in distribution for random variables.*
3. $KL(\mathbb{P}_n \| \mathbb{P}) \to 0$ *or* $KL(\mathbb{P} \| \mathbb{P}_n) \to 0$ *imply the statements in (1).*
4. *The statements in (1) imply the statements in (2).*

The previous theorems show that:

1. EM distance is a much more sensible cost function than at least the Jensen-Shannon divergence for the development of a GAN.
2. KL, JS, and TV distances are not sensible cost functions when learning distributions supported by low dimensional manifolds.

Proofs for them are depicted in [1].
As we saw in Theorem 2.4.3, Wasserstein distance is nicer to optimize with respect to Jensen-Shannon distance, but dealing with the infimum in its equation can be tough. To avoid this problem we can differentiate $W(\mathbb{P}_r, \mathbb{P}_\theta)$ by back-propagating through the Kantorovich-Rubinstein duality:

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)] \tag{34}$$

This intuition is proved in the following theorem(depicted and proved in [1]):

Theorem 2.4.4. *Let* $\mathbb{P}_r$ *be any distribution. Let* $\mathbb{P}_\theta$ *be the distribution of* $g_\theta(Z)$ *with* Z *a random variable with density* p *and* g_θ *a function satisfying*

Assumption 2.4.1. Then, there is a solution $f : \chi \to \mathbb{R}$ to the problem:

$$\max_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)] \tag{35}$$

and we have:

$$\nabla_\theta W(\mathbb{P}_r, \mathbb{P}_\theta) = -\mathbb{E}_{z \sim p_z(z)}[\nabla_\theta f(g_\theta(z))] \tag{36}$$

when both terms are well defined.

To find the function f that solves Equation 34, we train a neural network initialized with weights $w \in \mathcal{W}$, where $\mathcal{W}$ is a compact space, and then performing back-propagation through $\mathbb{E}_{z \sim p_z(z)}[\nabla_\theta f_w(g_\theta(z))]$, like in a normal GAN. In order to have the parameters w to lie in a compact space we use *weight clipping*, that consists in clamping the weights to a fixed box(for example $\mathcal{W} = [-0.01, 0.01]^l$). Now we show the WGAN procedure, as proposed in [1]:

Algorithm 5 WGAN

Input : α, the learning rate. c, the clipping parameter. m, the batch size. n_critic, the number of iterations of the critic per generator iteration. w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ *is not converged* **do**

  for $t = 0, ..., n\_critic$ **do**

    - Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data;
    - Sample $\{z^{(i)}\}_{i=1}^{m} \sim p_z(z)$ a batch of prior samples;
    - $g_w \leftarrow \nabla_w[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))]$;
    - $w \leftarrow w + \alpha \cdot RMSProp(w, g_w)$;
    - $w \leftarrow clip(w, -c, c)$;

  end

  - Sample $\{z^{(i)}\}_{i=1}^{m} \sim p_z$ a batch of prior samples;
  - $g_\theta \leftarrow \nabla_\theta[\frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))]$;
  - $\theta \leftarrow \theta - \alpha \cdot RMSProp(\theta, g_\theta)$;

end

2.4.5 Generative Adversarial Minority Oversampling

In [39], the authors propose a variation of the GAN, where the is an adversarial game with three players instead of two. The game is between a convex generator G, a classifier network M, and a discriminator D to perform oversampling. M plays an adversarial game against the generator G, that tries to generate fake samples that will be misclassified.

They have a training dataset $X \subset \mathcal{R}^D$, in a c-class classification problem. Let

P_i be the prior probability of the *i-th* class, where $i \in C = (1, 2, ..., c)$ is the set of classes. The authors used this method as an oversampling technique for multi-class dataset, while we used it in the context of fraud detection, so with only two classes.

Let X_i be the set of all n_i training point of class $i \in C$. They want to train the classifier M, that outputs, for all the classes, the probability of any $x \in X$ to belong to class i. The main idea is that the classifier will learn the boundaries between the classes by focusing on such difficult points near the fringes of the minority class. In the same way, the generator will be helped to generate points on those regions by the performance of M.

To avoid that G generates point that fall outside of the distribution of the minority class, they constraint it to generate point only as convex combination of existing points belonging to that class. Hence, G can generate new samples for class i as a convex combination of the data points in X_i:

$$G(\mathbf{z}|i) = \sum_{j=1}^{n_i} g_i(t(\mathbf{z}|i))x_j \tag{37}$$

where $\mathbf{z}$ is a latent variable drawn from a standard normal distribution and $x_j \in X_i$.

The adversarial game between M and G is the following, being p^i_{data} and p^i_g respectively denote the real and generated class conditional probability distributions of the *i-th* class.:

$$\min_G \max_M J(G, M) = \sum_{i \in \mathcal{C}} J_i \tag{38}$$

$$J_i = J_{i1} + J_{i2} + J_{i3} + J_{i4} \tag{39}$$

where $J_{i1} = P_i \mathbb{E}_{x \sim p^i_{data}}[logM_i(x)]$,
$J_{i2} = \sum_{j \in \mathcal{C} \setminus \{i\}} P_j \mathbb{E}_{x \sim p^j_{data}}[log(1 - M_i(x))]$,
$J_{i3} = (P_c - Pi)\mathbb{E}_{G(\mathbf{z}|i) \sim p^i_g}[logM_i(G(\mathbf{z}|i))]$, and
$J_{i4} = \sum_{j \in \mathcal{C} \setminus i}(P_c - Pj)\mathbb{E}_{G(\mathbf{z}|j) \sim p^j_g}[log(1 - M_i(G(\mathbf{z}|j)))]$.

We show now, as they prove in the supplement document of [39], the following theorem:

Theorem 2.4.5. *Optimizing the objective function J is equivalent to the problem of minimizing the following summation of Jensen-Shannon divergences:*

$$\sum_{i=1}^{c} JS\left((P_i p^i_{data} + (P_c - P_i)p^i_g) \Big\| \sum_{j=1, j\neq i}^{c} (P_j p^j_{data} + (P_c - P_j)p^j_g)\right) \tag{40}$$

The optimization problem in Theorem 2.4.5 tries to move the generated distributions for each class closer to the real distributions for all other classes. This results in generating points near the class boundaries, where it is critical for the learning.
Nevertheless, points can still be generated in locations within the convex hull of the corresponding class, but which don't fall inside its distribution. For this reason they add to the network an additional conditional discriminator, which goal is to make sure that the new samples do not fall outside of the actual distribution of the minority class.
The resulting network is characterized by the following three players adversarial game:

$$\min_G \max_M \max_D Q(G, M, D) = \sum_{i\in\mathcal{C}} Q_i \tag{41}$$

where $Q_i = (J_{i1} + J_{i2} + J_{i3} + J_{i4} + Q_{i1} + Q_{i2})$,
$Q_{i1} = P_i \mathbb{E}_{x\sim p^i_{data}}[logD(\mathbf{x}|i)]$, and
$Q_{i2} = (P_c - P_i)\mathbb{E}_{G(\mathbf{z}|i)\sim p^i_g}[log(1 - D(\mathbf{G}(\mathbf{z}|\mathbf{i})|i))]$.
Here we show the complete algorithm, as described in [39]:

Algorithm 6 GAMO

Input : X, training set. l, latent dimension. b, minibatch size. u, v, (hyperparameters, set to $\lceil \frac{n}{b} \rceil$ in their implementation)

Output: A trained classification network M

Note : For flattened images there is no need to train F , i.e., $F(X)$ can be replaced by X

while *not converged* **do**

  for u *steps* **do**

    - Sample $B_d = (x_1, x_2, ..., x_b)$ from X, with corresponding class labels Y_d;
    - Update F by gradient descent on $(M(F(B_d)), Y_d)$ keeping M fixed;

  end

  for v *steps* **do**

    - Sample $B_d = (x_1, x_2, ..., x_b)$ from X, with corresponding class labels Y_d;
    - Sample $B_n = (z_1, z_2, ..., z_b)$ from l dimensional standard normal distribution;
    - Update M and D by respective gradient descent on $(M(F(B_d)), Y_d)$ and $(D(F(B_d)|Y_d), 1)$, keeping F fixed;
    - Generate labels Y_n by assigning each $z_j \in B_n$ to one of the $c-1$ minority classes, with probability $\propto (P_c - P_i), \forall i \in \mathcal{C} \backslash \{c\}$;
    - Update M and D by respective gradient descent on $(M(G(B_n)), Y_n)$ and $(D(G(B_n)|Y_n), 0)$, keeping G fixed;
    - Sample $B_g = (z_1, z_2, ..., z_b)$ from l dimensional standard normal distribution;
    - Generate labels Y_g by assigning each $z_j \in B_g$ to one of the $c-1$ minority classes with equal probability. Take ones' complement of Y_g as $\bar{Y}_g$;
    - Update G by gradient descent on $(M(G(B_g|Y_g)), \bar{Y}_g)$ keeping M fixed.
    - Update G by gradient descent on $(D(G(B_g|Y_g)|Y_g), 1)$ keeping D fixed.

  end

end

2.5 Genetic Algorithms

Genetic Algorithms, which became popular thanks to *John H. Holland* in 1975 [25], are a class of searching algorithms that search for a nearly optimal solution for a given problem in a solution space. The search is performed by mimicking the genetic evolution: there is a "population" from which the best candidates are taken and bred to form a new generation of samples.
The population is a set of possible solutions. While the algorithms is running, new candidates are "born" in the population, while others "die". This process depends on the *fitness function*. The fitness function is a function that gives a measure of how good a solution is for the given task, and it grows along with the goodness of the solution.
The fundamental steps for a genetic algorithm are:

- *Population Generation:* generate the initial population.
  Each sample of the population is called *individual*, and each individual has a set of parameters(features) called *genes*. The set of genes is known as *chromosome*. In Figure 2.8 we can see an illustration of the population.

- *Fitness Function:* compute the fitness score for each chromosome in the population. The probability that an individual will be selected for reproduction depends on it.
  The fitness function is chosen with respect to the specific problem taken into account.

- *Selection:* select n chromosomes for the reproduction according to their fitness score .
  The selected chromosome are called *parents*, and there are many ways to pick the best ones. The most used methods are *roulette wheel selection*, where the probability of being picked is proportional to the fitness of each individual, *rank selection*, where the probability of being picked is proportional to the rank of each individual, and *tournament selec-*

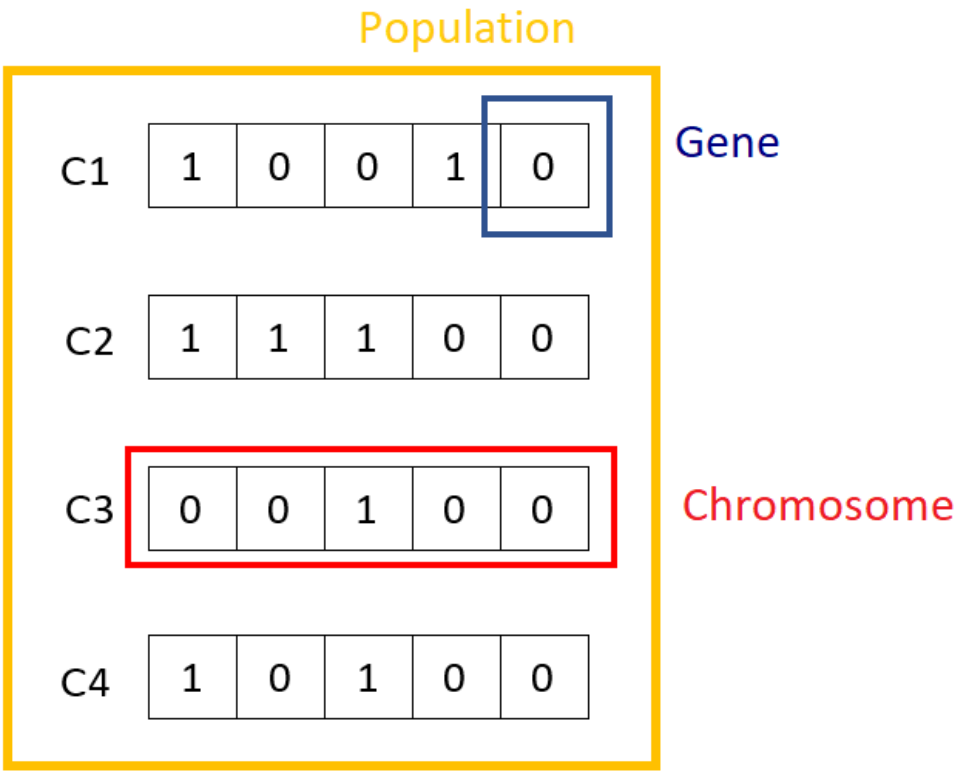

Figure 2.8: Illustration of population, chromosomes and genes.

tion, where a random subset of the population is taken into account and a tournament is held between the individuals.

- *Crossover*: mix the parents to form new individuals, with a crossover probability.
  The crossover operation consists in recombining genes of the parents in new samples. There are various operator to perform crossover, as *single point crossover*, where a random crossover point is taken at random to split the chromosome, and the new individuals will be formed by a part from the first parent and a part from the second one, and *uniform crossover*, where genes are randomly selected from one of the parents to create individuals.

Crossover works well because in selecting parents that contains good sequences, there is a probability that the process takes the best from each parents generating a better individual.

- *Mutation:* with a low mutation probability mutate some genes of the new offspring.
  Mutation is performed because with that the diversity within the population is maintained and premature convergence is avoided.

After the mutation step the new individuals are placed in the population instead of the individuals with the worst fitness score.
Fitness evaluation, selection, crossover and mutation are then repeated until a termination condition is met or a good enough solution is found.
Genetic algorithms are very versatile and customizable, and thanks to that they can be used for a large variety of problems. All the steps of the process can be modified and adjusted with respect to the problem, so a wide set of implementations can be produced with respect to the problem to solve.

One of the problems that we have to face in fraud detection is class imbalance, and one of the remedies for class imbalance is oversampling. When we want to oversample a class we want to find samples good enough to be assigned to that class.
As stated in [50] *"Genetic algorithms are particularly useful for problems where it is extremely difficult or impossible to get an exact solution, or for difficult problems where an exact solution may not be required."*. With oversampling we don't want an exact solution, but we want a lot of good enough solutions in order to balance the classes, and if we define a proper fitness function we can make use of genetic algorithm to produce samples that we can add to the minority class. If we repeat this process until we have enough samples, we can use genetic algorithms as an oversampling method.
In [31] we have an example of how this can be done. The authors of the paper propose an algorithm called *GenSample* for oversampling the minority class in imbalanced datasets. It was tested on 9 real-world imbalanced datasets

and it gave good results compared to SMOTE and ADASYN.
The algorithm takes as initial population the entire minority class. The fitness of a sample takes into account the type of sample(safe, boarderline, rare or outlier), and the performance improvement achieved by oversampling it. The fitness function is the following:

$$fitness(x) = \beta \times minority\_label\_weight + (1 - \beta) \times \Delta Fscore \tag{42}$$

where $0 < \beta < 1$, *minority_label_weight* is calculated with respect of the number of majority class samples in the k-neighborhood of x and $\Delta Fscore$ is the change in F_1 score produced by resampling x [31].
The selection is performed by picking the individual with the highest fitness score in the minority class as first parent, while the second parents is chosen randomly between the k nearest neighbors belonging to the minority class.
The crossover mechanism between the two parents is similar to the interpolation performed in [8]. At each iteration two children are produced, each one closer to one of the parents, following these equations:

$$child_1 = parent_1 + (parent_2 - parent_1) \times \lambda \tag{43}$$
$$child_2 = parent_1 + (parent_2 - parent_1) \times (1 - \lambda) \tag{44}$$

where $0 < \lambda < 1$.
After the reproduction and the evaluation of the fitness of the two children, the fitter one replaces the least fit individual in the population, while the other one is only put in the dataset because he's not considered a good individual for reproduction.
Finally, in the paper, they address as mutation the selection, with a small probability, of a random individual from the population as first parent instead of the fittest one.
The steps are performed until the balance of the classes is reached or until

adding a new sample results in a degradation in performance.
The results presented in the paper show that this algorithm outperforms the other oversampling techniques for most of the datasets.

Chapter 3

Research Problem

3.1 Problem Formulation

In this section we provide a formal description of the alert-feedback interaction and give a description of the classification problem we face in our work. The goal of our research is to develop a system for fraud detection of credit cards transactions. As stated before, the framework we used is taken from [12]. Each transaction in the dataset is denoted as a vector $\{x_i, y_i\}$, where x_i is a *k-dimensional* feature vector associated with the *i-th* transaction, while y_i is the label of the *i-th* transaction. We have that $y_i \in \{0, 1\}$, and we denote transaction i as fraud if $y_i = 1$, and as genuine otherwise.
In our framework the goal is to train a classifier $\mathcal{K}$ in order to predict when a transaction is fraudulent or genuine. $\mathcal{K}$ is retrained every day, and we denote with $\mathcal{K}_{t-1}$ the classifier that has been trained on transactions available up to day $t-1$. The classifier $\mathcal{K}_{t-1}$ is then used to classify the transactions authorized at the following day, t. We train a classifier everyday because in a real-world model for a FDS, data are coming as a transaction stream, and the stream is affected by concept drift. By concept drift we mean an online learning scenario in which the relationship between the input and the target vector is changing over time [15]. In the case of fraud detection we

have to face this problem because fraudsters can study the real distribution of the transactions and adapt their methods accordingly. They can change their behaviors according to the seasonality (e.g. during Christmas period the number of transactions is higher with respect to the rest of the year), to new techniques for the detection, or to some particular event.
$\mathcal{K}_{t-1}$ returns us, for every transaction x_i, the posterior probability for the transaction to be a fraud, $P_{\mathcal{K}_{t-1}}(1|x_i)$. As stated before when describing a real-world FDS, the alerted transactions are checked manually by human investigators, that have a limited amount of time every day. For this reason only the *k-th* most risky transactions are alerted every day, and they are defined as:

$$A_t = \{x_i \in T_t | r(x_i) \leq k\}$$

where T_t is the set of transactions authorized at day t, and $r(x_i) \in \{1, ..., |T_t|\}$ is the rank of the *i-th* transaction, according to its posterior probability.
The alerts, once that the investigators make their checks, are added to the Fraud Detection Systems as feedbacks, and they are treated separately from the rest of the transactions. In a real-world FDS, feedbacks are the only supervised information available in the short term, and they are not representative of the total distribution of the transactions. They are taken from the k most risky transactions, hence they are affected by a problem known as *Sample Selection Bias* [11], that is present when we choose a subset of the data not in a randomic way.
The set of the feedbacks is modeled as follows:

$$F_t = \{(x_i, y_i) | x_i \in cards(A_t)\}$$

where $cards(A_t)$ is the set of cards for which at least a transaction has been alerted.
After a certain amount of time, that in here we assume is fixed and denoted by

δ(*verification latency*), the clients have spotted all the corresponding frauds, while the non-disputed transactions are labeled as genuine, hence all the transactions are available to the FDS with their own label. In this way at day t all the transaction up to day $t - \delta$ are available with their labels. The set of delayed samples is modeled as:

$$D_{t-\delta} = \{(x_i, y_i) | x_i \in T_{t-\delta}\}$$

From these definitions it follows that $F_{t-\delta} \subset D_{t-\delta}$.

3.2 Workflow Schema

Now that we defined the scheme of our training sets, we will present the workflow of our framework. As already stated in Section 2.3, the classifier K_t is the aggregation of two different classifiers. One is trained exclusively on the feedbacks, and one is trained exclusively on the delayed samples. As we said before, this is done because the distribution of the two datasets is very different, since the feedbacks are affected by the Sample Selection Bias, and they depends on the performance of the classifier, while the delayed samples are highly unbalanced in favor of the genuine class.
From the study of the literature we understood that Random Forests are the models that achieve the best performance [4] [42] [51] for this kind of task, so we will use this model as baseline for our classifiers, with number of trees depending on the experiment.
The feedback classifier

$$\mathcal{F}_t = TRAIN(\{F_t, ..., F_{t-\delta}\})$$

is trained on the feedbacks provided from the investigators in the last δ days.

The delayed classifier

$$\mathcal{D}_t = TRAIN(\{D_{t-\delta}, ..., D_{t-(\delta+M-1)}\})$$

is an ensemble of M random forests, where $M = Q - \delta$, where Q is the time window on which the daily classifier is trained. Each random forest is trained on the whole set of delayed samples of a specific day, and then the results are aggregated by averaging the scores of the M classifiers. Being $\mathcal{D}_t = \{\mathcal{M}_{t-Q}, ..., \mathcal{M}_{t-Q-M}\}$ the ensemble learned at day t, we have that the posterior probability of transaction x being a fraud for the ensemble is:

$$\mathcal{P}_{\mathcal{D}_t}(1|x) = \frac{\sum_{i=t-Q-M}^{t-Q} \mathcal{P}_{\mathcal{M}_i}(1|x)}{M} \tag{45}$$

where $\mathcal{P}_{\mathcal{M}_i}(1|x)$ is the posterior probability of x being a fraud, given as output from the individual classifier i.

The daily datasets are extremely unbalanced since the fraudulent transactions are way outnumbered by genuine transactions. Also the feedback datasets could be affected by imbalance in the opposite way, but we didn't focus on them in this thesis. Class imbalance leads traditional machine learning methods to focus their attention on samples belonging to the majority class, resulting in very poor performance on the minority class. We try to solve this problem by performing oversampling of the minority class on the delayed datasets.

In practice, at every day t, the transactions authorized are stored in a new dataset for the corresponding day, and on this dataset oversampling is performed before training the corresponding model. We compare in our work different oversampling methods (already described in Section 2.4): ROS, SMOTE, WGAN oversampling, GAMO oversampling and oversampling with Genetic Algorithm. Additionally, we propose a new oversampling technique

based on genetic algorithms. In the comparison we also add a framework in which we don't perform any kind of oversampling, leaving the dataset as it is.

At day t, after the training of $\mathcal{F}_t$ on the feedback collection and of $\mathcal{D}_t$ on the oversampled daily datasets, we aggregate the results of the predictors in an unique classifier, $\mathcal{K}_t$, that gives us the posterior probability of transaction x being a fraud, as:

$$\mathcal{P}_{\mathcal{K}_t}(1|x) = \alpha\mathcal{P}_{\mathcal{F}_t}(1|x) + (1-\alpha)\mathcal{P}_{\mathcal{D}_t}(1|x) \tag{46}$$

where $0 < \alpha \leq 1$ is the weight parameter that balances the contribution of both the classifiers. In our experiments we chose $\alpha = 0.5$, that corresponds to the average between the two predictions. The aggregate classifier $\mathcal{K}_t$ is then tested on T_t, that is the set of transactions authorized at day t.

From Figure 3.1 we can better understand how the workflow pipeline of our system works. Every day an ensemble of Random Forests is trained on the set of Delayed Datasets, and a Random Forest is trained on the Feedback Dataset. The two scores obtained are then aggregated and used to compute the final predictions. Then, the 100 most risky transactions are checked by the investigators, and once this operation is done they will replace the feedbacks relative to the oldest day in the Feedback Dataset.

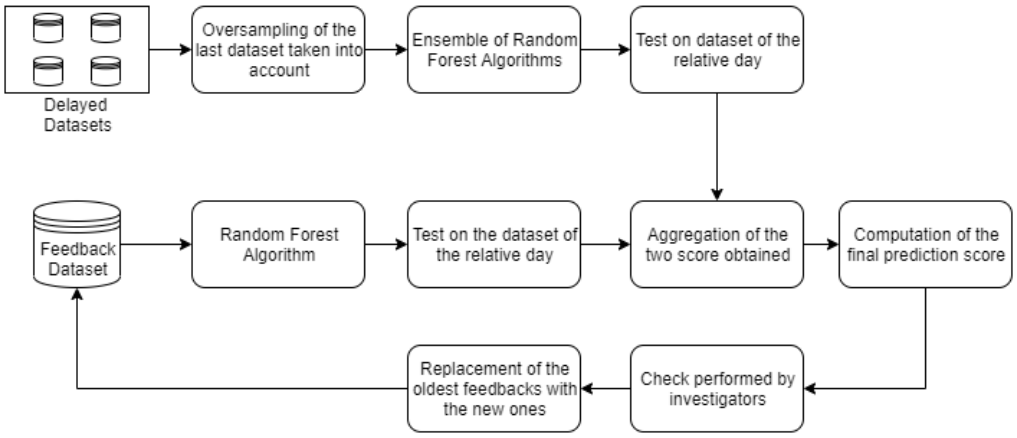

Figure 3.1: Daily Workflow Schema. Every day two models are trained, an ensemble of Random Forests for the Delayed Datasets and a Random Forest for the Feedback Dataset. Then the results are aggregated and the predictions are made. The 100 most risky transactions are then checked from the investigators, and then the feedbacks are added to the Feedback Dataset for the following day.

3.3 Genetic Algorithm Solution

In this section we present the oversampling algorithm we developed based on genetic algorithms. This algorithm is the one that gave us the best performances overall in the experiments. This is probably due to the fact that this method has been created specifically for our problem, exploiting the characteristics of our dataset.
Now let's analyze how this algorithm was developed based on the steps of a genetic algorithm:

- *Population Generation*: the initial population in our case is formed by the fraudulent transactions in the dataset. Considering how we developed our solution, the oversampling was performed for each hour dataset, hence the initial population is different for everyone of them.

- *Fitness Function*: the fitness function is based on the feature importance computed in [18]. As we can see from Figure 3.2, the most

important features are $V14$, $V4$, $V10$, $V17$ and $Time$. The importance of the features is obtained with XGBoost, that provides a method to compute it. The importance is calculated for each attribute for every single tree as the amount of improvement in the performance that an attribute split point gives. Then the feature importance are averaged across the decision trees.
The formula of the fitness functions is the following:

$$\begin{aligned} fitness(x) = w_1 \times x(V14) + w_2 \times x(V4) + w_3 \times x(V10) \\ + w_4 \times x(V17) + w_5 \times x(Time) \end{aligned} \tag{47}$$

where w_i is the weight relative to the *i-th* feature on the $Fscore$, computed as:

$$w_i = \frac{Fscore_i}{\sum_{j=1}^{5} Fscore_j} \tag{48}$$

The weight is computed only on the features taken in consideration for the computation of the function.

- *Selection*: the selection for the mating partners is done through the fitness function. The parents will be the best four chromosomes according to their fitness. In some of the datasets there were less than four frauds, hence it was not possible to choose four parents. To solve the problem the original frauds from the previous dataset were added to the ones of the current dataset.

- *Crossover*: the operator we choose for crossover is the *single point crossover*. Having 30 features, we took 15 from the first parent, and the rest from the second one. The crossover was performed on four couples from the parents, formed by the first with the second, the second with the third, the third with the fourth, and the fourth with the first. From each mating two children were formed, one for each possible combination.

- *Mutation*: the mutation was performed by adding a random value in the interval $(-1, 1)$ to a random attribute of the offsprings.

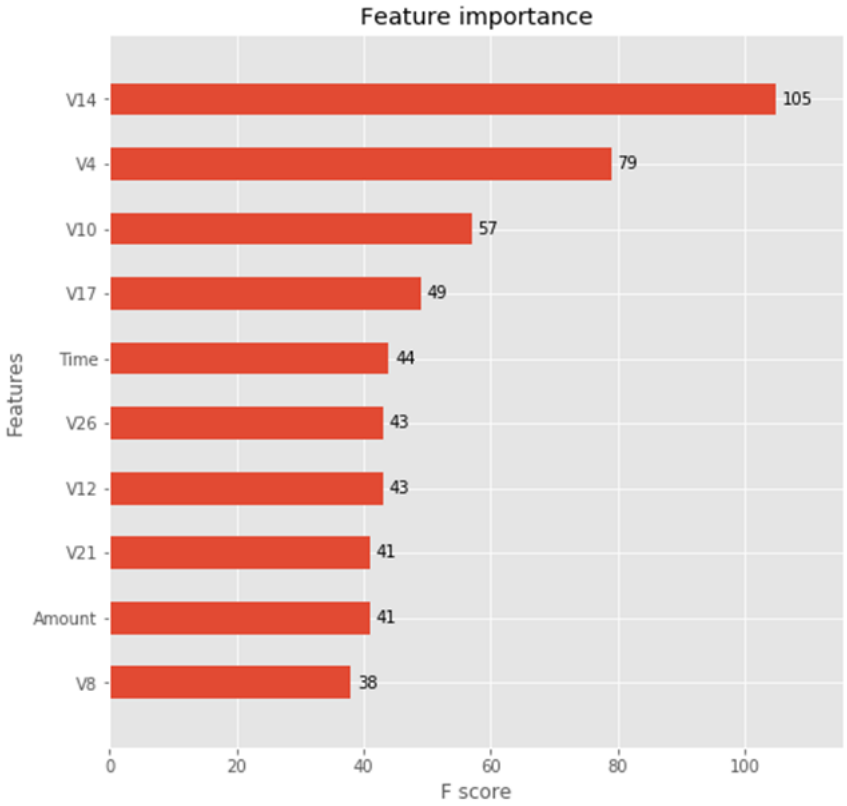

Figure 3.2: Feature Importance. The picture is taken from [18]

After each iteration of the genetic algorithm the children that are fitter than the worst individuals in the original population are added both to the dataset and to the population for the next round of the genetic algorithm. The algorithm is stopped when the desired balance is reached, in our case when the number of frauds is equal to the number of genuine transactions. This algorithm gave us competitive results for the problem we considered, resulting in the best performance overall.

Chapter 4

Experiments

In this section we will describe the details of our implementation and of our experiments with the tools and the software we used, as well as an implementation in PySpark, to leverage Big Data.

4.1 Tools And Software

All our experiments have been written in *Python 3.6* [46], one of the most used programming languages for machine learning application, using JupyterLab [29], a web-based interface to handle Python notebooks.
To implement our solution, we mainly made use of the following libraries:

- *Pandas*, that is a library used for data manipulation and analysis, offering data structures for numerical tables and time series, and a set of operations and functions to manipulate them. We used Pandas widely to deal with DataFrame objects and to perform operations on them. [38]

- *MatPlotLib*, that is a multi-platform data visualization library in Python for 2D plots of arrays, built on NumPy arrays and designed to work

with the broader SciPy stack. We used MatPlotLib to create most of our plot and graphics. [26]

- *SciKit-Learn*, that is one of the most widely-used Python libraires for data science and machine learning. It provides you with many operations and algorithms, and it's built on NumPy, SciPy and MatPlotLib. We used SciKit-Learn for the implementation of our machine learning algorithms(e.g. Random Forest) and for the computation of metrics of relevance. [41]

- *NumPy*, that is a library that adds support for arrays and matrices, and that provides us with mathematical functions to operate on these objects. We used NumPy mostly when dealing with arrays in the GAMO implementation. [40]

- *Tensorflow*, that is an end-to-end open source platform for machine learning. We used it to support the develepement of our neural network architectures. [37]

- *Keras*, that is a high-level neural networks API, capable of running on various platforms. We used it on top of Tensorflow, and we used it to develop our neural network architectures. [10]

For the PySpark [45] implementation, we made use of the platform *DataBricks* [13], a platform for big data processing founded by the creators of *Apache Spark* [54]. We used a cluster available with the *Community Edition* of DataBricks, with 15.3 GB of memory and 2 cores, optimized for machine learning applications.
The libraries we used, apart from the ones already described, were mainly two:

- *pyspark.sql*, a class of algorithm for SQL and DataFrame implementation, that we mainly used for the creation and the manipulation of the

dataframes, and the operations on them. [2]

- *pyspark.ml*, a DataFrame-based machine learning API. We used this libraries to implement our classifications methods. [44]

4.2 Architecture Schema

As we can see from Figure 4.1, these are the main components of our architecture:

- the *Datalake*, that is the centralized repository of our architecture. This component allows us to store structured and unstructured data at any scale. For the nature of our problem, in a real setting, we would have a massive and continuous stream of data, making this solution ideal for our needs. The Datalake communicates with most of the other modules of the architecture, sending and receiving data.

- the *Data Preprocessing module*, that is the module responsible for the preprocessing of the data. The operation performed by this module can be data cleaning, data editing, feature aggregation and all the operation related to the preprocessing part. The Data Preprocessing module takes data from the Datalake and returns them cleaned.

- the *Delayed Datasets*, that we already described in the previous chapter, are the datasets relative to the transactions of the set of days taken into account for the ensemble of classifiers, for which we already have labels. This module communicates with the Datalake to take the transactions of interests, and with the Oversampling module and with the Delayed Classifier.

- the *Oversampling module*, that is responsible for the oversampling operation on the Delayed datasets, that are higly unbalanced. The operations perfromed depend on the oversampling technique used. This

module takes the data from the Delayed Datasets and return the relative balanced datasets.

- the *Delayed Classifier*, that is the ensemble of classifier for the labeled data coming form the Delayed datasets. This module performs the training on the labeled data and the predictions on the test set for every classifier in the ensemble and then aggregate their results. It takes the data from the Delayed Datasets and it returns the final predictions to the Aggregate Predictor.

- the *Feedback Classifier*, that is the classifier used for the feedbacks, that as already explained are the transactions checked by investigators. This module performs the training on the feedbacks and the predictions on the test set. It takes the data from the Feedback Datasets and it returns the predictions to the Aggregate Predictor.

- the *Aggregate Predictor*, that is the modules responsible for the aggregation of the two types of predictions and of the computation of the relevant metrics for the evaluation. The aggregation is done by averaging the two results coming from the Delayed Classifier and from the Feedback Classifier. This module, after the aggregation part, send the predictions to the Feedback Collector.

- the *Feedback Collector*, that is the module that has to check the predictions made by the classifier, correct them if necessary and update the Datalake with the labels. This component also has to check the 100 most risky transactions, in order to update the Feedback Dataset. This module takes the predictions from the Aggregate Predictor and it returns the results of the operations to the Datalake and the Feedback Dataset.

- the *Feedback Dataset*, that is the dataset relative to the feedbacks, relative to the most risky transactions checked by investigators relative

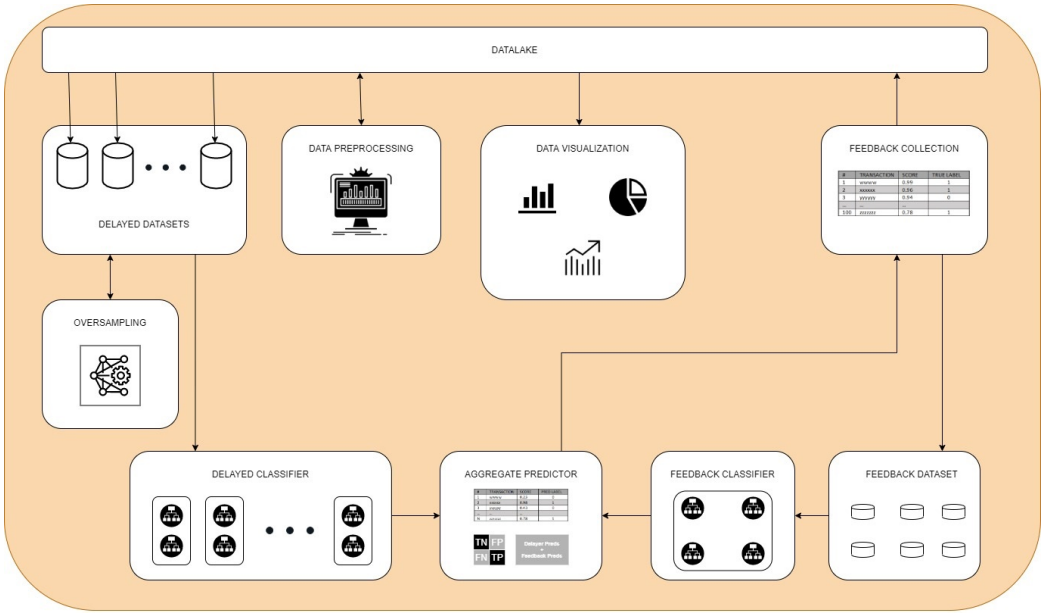

Figure 4.1: Architecture of the System. As we can see from this picture the architecture is composed of various components that communicate between each other in order to guarantee the right workflow of the execution. In this picture we can see the dependences between the modules and how they interact with each orther.

to the previous few days. This module takes the data from the Feedback Collector and then is asked for them from the Feedback Classifier.

- the *Data Visualization module*, that is the module responsible for the production of charts, plots, and graph relative to the data coming form the Datalake, that is the only component of the architecture that communicates with it.

In Figure 4.2 we can see an example of how the different components interact at runtime between each other. As we can see the process starts with the data preprocessing from the data in the Datalake and the subsequent division in daily/hourly dataset. The next step is the Delayed Dataset component taking the dataset from the Datalake and performing oversmapling on them. Then the Aggregate Classifier asks for the predictions to the Delayed and the Feedback classifiers. The last one has to take it from the Feddback

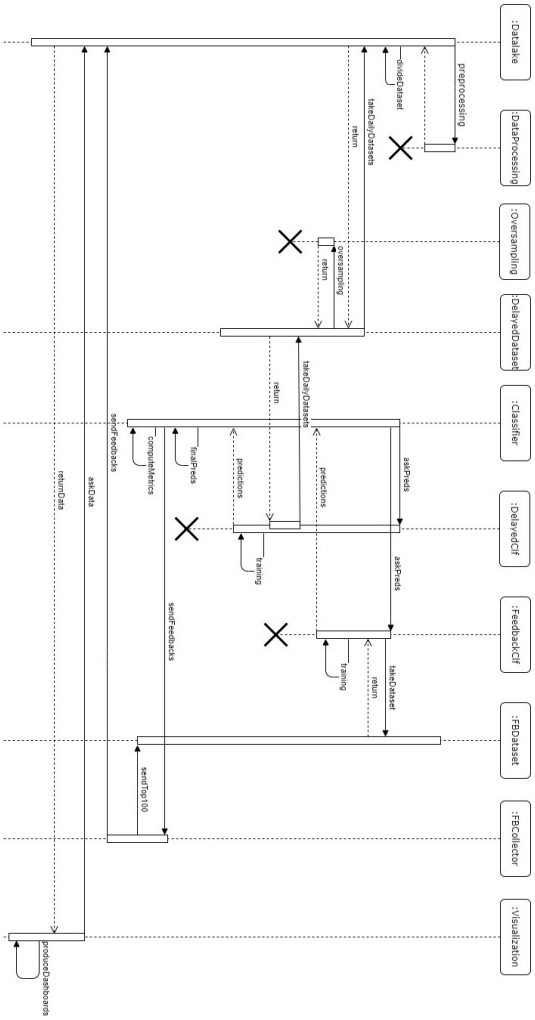

Figure 4.2: Sequence Diagram.

Dataset, and then both the classifiers perform the training and return the results to the Aggregate Classifier. At last, the final predictions and the metrics are computed, and the feedbacks are sent to the Feedback Collector, that sends the 100 most risky transactions to the Feedback Dataset and the whole set of the feedbacks to the Datalake. Then, eventually, dashboards and visualization plots are computed if needed.

4.3 Implementation Of The Proposed Solution

As already said in the previous section, we conducted our set of experiments on a dataset available on the *Kaggle Repository* [30].
The first experiments were done considering the whole dataset without implementing it into the framework as done later.
Before running any algorithm there was a step in which we converted the *Time* variable from seconds elapsed from first transaction to time of day in hour. This was done because otherwise almost every sample would have had a different value. This kind of features can lead to bad performance on decision trees, because at some step they could split on this feature with high probability, even if it is not informative at all, being different for almost every transaction.
The procedure in this part, after dealing with the feature *"Time"*, has been the following: 1) the dataset has been split in *train set* and *test set*, taking respectively 80% and 20% of the data(splitting separately fraud and genuine transactions); 2) the train set, that contains 227452 genuine transactions and 393 fraudulent ones, has been over-sampled in order to make it balanced; 3) a random forest has been trained on the train set and tested on the test set; 4) performance metrics have been computed.
Different over-sampling techniques have been used in this phase: ROS, oversampling with WGAN, oversampling with GAMO, and oversampling with our Genetic Algorithm. The results have been compared between each other and then also with the results of the experiments without the over-sampling phase.
In this phase we didn't implement the *Alert Precision* and *Recall* on the top 100 risky transactions(referred later as *Top100 Precision* and *Top100 Recall*), that has been implemented later, being these just an explorative stage of our work.
For what concerns the experiments using ROS, the procedure has been triv-

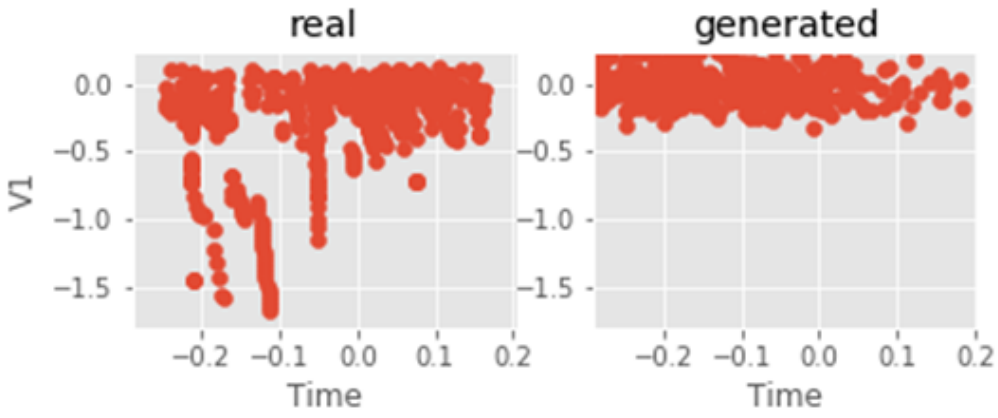

Figure 4.3: In this plot we can see the real distribution of the samples with respect to the generated ones at the first iteration of the learning algorithm. As we can see the generation is done with very poor quality, being one pig cluster of data.

ial, since the python library *imblearn* [34] provides us with a method for doing so, while in using WGAN and GAMO oversampling we had to do some background work. Finally for what concerns the Genetic Algorithm the implementation was the one described earlier (DA CONTROLLARE). For WGAN oversampling we have have followed a procedure similar to the one described in [18]. In particular we have used widely the *GAN_171103* repository, for the creation and the training of the GAN models, and for the generation of the synthetic data. The procedure has been the following:

1. Create WGAN model(discriminator and generator).
2. Train on the fraudulent transactions for 500 steps.
3. Generate fraudulent transactions in order to make the dataset balanced.

In Figure 4.3 and in Figure 4.4 we can see how the generation of the data changes after various iterations of the learning algorithm. At first it is really poor, resulting in one big cluster of points, while going on with the learning the model begins to capture the distribution of the data, and the generation improves its quality.

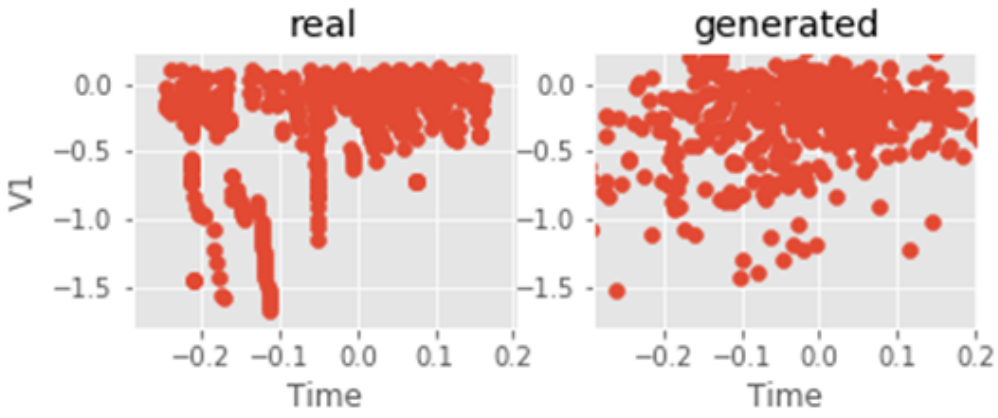

Figure 4.4: In this plot we can see the real distribution of the samples with respect to the generated ones at the 500th iteration of the learning algorithm. As we can see the generation is more precise, and it is starting to capture the real distribution of the data.

For what concerns GAMO oversampling, we used the code provided with [39], available at the GitHub page of the author [16]. In particular we made use of the modules *dense_net* and *dense_suppli*, in order to create the model and generate the data, while we have re-adapted the code in *dense_gamo_main* in order to make it work with data that are not pictures, given that the one provided was specifically developed for dataset of images. The procedure has been the following:

1. Initialize the model creating generator, discriminator, and additional classificator(multi-layer perceptron).

2. Train the model for 5000 steps.

3. Generate fraudulent transactions in order to make the dataset balanced.

After the oversampling phase, performed after that the dataset has been split into train set and test set, the generated samples have been concatenated to the original transactions in the train set. Then a Random Forest with a variable number of trees has been trained on the train set and then tested

on the test set. The number of trees has been chosen depending on the over-sampling algorithm employed.

In [17] the author showed the distribution of the features for genuine and fraudulent transactions after that these have been normalized, as we can see in Figure 4.5. We can see how for most of the features genuines and frauds can be easily distinguished, while for some of them there are some overlapping. For this reason we decided to perform a set of experiments excluding these overlapped features. The features that have been excluded are $V13$, $V15$, $V22$, $V24$, $V25$ and $V26$. Excluding these features didn't show significant improvement in the performances, but on the contrary, they worsened them, as we can see in Table 5.1.

The experimental setting was further improved in this phase by implementing at first the *Top100* metrics, and then the framework as the authors described in [12], that we will depict in detail in Chapter 5.
For the metrics, the procedure was straightforward. Random Forests are probabilistic models, hence they provide you with the probability of a sample from the test set to be in a specific class. The probability given as output, in our case, is the probability for the transaction to be a fraud. In order to evaluate our results on the 100 most risky transactions, we ordered the transactions in descending order by the output probability, and then computed the metrics on the first 100 transactions for what concerns *Top100 Precision* and *Top100 Recall*, while on the whole test set for what concerns AUC.

The last step for this set of experiments was the implementation of the framework for the validation of our results. The first thing we did was the division of the dataset in daily transactions. The Kaggle dataset we used for this was composed of a set of transactions covering 48 hours, making it impossible to divide it on a daily basis. To overcome this limitation, we divided the dataset on an hour basis in order to keep the functioning unaltered. The result of

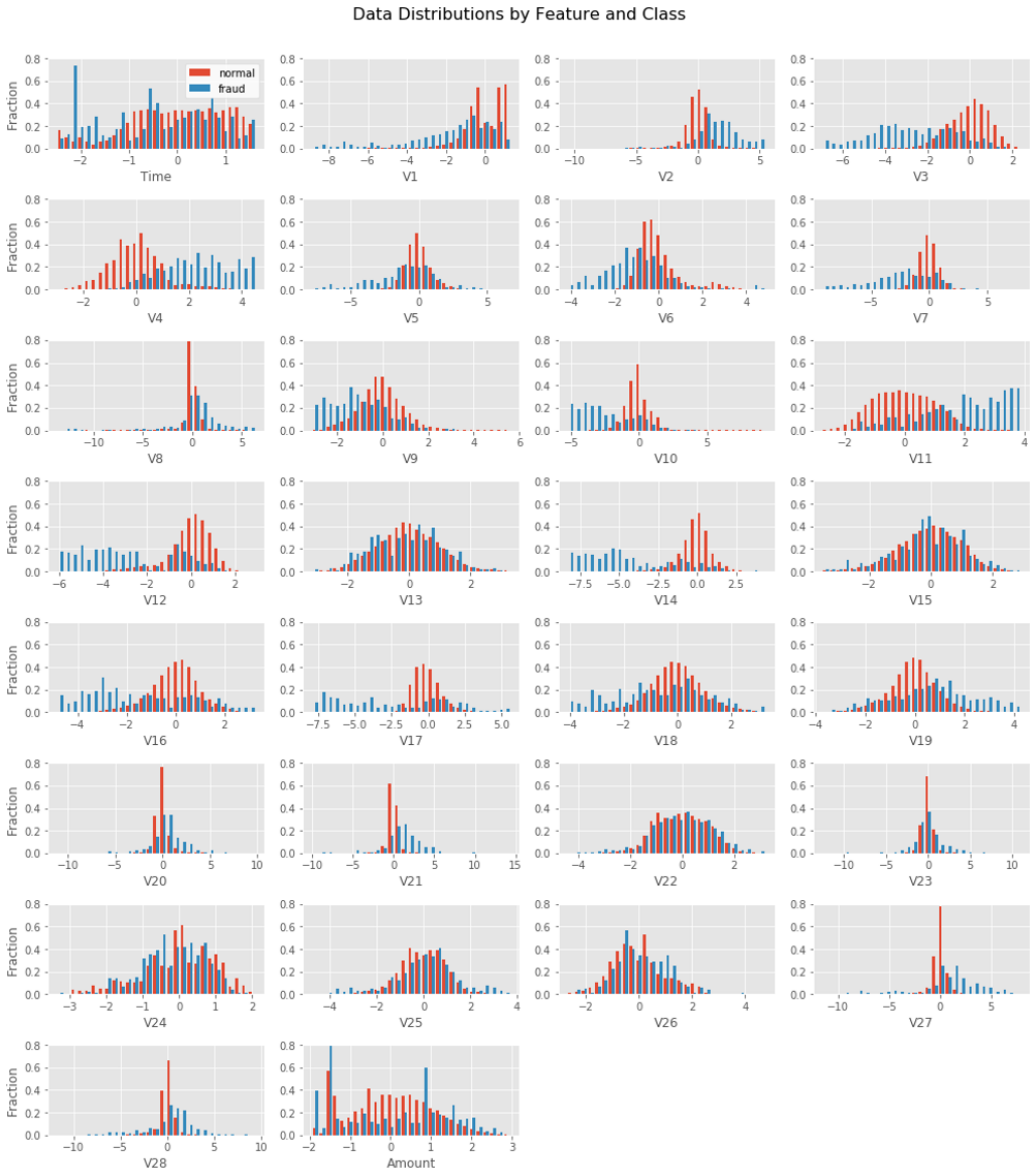

Figure 4.5: The distribution of the data by feature and class shows us how for most of the features the two classes can be easily separated, while for some of them(V13, V15, V22, V24, V25, V26) there is a marked overlapping that could affect predictions. Pictures taken from [18].

this division was a set of 48 dataset, one for every hour covered, on which we tested the framework.

We choose as hyperparameters of our framework $M = 8$ and $\delta = 7$, hence we have $Q = M + \delta = 15$. At the beginning of the training we didn't have any feedback for this phase, because no prediction has been performed yet. To initialize the feedback collection, 8 random forests have been trained on the first 8 hour datasets, and tested separately on every dataset from the ninth to the fifteenth, since we set as hyper-parameter the number of hours for the feedback dataset equal to 7.

Once that the initialization step was concluded our training procedure started. At every step of the learning process we trained our model on the feedback dataset, that contains feedbacks for the last 7 hours of transactions, and on the delayed datasets relative to the last 8 hours before the start of the feedbacks.

The random forests relative to the deleyed datasets have been applied after oversampling has been performed on each dataset, and the results have been compared with each other and with the results obtained without oversampling. The oversampling methods we used are ROS, GAN, GAMO and Genetic Algorithm.

For the oversampling procedure one problem that we met was the absence of fraudulent transactions in some specific hours. This was problematic because when we didn't have any fraudulent case, we couldn't use any sample as examples for the generative methods, making it impossible to perform oversampling. To overcome this problem we appended to the dataset the fraudulent transactions of the original dataset relative to the previous hour. This worked well since the transactions appended were close enough in time to the relative hour to be considered in the same dataset.

Testing is performed at every step of the learning procedure on the hour after the hour of the last feedback of the collection, and at the end of the procedure the results of every hour are averaged together to get the final performance. The metrics used in this phase of our experiments were *Top100 Average Precision*, *Top100 Average Recall*, *Average AUC* and *Average Detection Rate*. We couldn't make use of the *Card Precision* metric as they did in [12] since

in the Kaggle dataset the data are anonymized, and we didn't have any information regarding the cardholders or about the numbers of the cards.

After this phase was concluded, we conducted more experiments on the Genetic Algorithm oversampling, given that it gave us the best results, as we can see from Table 5.3.
We performed tuning experiments on the random forests relative to the delayed datasets in order to improve our performances.

We made the tuning by hand, trying different settings for the parameters, reaching the best performances for this settings: $n\_estimators = 60$, $max\_features = 10$, $max\_depth = 10$, $criterion = entropy$ for the Genetic Algorithm oversampling, and $n\_estimators = 60$, $max\_features = 15$, $max\_depth = 10$, $criterion = gini$ for GAMO(we invite to consult [47] for the documentation relative to random forests and their attributes).
We also changed the classifiers for the ensemble on the delayed datasets. We tried to use Neural Networks and XGBoost [9], and to compare their results with the one already obtained.
For what concerns Neural Networks, after a tuning phase we found our best solution in a network composed from 1 input layer, 11 dense hidden layers and 1 output layer. The hidden layers had respectively 20, 40, 60, 80, 100, 120, 140, 160, 120, 80 and 40 neurons. Between the fourth and the fifth layer we inserted a dropout layer, with dropout rate of 0.2. All the layers had *Scaled ELU* [33] as activation function, except for the output layer that used the *sigmoid* function. The optimizer used was *AdamOptimizer* [32] and the loss function *binary cross-entropy*. We also tried different settings for the number of epochs and for the batch size, achieving the best results for 50 epochs and a batch size of 200.
In the experiments with XGBoost the best setting was: $learning\_rate = 0.1$, $n\_estimators = 200$, $max\_depth = 7$ and $min\_child\_weight = 3$(we invite to consult [53] for the documentation relative to XGBoost and its attributes). The parameters were found through tuning by hand, performing various ex-

periments. XGBoost was the classifier that reached the best performances in terms of AUC, with this particular setting being the best ever for this part of our activity, as we can see in Table 5.4.
For both algorithms, the tuning of the parameters has been performed on the whole dataset, and the settings that achieved the best scores were then used on the framework. For what concerns neural network, on the framework's experiments different number of epochs and batch size were tested.

4.4 PySpark Implementation

In this section we will describe the Spark ecosystem and why it is widely used for machine learning applications, and we will present our PySpark implementation.
Spark [54] is an open source framework for distributed computing that can process operation on very large datasets very quickly. This is done through the usage of cluster computing, meaning that it can use resources from many processors all together for its operations, that are divided in multiple jobs.
The jobs are managed by a centralized unit, the *driver*, that schedules the code in multiple tasks and distributes them to one or more nodes, where they are executed by one or more *executors*.
Some of the most used programming languages(e.g. Python, Java, Scala and R) are supported by Spark, and are furnished with libraries for different tasks like SQL, streaming and machine learning. For this reason Spark is a solution really easy to start with, that is able to scale up to Big Data environments on a vary large scale and that is able to tackle a lot of machine learning problems.
We chose to include Spark in our experiments because of the nature of our problem: in a real world application there would be millions of incoming transactions every day, and with traditional machine learning techniques it would be difficult to manage this huge amount of data. On the contrary, with the usage of Spark we can easily scale up to a Big Data context, as it

is in our case, without a dropout in computational performances.
As already stated, for our Spark implementation we used Databricks, that is a cloud based Apache Spark cluster service developed by the creators of Spark, that offers scalable clusters and access to all Spark modules. It is also provided with a cluster manager, a jobs scheduler, and it has data visualization capability, reason why it is widely used in machine learning applications. In Databricks we can use notebooks in either Scala, Python or SQL. Given that we developed our solution in Python, we used this kind of notebook with the support of PySpark to implement it. PySpark is a Python API to support Apache Spark, and we chose to use it mainly because of its natural integration with Python code, that was the language we used for our previous experiments. We used it also because of the presence a dedicated library for machine learning applications, furnished with the most used algorithms and techniques.

For what concerns the practical implementation, what we did was to convert the Python code we already wrote in PySpark in the parts in which it was required. We used PySpark for the operations on the datasets and for the classification task relative to the delayed datasets. For the classification relative to the feedbacks, Spark was not used since the length of the dataset was fixed and not high enough to appreciate the enhancement in the performances. For what concerns the oversampling part, PySpark doesn't have built-in methods for that, hence it was done through the *imblearn* library as in the normal implementation.
The main difference between the normal implementation and the PySpark implementation is the type of DataFrame used: in the first one we used Pandas DataFrame, while in the second one we used PySpark DataFrame. The main difference between these two types of DataFrame is that in Pandas they are stored in memory, while in PySpark they are distributed on your cluster. It follows that PySpark DataFrame can leverage a bigger amount of data. They are also more difficult to manipulate, and most of the work was about

how to translate operations from Pandas.
For the classification part we tested only the Random Forest implementation. We made use of the library for classification available with PySpark(*pyspark.ml.classification*). The main difference between this implementation and the one with sklearn is the type of data that can be passed to the functions. While with sklearn you can pass a DataFrame directly, with PySpark this is not possible and the dataset has to be vectorized first. This basically means that every record has to be transformed in a vector of features to be passed to the classifier fro training and testing.

Chapter 5

Results And Evaluation

5.1 Dataset

In our work we conducted a first set of experiments on a dataset available on the *Kaggle Repository* [30]. The dataset has been provided by the *Machine Laarning Group* of the *Université Libre de Bruxelles*. It contains $284,807$ transactions, with 492 frauds, that account for 0.172% of the total. Figure 5.1 shows us the distribution of the two classes, which are very skewed in favor of the genuines.

All the features of the original space, except for "Time" and "Amount", have been transformed through a PCA. They are called $V_1, V_2, ..., V_{28}$, and they are the result of the transformation, in form of numerical inputs.

"Time" represents the seconds elapsed between a transactions and the first one in the dataset, "Amount" is the amount of the transactions, while all the other features are kept secret for confidentiality issues. The labels of the transactions is denoted by the feature "Class", which is equal to 1 if it's a fraud and equal to 0 otherwise.

The dataset has been mainly used for experiments at the beginning of the work in order to assess the metrics, the techniques and the framework. For the framework, given the limited number of days that the dataset covers, we

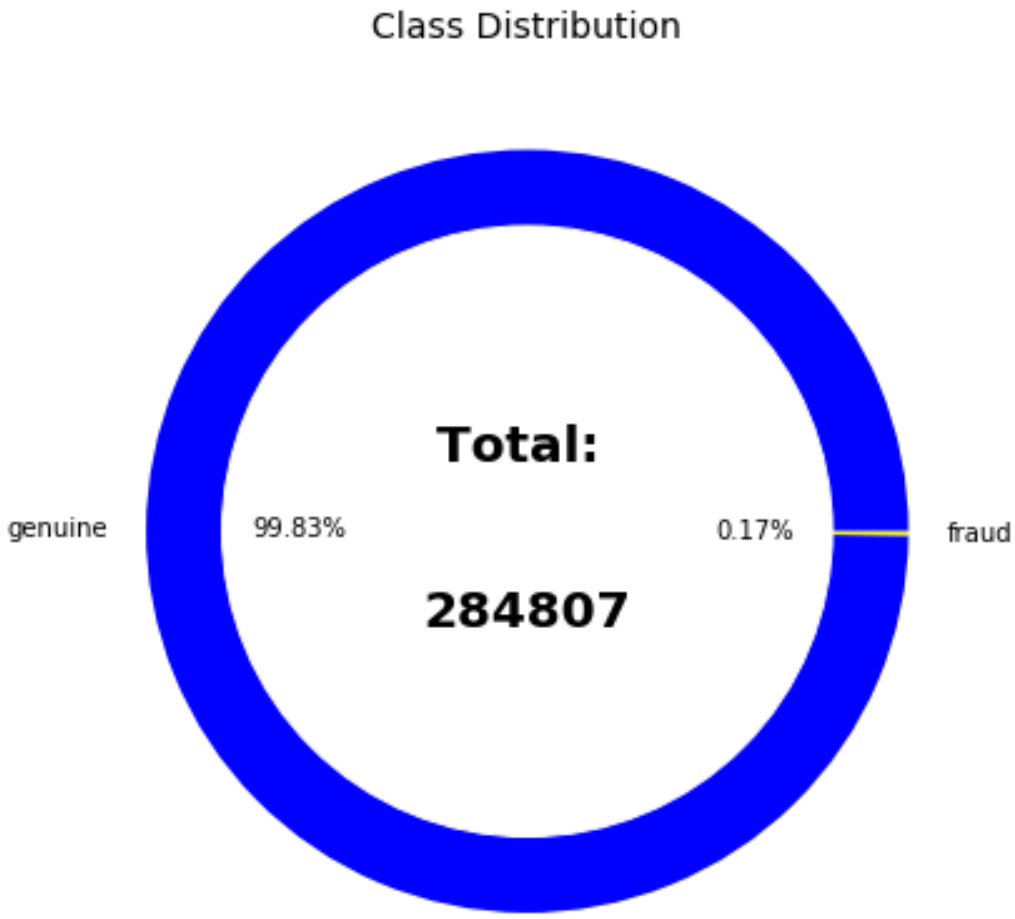

Figure 5.1: The doughnut plot above shows us that the distribution of the classes is skewed for the Kaggle dataset. As we can see fraudulent transactions are only the 0.17% *of the total transactions.*

made experiments on an hour base, instead of a daily base.

5.2 Metrics

For a classification task the basic units to compute performance metrics are:

- *True Positives (TP)*: fraudulent transaction correctly classified as fraud
- *True Negatives (TN)*: genuine transactions correctly classified as gen-

uine

- *False Positives (FP)*: genuine transaction misclassified as fraud
- *False Negatives (FN)*: fraudulent transactions misclassified as genuine

True Positives, True Negatives, False Positives and False Negatives are usually aggregated together in the *confusion matrix*. The confusion matrix is a table that allows the visualization of the performances of an algorithm. As we can see from Figure 5.2, the rows of the matrix represent the samples belonging to the different classes, while the columns represents the samples predicted to belong to a certain class. In the case in example, denoting the label 1 with the positive class, we have that on the top-left there are the True Negatives, on the top-right the False Positives, on the bottom-left the False Negatives, and on the bottom-right the True Positives.

Starting from these units, performance metrics are computed in order to evaluate and compare different algorithms. The most used are:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$FalsePositiveRate(FPR) = \frac{FP}{FP + TN}$$

Recall is also called *True Positive Rate*(TPR).
As stated in Section 2.3, these metrics are not suited for evaluating performances in fraud detection tasks. In the case of accuracy, given that frauds are usually around 1% of the total transactions, we can achieve really high

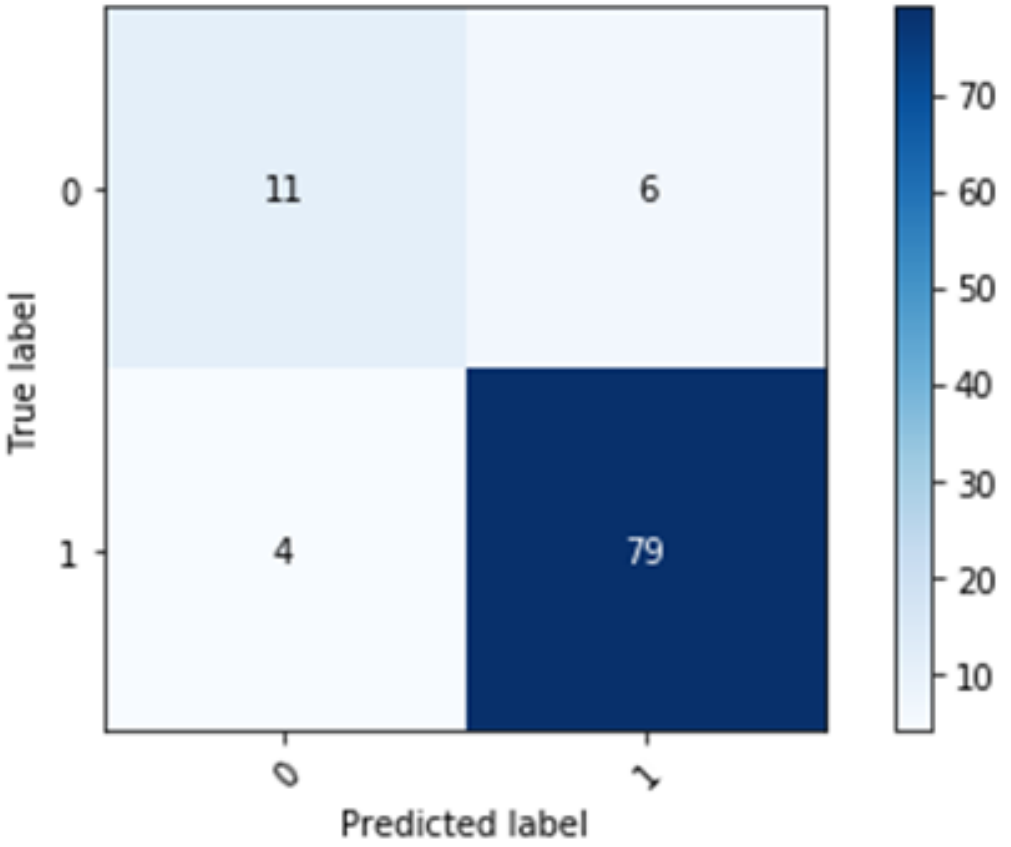

Figure 5.2: Confusion Matrix. In here we can see that for that specific classification task, on a total of 100 samples, there were 83 positive samples, of which 79 have been predicted correctly and 4 have been predicted wrongly, and 17 negative samples, of which 11 have been predicted correctly and 6 have been predicted wrongly.

scores also for classifiers that classify everything as genuine, but our goal is to identify the very few fraudulent activities. In the case of precision and recall, instead, they are not good indicators because in a real-world setting investigators can check only few, high risky, transactions, so we're not interested in the performance on the whole test set, but we want to focus on alerting the smallest possible number of false positive, but very accurate.
For these reasons we assess our results in terms of *alert precision* $P_k(t)$, that is defined as:

$$P_k(t) = \frac{|TP_k(t)|}{k}$$

with $TP_k(t) = \{(x_i, y_i)|x_i \in A_t, y_i = 1\}$, is the proportion of fraud in the alerted transactions at day t.
In a real-world scenario when an investigator contacts a cardholder, he checks up for all the recent transactions. For this reason, the alert precision should be measured in terms of cards rather than transactions, in a way such that multiple fraudulent transactions coming from the same card count as a single detection [12]. Unfortunately in our work we couldn't do it because the data were anonymized and we didn't have any information on cards or cardholders.
where C_t^1 is the set of fraudulent cards alerted at day t.
In [51] and [43], they assess the *area under the ROC curve* (AUC) as one of the most used performance measures for fraud-detection problems. The *Receiving Operator Characteristic* [22] is a graphical representation that plots TPR against FPR at various discrimination threshold settings, in order to detect how good a model is at binary classification. The AUC is the area under this curve, with a maximum of 1, and the larger the area, the better the model. In Figure 5.3 we can see two models compared with respect to their AUC, and we can see that none of them consistently outperform another, but that one is better for certain threshold and one for others.
In our setting this measure can be interpreted as the probability that a classifier ranks frauds higher than genuine transactions.

Another metric that we used for our evaluation of the performance was the *Detection Rate*, that is the percentage of detected fraudulent transactions over the total:

$$DR(t) = \frac{|TP_k(t)|}{TP(t) + FN(t)} \tag{49}$$

This equation is similar to the Recall measure, but it's not the same. The difference is that the numerator is relative to the first k alerted transactions,

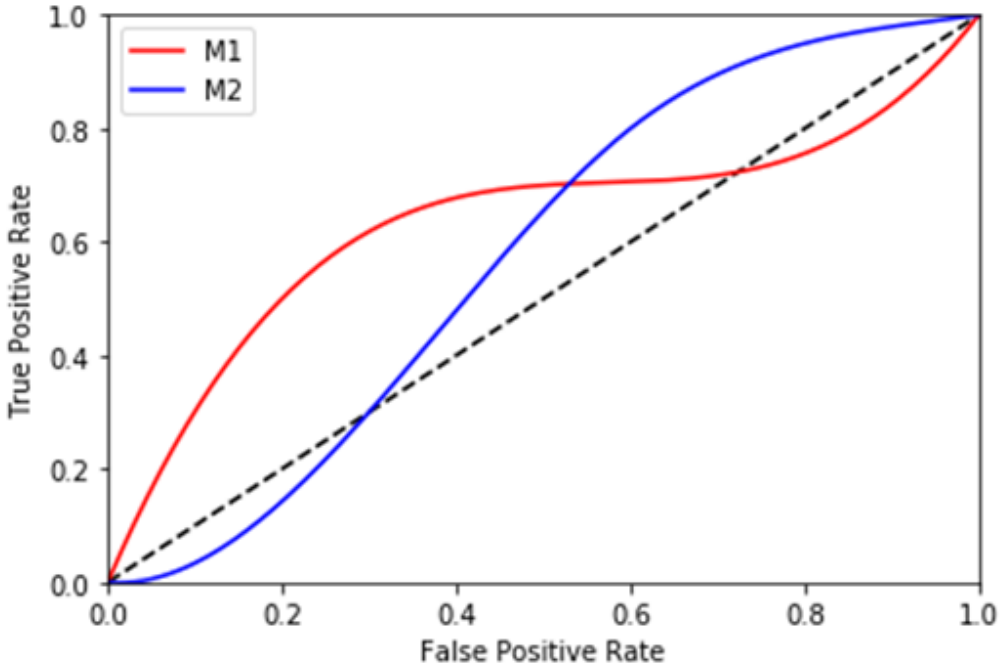

Figure 5.3: AUC. In this pictures we can see two models, $M1$ and $M2$. The dashed diagonal line is equivalent to random guessing, and its AUC is 0.5. Every model with an AUC lower than 0.5 is useless since it's worse than random guessing. In our case we can see how our two models are not constantly over-performing another, but that $M1$ is better when we have small FPR, while $M2$ is better when we have large FPR. In fraud detection the first model would be better, because we aim at a model with small FPR.

Method	Precision	Recall	AUC
No OS	0.8988	0.8080	0.9336
No OS without overlapping features	0.9294	0.7979	0.9336
ROS	0.9195	0.8080	0.9385
ROS without overlapping features	0.9080	0.7979	0.9385
WGAN	0.8863	0.7878	0.9412
WGAN without overlapping features	0.9058	0.7777	0.9283
GAMO	0.9058	0.7777	0.9384
GAMO without overlapping features	0.9176	0.7878	0.9284
GA	0.9404	0.7979	0.9384
GA without overlapping features	0.9294	0.7979	0.9383

Table 5.1: Performance of the different methods used on the whole dataset.

while the denominator is relative to the whole dataset taken into account.

5.3 Results

We will now present and discuss the results obtained in our first set of experiments. The results can give us useful insights on the advantages of a method against another one when the experiments have been performed on the whole dataset and when they have been performed on the framework implementation. Even if we consider the small amount of data available, the results can be meaningful since they take into account real transactions, and they were achieved implementing a framework able to scale up with more data.

As we can see from Table 5.1, the results of the different methods are similar, but there are some small changes in performances. In particular we notice that excluding the features with overlapping distributions between the two classes led to a better precision overall, but also to a decrease in recall

Method	Precision Top100	Recall Top100	AUC
No OS	0.9080	0.9634	0.9284
ROS	0.9186	0.9518	0.9335
WGAN	0.9069	0.9629	0.9327
GAMO	0.9101	0.9506	0.9336
GA	0.9404	0.9753	0.9384

Table 5.2: Performance of the different methods used on the whole dataset for Top100 metrics.

and AUC. Furthermore, the only metric used in here and also taken into consideration for our analysis is the AUC metric, and we can notice that the best performance was obtained by the method using WGAN as oversampling technique.

In Table 5.2 are reported Top100 Precision, Top100 Recall and AUC when the experiments were done considering the whole dataset. We can see in here that oversampling techniques gave some boost on the performance on the classification task, with an AUC score that is better than the basiline for every method. We can see from the table that the best results for all the metrics taken in consideration are achieved by the oversampling by use of Genetic Algorithm. For the other methods we can see that ROS and GAMO are better than WGAN in Precision and AUC, while the latter overcome the others in Recall.

As already explained previously, the last step in these set of experiments was to implement our framework model. We implemented and tested it using ROS, WGAN oversampling, GAMO oversampling, GA oversampling, and no oversampling.

In Table 5.3 the results of these experiments are shown. As we can see the best results overall are achieved with GA oversampling. In particular this method showed competitive results for all the metrics, with the second best detection rate of 63.95%. The use of ROS and GAMO architecture for over-

sampling leads to the worst performances in Detection Rate, even if they show good results in Precision and AUC. WGAN oversampling gave very bad results in terms of Precision and AUC, but a Detection Rate of 65.74%, the best registered for this phase. This can be explained because with this high precision there were a lot of false positive identified, leading to very good performances in identifying the frauds, with the drawbacks of alerting a lot of genuine transactions. As already said in the previous chapters, this can be a problem in real world settings, since we want to alert few transactions, but very precise.

From the results relative to this phase we can see how Random Oversampling gave us the worst results overall, especially for Detection Rate where it spotted only the 53.44% of the fraudulent transactions. The best scores were reached by GA oversampling, that was the one leading to the most balanced results for all the metrics, with the best scores for both AUC and Detection Rate. We can also see that considering our framework, WGAN oversampling gives promising results, leading to the second best Detection Rate score, with the side effect of low precision. For what concerns GAMO oversampling, the results in terms of AUC were the second best, while the other metrics were worse with respect to the others methods.

Method	Precision Top100	Recall Top100	AUC	DetRate
No OS	0.9218	0.7881	0.9340	0.6334
ROS	0.9431	0.6759	0.9225	0.5344
WGAN	0.8206	0.7966	0.9215	0.6583
GAMO	0.8444	0.7527	0.9360	0.6207
GA	0.9016	0.8152	0.9394	0.6783

Table 5.3: Performances of the different methods used on the framework architectures for Top100 metrics.

In Table 5.4 we report the best results for the tuning phase of our experiments. As we can see all the experiments brought some improvement with respect to the ones without tuning. In particular we can see how the combined use of Random Forest and Genetic Algorithm led to an improvement of almost 2% in AUC.
We can see how also after the tuning phase the Genetic Algorithm oversampling has better results than the GAMO oversampling, confirming the good performances of this customized method. In particular, the best results have been achieved for the combined use of Genitic Algorithm and XGBoost classifier, reaching an AUC score of 96.2%, that is the best score for our experiments on this dataset.

Method	Precision Top100	Recall Top100	AUC	DetRate
GAMO + RF	0.8491	0.8363	0.9369	0.6887
GA + RF	0.9091	0.8445	0.9567	0.7117
GA + NN	0.8531	0.7336	0.9485	0.6042
GA + XGB	0.8813	0.8240	0.9620	0.7001

Table 5.4: Performances for the relevant metrics after tuning of Random Forest, Neural Networks, and XGBoost. For what concerns the neural network, we trained it for 50 epochs with a batch size of 200.

The use of neural networks led to worst results within the experiments on Genetic Algorithms, bust still better than the ones obtained before the tuning phase. For what concerns neural networks, this results are in contrasts with what we saw in the tuning experiments involving the whole dataset. As we can see in Figure 5.4, on the best run(from which we took the network's specifics for the experiments on the framework) the algorithm alerted 93 frauds out of 99 on the test set, leading to an AUC of 96.97%, that is the best result achieved on the whole dataset.
On the experiments we conducted on the framework settings these results

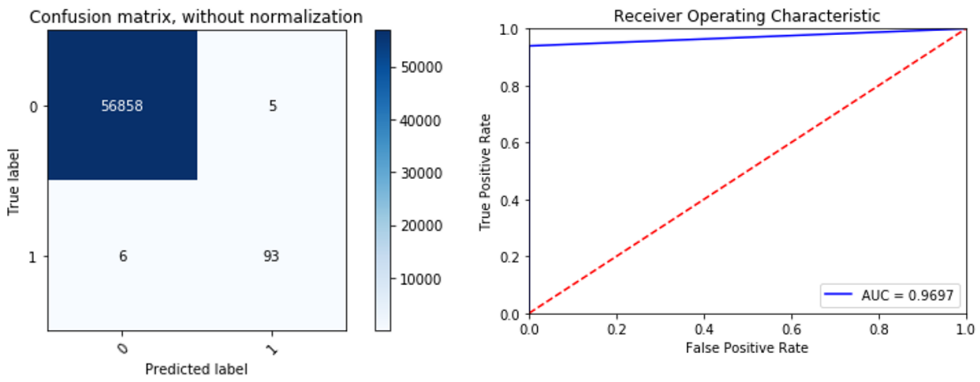

Figure 5.4: Confusion Matrix and AUC plot for the best iteration of the neural network's experiments on the whole dataset with Genetic Algorithm oversampling. As we can see this run was able to recognize as fraudulent 93 transactions on 99, while with the others methods the alerted transactions were never more than 83

wasn't confirmed, and even if the scores were very good, they were lower than the one with other classification methods that performed worse on the whole dataset. This could have happened because, even if the dataset used was the same, in the framework setting the transactions available at each hour were not enough for the training of the neural network, that didn't have enough samples to learn his task, given that the dataset was divided in order to get the hourly set.

Chapter 6

Conclusions And Future Reasearch

In this thesis we presented a framework for fraud detection using machine learning techniques for classification and oversampling techniques to tackle the class imbalance problem. The major goal of our work was to compare different oversampling techniques to see which one performed better and to see when they improve the performances with respect to the baseline implementation, that is the one without oversampling. The oversampling techniques we used were ROS, GAN oversampling, GAMO oversampling, and oversampling with genetic algorithms.

Our analysis were conducted on a dataset available on the Kaggle repository [30], and it followed a precise pipeline. At first the dataset was divided by hours, obtaining 48 different datasets. Then two classifiers were trained, one relative to transactions that were already labeled, available with a delay of 7 hours in our experiments, and one relative to the feedbacks provided from the investigators, that are relative to the transactions alerted at one hour, in our case the 100 most risky transactions with their true label.

The first classifier was an ensemble of classifiers, one for each of the first 8 hours for which all the labels were available, while the second was a random

forest that took as samples the feedbacks coming from the first 7 hours before the current one. Then, the results were aggregated and tested on the following hour. This procedure has been carried on for the whole dataset, hence for 48 hours, and then the results for every hours have been averaged to obtain the final results. Then the 100 most risky transactions were taken to compute the evaluation metrics: Top100 Precision, Top100 Recall, AUC and Detection Rate. After the computation of the metrics, these transactions were stored in the feedback collection, while the oldest were discarded.

In our experiments oversampling was demonstrated to improve performances for fraud detection, and the best results were obtained with the oversampling technique based on genetic algorithm, while there was no classifier that performed clearly better than the others. For what concerns AUC XGBoost obtained the best results, with a score of 96.2% and a detection rate of 70%; for what concerns Precision, the best results were obtained with the classic Random Forest, with a score of 90.9% and a detection rate of 71.2%. Neural Networks didn't perform well on the divided dataset, while they performed the best when we were considering the whole dataset. This inconsistency in the data can be explained by the small amount of data that was fed to the neural networks in the framework implementation, that could have led to bad performances.

The main limitation of our work was the size of the dataset we worked on. As we already said it was taken from the Kaggle repository, and it contains credit card transactions covering 48 hours. This period was enough to make our experiments, but the results are limited to the dataset we considered. Another limitation was that the features were all anonymized except for *"Time"* and *"Amount"*, so we didn't have any information on them, and we couldn't perform any feature engineering step on them.

The future step of research for this work could be to test the implementation on a larger dataset, with features of which we have knowledge about. This could give us a hint on the effectiveness of the solution in a real world scenario, and the possibility of feature engineering could help us improve the

performance. Also the oversampling based on genetic algorithm could be improved with knowledge on the features we have at disposal. Our solution is based solely on the feature importance given by the XGBoost algorithm, while with different features this could be improved and transformed in something more explainable and effective.
In conclusion, we have shown that oversampling improved the performance of fraud detection for the dataset we used, and we think that it should be tested on a bigger dataset in order to evaluate the performances on a dataset more similar to a real world environment.

Bibliography

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN*. 2017. arXiv: 1701.07875 [stat.ML].

[2] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *SIGMOD '15*. 2015.

[3] Hung Ba. "Improving Detection of Credit Card Fraudulent Transactions using Generative Adversarial Networks". In: *CoRR* abs/1907.03355 (2019). arXiv: 1907.03355. URL: http://arxiv.org/abs/1907.03355.

[4] Siddhartha Bhattacharyya et al. "Data mining for credit card fraud: A comparative study". In: *Decision Support Systems* 50.3 (2011). On quantitative methods for detection of financial fraud, pp. 602–613. ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2010.08.008. URL: http://www.sciencedirect.com/science/article/pii/S0167923610001326.

[5] Leo Breiman. "Bagging Predictors". In: *Machine Learning* 24 (1996), pp. 123–140.

[6] Leo Breiman. "Random Forests". In: *Mach. Learn.* 45.1 (Oct. 2001), pp. 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

[7] Fabrizio Carcillo et al. "Combining unsupervised and supervised learning in credit card fraud detection". In: *Information Sciences* (2019). ISSN: 0020-0255. DOI: https://doi.org/10.1016/j.ins.2019.05.

042. URL: http://www.sciencedirect.com/science/article/pii/S0020025519304451.

[8] Nitesh Chawla et al. "SMOTE: Synthetic Minority Over-sampling Technique". In: *J. Artif. Intell. Res. (JAIR)* 16 (Jan. 2002), pp. 321–357. DOI: 10.1613/jair.953.

[9] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *CoRR* abs/1603.02754 (2016). arXiv: 1603.02754. URL: http://arxiv.org/abs/1603.02754.

[10] François Chollet et al. *Keras*. https://keras.io. 2015.

[11] Corinna Cortes et al. *Sample Selection Bias Correction Theory*. 2008. arXiv: 0805.2775 [cs.LG].

[12] Andrea Dal Pozzolo et al. "Credit Card Fraud Detection: A Realistic Modeling and a Novel Learning Strategy". In: *IEEE Transactions on Neural Networks and Learning Systems* PP (Sept. 2017), pp. 1–14. DOI: 10.1109/TNNLS.2017.2736643.

[13] *DataBricks*. https://https://databricks.com/.

[14] Jerome H. Friedman. "Greedy function approximation: A gradient boosting machine." In: *Ann. Statist.* 29.5 (Oct. 2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451. URL: https://doi.org/10.1214/aos/1013203451.

[15] João Gama et al. "A survey on concept drift adaptation". In: *ACM Comput. Surv.* 46 (2014), 44:1–44:37.

[16] *GAMO: Generative Adversarial Minority Oversampling*. https://github.com/SankhaSubhra/GAMO. Accessed: 2019-11-19.

[17] *GAN for Credit Card Data*. https://www.toptal.com/machine-learning/generative-adversarial-networks. Accessed: 2019-11-14.

[18] *GAN for Credit Card Data-GitHub*. https://github.com/codyznash/GANs_for_Credit_Card_Data. Accessed: 2019-11-14.

[19] *GANs: One of the Hottest Topics in Machine Learning*. https://www.linkedin.com/pulse/gans-one-hottest-topics-machine-learning-al-gharakhanian/?trk=pulse_spock-articles.

[20] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[21] Xinjian Guo et al. "On the Class Imbalance Problem". In: *Fourth International Conference on Natural Computation, ICNC '08* Vol. 4 (Oct. 2008). DOI: 10.1109/ICNC.2008.871.

[22] J.A. Hanley and Barbara Mcneil. "The Meaning and Use of the Area Under a Receiver Operating Characteristic (ROC) Curve". In: *Radiology* 143 (May 1982), pp. 29–36. DOI: 10.1148/radiology.143.1.7063747.

[23] Hassan Hassan et al. "ASSESSMENT OF ARTIFICIAL NEURAL NETWORK FOR BATHYMETRY ESTIMATION USING HIGH RESOLUTION SATELLITE IMAGERY IN SHALLOW LAKES: CASE STUDY EL BURULLUS LAKE." In: *International Water Technology Journal* 5 (Dec. 2015).

[24] Haibo He et al. "ADASYN: Adaptive Synthetic Sampling Approach for Imbalanced Learning". In: July 2008, pp. 1322–1328. DOI: 10.1109/IJCNN.2008.4633969.

[25] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992. ISBN: 0262082136.

[26] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[27] Hamza Jaffali and Luke Oeding. "Learning Algebraic Models of Quantum Entanglement". In: (Aug. 2019).

[28] Nathalie Japkowicz. “Learning from Imbalanced Data Sets: A Comparison of Various Strategies”. In: *Learning from Imbalanced Data Sets: A Comparison of Various Strategies* 68 (June 2000).

[29] *JupyterLab*. `https://github.com/jupyterlab/jupyterlab`.

[30] *Kaggle Dataset Credit Cards*. `https://www.kaggle.com/mlg-ulb/creditcardfraud`. Accessed: 2019-11-12.

[31] Vishwa Karia et al. *GenSample: A Genetic Algorithm for Oversampling in Imbalanced Datasets*. 2019. arXiv: `1910.10806 [cs.LG]`.

[32] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: `1412.6980 [cs.LG]`.

[33] Gunter Klambauer et al. *Self-Normalizing Neural Networks*. 2017. arXiv: `1706.02515 [cs.LG]`.

[34] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. “Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning”. In: *Journal of Machine Learning Research* 18.17 (2017), pp. 1–5. URL: `http://jmlr.org/papers/v18/16-365.html`.

[35] Alexander Y. Liu. “The Effect of Oversampling and Undersampling on Classifying Imbalanced Text Datasets”. In: 2004.

[36] Fei Tony Liu, Kai Ming Ting, and Zhi-hua Zhou. “Isolation Forest”. In: *In ICDM â08: Proceedings of the 2008 Eighth IEEE International Conference on Data Mining. IEEE Computer Society*, pp. 413–422.

[37] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[38] Wes McKinney. “pandas: a Foundational Python Library for Data Analysis and Statistics”. In: ().

[39] Sankha Subhra Mullick, Shounak Datta, and Swagatam Das. “Generative Adversarial Minority Oversampling”. In: *ArXiv* abs/1903.09730 (2019).

[40] Travis Oliphant. *NumPy: A guide to NumPy*. USA: Trelgol Publishing. 2006–. URL: http://www.numpy.org/.

[41] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[42] Andrea Dal Pozzolo et al. "Credit card fraud detection and concept-drift adaptation with delayed supervised information". In: *2015 International Joint Conference on Neural Networks (IJCNN)* (2015), pp. 1–8.

[43] Andrea Dal Pozzolo et al. "Learned lessons in credit card fraud detection from a practitioner perspective". In: *Expert Syst. Appl.* 41 (2014), pp. 4915–4928.

[44] *PySpark ML*. https://spark.apache.org/docs/latest/api/python/pyspark.ml.html.

[45] *PySpark Package*. https://spark.apache.org/docs/latest/api/python/pyspark.html.

[46] *Python 3.6.8*. https://www.python.org/downloads/release/python-368/.

[47] *RandomForestClassifier SKLearn*. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[48] Robert E. Schapire. "The Strength of Weak Learnability". In: *Mach. Learn.* 5.2 (July 1990), 197â227. ISSN: 0885-6125. DOI: 10.1023/A:1022648800760. URL: https://doi.org/10.1023/A:1022648800760.

[49] Dennis J. N. J. Soemers et al. "Adapting to Concept Drift in Credit Card Transaction Data Streams Using Contextual Bandits and Decision Trees". In: *AAAI*. 2018.

[50] Scott Thede. "An introduction to genetic algorithms". In: *Journal of Computing Sciences in Colleges* 20 (Oct. 2004).

[51] Véronique Van Vlasselaer et al. “APATE: A novel approach for automated credit card transaction fraud detection using network-based extensions”. In: *Decision Support Systems* 75 (2015), pp. 38–48.

[52] Svante Wold, Kim H. Esbensen, and Paul Geladi. “Principal Component Analysis”. In: 1987.

[53] *XGBoost Parameters*. `https://xgboost.readthedocs.io/en/latest/parameter.html`.

[54] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *HotCloud*. 2010.

[55] Li Zhou. *A Survey on Contextual Multi-armed Bandits*. 2015. arXiv: `1508.03326 [cs.LG]`.