POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Numerical Analysis for Machine Learning Project

Authors: **Karanbir Singh (10865124)**
Matteo Vitali (10800443)

Academic Year: 2024-2025

Contents

1 | Introduction

In recent years, thanks to government promotion and online payments, the use of credit cards for payments has significantly increased, replacing cash transactions in many parts of the world. Unfortunately, especially in online transactions, the fraud rate is increasing, so it is crucial to develop effective systems to detect fraudulent activities in real time.

In general, there's no strict set of rules that defines how a fraud occurs, but fraudulent transactions often share certain patterns, so we can train machine learning models on historical transaction data to make a classification. With those models, we can uncover hidden correlations that might be difficult for humans to detect by processing vast amounts of data.

Various methods have been explored to improve fraud detection, including those presented in the research by Emmanuel Ileberi, Yanxia Sun, and Zenghui Wang in their paper ***"A Machine Learning-Based Credit Card Fraud Detection Using the GA Algorithm for Feature Selection"***. Here, the authors propose a two steps approach. Initially, a genetic algorithm is applied to the dataset to identify the most relevant and informative features. Then, they used the *Scikit-Learn* framework to train multiple classifiers, including:

- Decision Trees (DT);
- Random Forest (RF);
- Naive Bayes (NB);
- Artificial Neural Networks (ANN);
- Logistic Regression (LR).

In this project, we will implement, using just some basic libraries, all proposed classifiers from scratch along with the genetic algorithm. Once finished, we'll compare our results with the ones reported in the paper to see whether researchers' work can be replicated with similar performances.

2 | Dataset inspection

The employed dataset is available at the site `https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud`.

It contains transactions made by credit cards in September 2013 by European cardholders.

This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced: the positive class (frauds) accounts for 0.17275% of all transactions. It contains only numerical input variables which are the result of a PCA transformation, useful for maximizing the **variance of the features** and minimizing the **covariance** between them. Moreover, the dataset doesn't have null or missing values.

Due to confidentiality issues, the original features and more background information about the data are not provided. Features $V_1, V_2, \ldots, V_{28}$ are the principal components obtained with PCA, the only features which have not been transformed with PCA are *Time* and *Amount*.

The feature *Time* contains the seconds elapsed between each transaction and the first transaction in the dataset.

The feature `Amount` is the transaction "amount"; this feature can be used for example-dependent cost-sensitive learning.

The feature `Class` is the response variable, and it takes the value 1 in case of fraud and 0 otherwise.

From an inspection of the **correlation matrix**, it's possible to see that there are no (or really small) correlations between features, so it's not possible to find any patterns or predict any structure which can be exploited in order to make an initial feature selection phase.

Further insights and plots are analysed in the subsequent sections.

3 | Pre-processing

3.1. Min-max scaling

Min-max scaling is the proposed method for *normalizing* the dataset in order to ensure that all the input values are within a predefined range (in this case $[0, 1]$). Here's the mathematical definition:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

where X is the original dataset, X_{min} is the *minimum* value in the feature, X_{max} is the *maximum* value in the feature, and then X_{scaled} is the *scaled* value in the range $[0, 1]$.

This method ensures that all features contribute equally to the analysis, preventing features with larger ranges from dominating those with smaller ranges.

3.2. Train-test-validation split

The data is ordered by increasing values of the feature `Time` that represents the number of seconds between a certain transaction and the first one of the dataset.

3.3. Undersampling

Undersampling is a way to handle the high class imbalance of the dataset. Building a classifier on the entire dataset would lower the accuracy since the model would overfit, or, in the worst case, classify all the transactions as normal. The method `undersample()` is used to undersample the majority class by dropping a `drop_percentage` samples belonging to it.

3.4. SMOTE

SMOTE is the acronym of *"Synthetic Minority Oversampling Technique"* and, in the paper, is used to face the class imbalance problem. As already pointed out, only 492 of

the total 284807 transactions are fraudulent (0.172%).

The idea of SMOTE is to generate synthetic samples by interpolating between existing minority samples and their nearest neighbors in the feature space. This means that we're moving the decision boundary closer to the majority class while maintaining the diversity of the minority class distribution: for example, if x_i and x_j are closely clustered minority samples, the synthetic sample $x_{\text{synthetic}}$ lies on the line connecting them, ensuring it represents the local structure of the minority class.

The algorithm is the following:

Algorithm 3.1 SMOTE (Synthetic Minority Over-sampling Technique)

Require: M: minorities set
$n_{samples}$: number of synthetic samples to generate
k: number of neighbors to compute

Ensure: S: set of synthetic samples

$S \leftarrow [\,]$
for each $x_i \in M$ **do**
  neigh $\leftarrow$ Neighbors$(x_i, k+1)$
  Choose one x_j from neigh randomly
  $\lambda \sim U(0,1)$
  $x_{\text{synthetic}} \leftarrow x_i + \lambda(x_j - x_i)$
  $S \leftarrow S \cup \{x_{\text{synthetic}}\}$
end for
return S

In the paper there's no clear specification of how SMOTE was applied or if it was applied only for some models. In our project, we alternate it with undersampling to get the best values for the metrics (accuracy, precision, recall, F1).

4 | Classifiers

4.1. Decision Trees

4.1.1. Overview

A decision is a self-supervised ML algorithm. It's said to be *non-parametric* because it seeks to best fit the training set with a large number of functional forms, while keeping the ability to generalize well. DTs are a good option for this dataset because we have a lot of data, but no prior knowledge about the features we're working with.

DTs employ a greedy *divide and conquer* strategy to choose the optimal split point within the tree: at each iteration, the data is split into subsets based on feature values to minimize *impurity* (i.e. maximize *information gain*). The majority of DTs algorithms are based upon Hunt's algorithm; the most popular ones are the following:

- **ID3**: shorthand for *"Iterative Dichotomiser 3"*, leverages entropy and information gain to evaluate candidate splits. We always split on a single variable, but if the variable is categorical with $|V|$ possible values the node where we branch will have (at most) $|V|$ outgoing edges.
- **C4.5**: an evolution of *ID3* that can use both information gain or gain ratios to evaluate a split point.
- **CART**: shorthand for *Classification And Regression Trees*, typically uses Gini impurity to choose where to split. Classification trees produce a label, while regression trees produce numeric data. They are typically used for continuous data; for categorical data we need to add n variables where $n = |V|$ (one hot encoding): the resulting dataset is said to be *sparse* (lots of zeros).

The DT implemented is a CART one even if it allows to use entropy instead of Gini impurity for the branching phase. Within a node, if its associated statement is true ($x_{f_i} \leq \tau$), we branch on the left, otherwise on the right.

The very top of the tree is called *root*, nodes with incoming and outgoing arrows are called

branches and *leafs* are the terminal nodes that contain the prediction.

4.1.2. (Im) - purity measures

Purity means how well a node is splitting the data or how mixed are the classes within it; a node is *pure* if all the samples in it belongs to the same class. Two of the most important measures for impurity are the following:

- **Gini impurity**: $\mathcal{G}(y)$.

  In a node, it measures the likelihood of incorrectly classifying a randomly chosen sample labelled according to the distribution of labels in that node. Mathematically, it is defined as:

  $$\mathcal{G}(y) = 1 - \sum_{k=1}^{K} p_k^2$$

  where K is the number of labels in the node and p_k is the probability of assigning the $k-$th label to a sample.

  For example, suppose that in the node there are labels L_1, L_2, L_3 and you generate a sample with label L_1: misclassifying it means assigning it labels L_2 or L_3. So, Gini impurity is:

  $$\mathcal{G}(y) = 1 - \text{probability of correctly classifying the sample} = 1 - \sum_{L_i} p_{L_i}^2$$

  given that generation and labelling are independent, each with probability p_{L_i}.

  If $\mathcal{G}(y) = 0$, the node is perfectly pure (all samples belongs to the same class). Closer to 0.5 $\mathcal{G}(y)$ is, more distributed samples are.

- **Entropy**: $\mathcal{H}(y)$.

  It measures the amount of uncertainty or disorder in the target variable. Mathematically, it is defined by:

  $$\mathcal{H}(y) = -\sum_{k=1}^{K} p_k \log_2(p_k)$$

  where K is the number of labels in the node and p_k is the probability of assigning the $k-$th label to a sample.

  As for the Gini impurity, $\mathcal{H}(y) = 0$ means that the node is pure. Higher $\mathcal{H}(y)$ means more disordered samples (maximum value is 1).

4.1.3. Branching

The aim of the branching (or splitting) phase is to determine how to partition the dataset to achieve the best separation of target classes. At each iteration (i.e. call of `find_best_split(...)`), the algorithm looks either for the best split using all available features F, or adopts a selection strategy like p, $\sqrt{F}$ or $\log_2(F)$ that given F total features, compute a subset of size respectively $p \cdot F$, $\sqrt{F}$ or $\log_2(F)$ of them. All possible features and thresholds are then evaluated: the split that that maximizes *information gain* is chosen. This can be computationally expensive, especially for large datasets.

Information gain, as in *ID3*, measures the reduction of impurity (either Gini or entropy) when a node is split into two child nodes. Mathematically, it's given by:

$$\mathcal{I}_G(y) = \mathcal{I}(y) - \text{weighted impurity of children} = \mathcal{I}(y) - \left(\frac{n_{left}}{n} \cdot \mathcal{I}(y_{left}) + \frac{n_{right}}{n} \cdot \mathcal{I}(y_{right}) \right)$$

where $\mathcal{I}(y)$ is the impurity of node y ($\mathcal{H}$ or $\mathcal{G}$), n_{left}, n_{right} are the number of samples in the left and right child nodes ($n = n_{left} + n_{right}$) and $\mathcal{I}(y_{left})$, $\mathcal{I}(y_{right})$ are the impurities of the left and right child nodes.

Let X be the feature matrix and y the target variable. For the splitting phase, the implemented algorithm considers each feature f_i and threshold τ:

- The dataset is split into the left child $X_{left}, y_{left} = \{(x, y) \mid x[f_i] \leq \tau\}$ and right child $X_{right}, y_{right} = \{(x, y) \mid x[f_i] > \tau\}$. There are several ways to obtain τ; the one chosen in this notebook is to compute all unique values in $x[f_i]$, sort them and choose randomly from the pairwise means (i.e. $\frac{x[k,f_i]+x[k+1,f_i]}{2}$, $\forall k < N$) a certain number of elements.
- $\mathcal{I}_G(y)$ and weighted impurity of the split are then computed.
- Then, information gain $\mathcal{I}_G(y)$ is computed.
- If (f, τ^*) maximizes $\mathcal{I}_G$, we set $(f, \tau^*) \leftarrow (f, \tau)$

Note that when the tree is complete, there might be leaves with samples of different classes $(n_{L_1}, n_{L_2}, \ldots, n_{L_k})$; we say that $L = \arg_{i \in \{L_1, \ldots L_k\}} \max(n_i)$ is the dominant label for that leaf and the confidence of the prediction is given by:

$$\frac{n_L}{\sum_{i \in \{L_1, \ldots L_k\}} n_i}$$

4.1.4. Building the tree

The method used to build the tree is recursive and DFS based. It starts from the root and builds the tree by splitting the nodes using the heuristics defined above until one of the following stopping conditions is reached:

- Maximum depth (d_{max}): the height of the tree (i.e. the length of the longest path from the root to a leaf) is equal to d_{max}.
- Minimum samples per leaf (n_{leaf}): stop when the number of samples n of the node to be branched is $n \leq n_{leaf}$.
- Minimum impurity decrease: once we have chosen the best feature and threshold to branch on, we compute the impurity decrease $\Delta\mathcal{I}$ and break if $\Delta\mathcal{I} \leq \epsilon$
- Homogeneity: stop if the node is a leaf.

In particular, (1), (2) and (3) are really useful to prevent overfitting that is one of the most common problems for DT.

4.1.5. Implementation

As for all the others classifiers, an object oriented approach has been used. Each `Decision Tree` is made of `Node` objects.

`Node` is very simple and it has only the methods to initialize its attributes: feature and threshold to branch on (`feature`, `threshold`), left and right children (`left`, `right`) for non-leaf, and most common label and label distribution (`value` and `class_prob`) for leaf ones. `DecisionTree` offers a bunch of tunable parameters:

- `max_depth`: the maximum depth of tree.
- `min_samples_split`: nodes with a lower number of samples, won't be branched (so, they will become leaves).
- `min_samples_leaf`: if a potential split results in a leaf node containing fewer samples than it, the split is not allowed (hence, it's skipped), so that each leaf contains a sufficient number of samples. In the notebook, you will find this parameter set to 1 because of the highly imbalanced dataset: each TP prediction is useful!
- `min_impurity_decrease`: the branch is discarded if it results in an impurity decrease less than it.

- `max_features`: the maximum number of features to choose from when searching for the best split (can be $\sqrt{F}$, $\log(F)$, $p \cdot F$ or an int).

The most important methods are:

- `find_best_split`: searches for the best pair of feature and threshold to branch on.
- `build_tree`: builds the tree in a DFS-like manner until one of the stopping criteria described above is met.
- `fit`: the entry point for building the tree.
- `predict`: makes predictions using the trained tree, along with its confidence.

Alongside them, we write utility methods for most common operations, like splitting the dataset (`split_samples`) or computing impurity (either Gini or entropy).

Choosing the right values for all the parameters is quite difficult; the ones we've used in our `Decision Tree` notebook have been obtained by trial and error. It's worth to notice that, in general, entropy-based trees are deeper than gini-based; moreover, small changes in `min_impurity_decrease` have a large impact on fitting time and metric values (like accuracy or recall).

Our implementation is fairly general: no assumptions were made on the number of labels, so `DecisionTree` class can be used even for multi-class classification problems.

4.2. Random Forest

4.2.1. Overview

As described in DT section, DT can be subject to high overfitting because they have low bias and high variance errors. The first is high if we have made a lot of assumptions about the target function, so if our modelling function is not flexible. The second, instead, refers to the variability of a target function w.r.t. different testing samples: models with high variance will change a lot when small modifications of the training dataset are made. The idea used by RF to overcome this limitations is to build multiple trees and average their prediction for regression or use majority voting (confidence average) for classification. Each tree is built using a different dataset and a random subset of features.

To reduce the variance error, RF employs *bootstrap aggregation* (or *bagging*). Among the different techniques used for it, the one that is used in our notebook was proposed by the inventors of RF Leo Breiman and Adele Cutler.

4.2.2. Building the Random Forest

Given a dataset $D = \{(X_1, y_1), \ldots, (X_N, y_N)\}$ where X_i are the features and y_i the labels, a bootstrapped dataset B is built by sampling with replacement N tuples from D. Because replacement is allowed, there might be some duplicate elements in D_k.

In RF algorithm, a decision tree T_i is trained for each bootstrapped dataset D_i: this allows us to parallelize the training process because different machines will fit one (or several) model. Without bootstrapping, all the trees would be highly correlated, i.e. if T_i predicts label $\tilde{y}$, there's a high chance that also T_k will predict $\tilde{y}$: consequently, if one overfits, the others will overfit either.

Moreover, each T_i is built considering only a random subset of $|F_i| = m$ features among the total $|F| = d$ features in each of the best split call (see `DecisionTree.find_best_split`): if we used the entire features vector, even with bootstrapping, there would be an high chance to keep using the mostly dominant features, not reducing correlation. Typically, $m = \sqrt{d}$ for classification and $m = \frac{1}{3}d$ for regression.

4.2.3. Making predictions

When it comes to make a prediction for a classification task, the idea is to use *majority voting*:

$$\tilde{y} = \arg\max_k \sum_{i=1}^{B} \mathbb{I}\{T_i(x) = k\}$$

where k is a label, $\mathbb{I}\{...\}$ is the indicator function and $T_i(x)$ is the prediction of the i-th tree. Another way is to exploit the probabilities outputted by each tree. For a specific sample x, the confidence of the RF in the prediction k is given by the average of the confidences of the trees:

$$P\{k \text{ for sample } x\} = \frac{1}{B} \sum_{i=1}^{B} P\{k \text{ assigned to } x \text{ by tree } T_i\}$$

Consequently, the predicted class is the one with highest probability:

$$\tilde{y} = \arg\max_k \frac{1}{B} \sum_{i=1}^{B} P\{k \text{ assigned to } x \text{ by tree } T_i\}$$

Both methods are supported by our implementation. In general, the latter approach is preferred when it is required to plot ROC curves since it outputs continuous probabilities, rather than agreement fractions (nevertheless, the first can still be used). Also, the second is more calibrated because it reflects the true probabilities of DT in the RF; since in our DT the `Node.value` of a leaf is given by the classes with largest number of samples belonging to it (so, the one with highest probability), both methods will produce the same results in terms of accuracy, precision, recall and F1.

In contrast, for a regression task (not necessary in our project, but reported for the sake of completeness):

$$\tilde{y} = \frac{1}{B} \sum_{i=1}^{B} T_i(x)$$

4.2.4. Implementation

As done for all the other classifiers, `RandomForest` is written in an Object Oriented manner.

The constructor initializes both random forest and decision tree (the building blocks of

RF) parameters. For the first, they are `n_trees` and `random_seed`; for the second, they are described in depth in 4.1 chapter. A list of `DecisionTree` instances is then created. Even if for classification tasks `max_features = "sqrt"` should be used, we left to the user this choice by making this parameter available also in the constructor and not burying it inside `self.trees` definition. Note that `RandomForest` can be used, also, for multi-class classification problems because no assumptions were made on the number of labels.

The only methods exposed by `RandomForest` are `fit` and `predict`. In `fit`, for each tree we first create a bootstrapped dataset by sampling with replacement $|X|$ elements from X and, then, we fit it. This means that more trees the RF contains, more time is required for training. `predict` is used to predict the label of a given set of samples. Alongside the predicted label, also the probability distribution of the possible classes for the sample are returned. As described above, two types of confidence measures are supported:

- `type = "prob"`: based on confidence of each tree.
- `type = "voting"`: based on number of votes.

4.3. Naive Bayes

4.3.1. Overview

A Bayes Classifier is able to estimate $f : X \to Y$ or equivalently $P\{Y|X\}$ where Y is a boolean random variable and X is a vector containing the attributes. Applying Bayes rule, we see that we can express $P\{Y = y_i|X = x_i\}$ as the following:

$$P\{Y = y_i|X = x_k\} = \frac{P\{X = x_k|Y = y_i\} \cdot P\{Y = y_i\}}{\sum_j P\{X = x_k|Y = y_j\} \cdot P\{Y = y_j\}}$$

where y_i denotes the i-th possible value of Y and x_k denotes the k-th possible vector value for X. One way to learn $P\{Y|X\}$ is to use the training data to estimate $P\{X|Y\}$ and $P\{Y\}$: we can then employ them, together with Bayes rule above, to determine $P\{Y|X = x_k\}$ for any new instance x_k. A classifier like this will output the probability distribution over all possible values of Y for each new instance X that we ask it to classify. As in DT and RF, there is no need to perform standardization: it's internally done by the method (RF and DT being based on `if` conditions are invariant w.r.t linear transformation of the input variables).

4.3.2. Conditional independence

Using a Bayes classifier to accurately estimate $P\{X|Y\}$ typically requires a lot of training examples. To overcome this problem, Naive Bayes classifiers assume that each X_i is conditionally independent of each of the others X_k given the label Y and also independent of each subset of the others X_k given Y. Given three sets of random variable X, Y and Z, we say that X is conditionally independent of Y given Z, if and only if the probability distribution governing X is independent of the value of Y given Z; that is:

$$P\{X = x_i|Y = y_j, Z = z_k\} = P\{X = x_i|Z = z_k\} \quad \forall i, j, k$$

With this hypothesis, we can write $P\{X|Y\}$ like:

$$P\{X|Y\} = P\{X_1, \ldots, X_n|Y\} = \frac{P\{X_1, \ldots, X_n\}}{P\{Y\}} = \frac{P\{X_1, \ldots, Y\}}{P\{X_2, \ldots, Y\}} \cdot \frac{P\{X_2, \ldots, Y\}}{P\{Y\}} =$$

$$P\{X_1|X_2, \ldots, Y\} \cdot P\{X_2, \ldots X_n|Y\} = P\{X_1|Y\} \cdot P\{X_2|Y\} \cdots P\{X_n|Y\} = \prod_{i=1}^{n} P\{X_i|Y\}$$

where we have applied n times the definition of conditional independence and a general property of probabilities.

Conditional independence assumptions hold in our dataset since, as can be seen in 2 chapter, the correlation of all possible pairs of features is almost zero, except for `Amount` and `Time`. It's possible to apply PCA again to have a perfectly independent set of features.

4.3.3. Naive Bayes algorithm

Naive Bayes algorithm works the same for attributes X being either discrete or real-valued. The expression for the probability that Y will take its k-th possible value (i.e. the posterior probability) is:

$$P\{Y = y_k|X_1, \ldots, X_n\} = \frac{P\{X_1, \ldots, X_n|Y = y_k\} \cdot P\{Y = y_k\}}{\sum_j P\{X_1, \ldots, X_n|Y = y_j\} \cdot P\{Y = y_j\}}$$

where the sum is taken over all possible values y_j of Y. By what have previously said about conditional independence, we can rewrite the expression as:

$$P\{Y = y_k|X_1, \ldots, X_n\} = \frac{P\{Y = y_k\} \cdot \prod_i P\{X_i|Y = y_k\}}{\sum_j P\{Y = y_j\} \cdot \prod_i P\{X_i|Y = y_i\}}$$

that is the fundamental equation for the Naive Bayes classifier; $P\{Y\}$ (the prior) and $P\{X_i|Y\}$ (the likelihood) have been already estimated.

If we are interested in only the most probable value of Y, then we have the Naive Bayes classification rule:

$$Y = \arg\max_{y_k} \frac{P\{Y = y_k\} \cdot \prod_i P\{X_i|Y = y_k\}}{\sum_j P\{Y = y_j\} \cdot \prod_i P\{X_i|Y = y_i\}}$$

Noticing that the denominator doesn't depend on y_k, we can get rid of it and write:

$$Y = \arg\max_{y_k} \; P\{Y = y_k\} \cdot \prod_i P\{X_i|Y = y_k\}$$

4.3.4. Mean and variance estimation

All of the features in our samples X are continuous, so we need a way to represent the distributions of $P\{X_i|Y\}$. One common approach is to assume that for each possible discrete value y_k of Y, the distribution of each continuous X_i is Gaussian defined by mean and standard deviation specific to X_i and y_k. Such parameters can be estimated using conditional expected value:

$$\mu_{i,k} = \mathbb{E}[X_i|Y = y_k] \qquad \sigma^2_{i,k} = \mathbb{E}[(X_i - \mu_{i,k})^2|Y = y_k]$$

for each feature X_i and possible label y_k of Y. This means that there are $2|Y| \cdot n$ of this parameters and each of them has to be estimated independently. Among many estimators, the ones used in the notebook are the following:

- MLE (Maximum Likelihood Estimator) for the mean:

$$\hat{\mu}_{i,k} = \frac{1}{\sum_j \mathbb{I}\{Y_j = y_k\}} \cdot \sum_j X_{i,j} \cdot \mathbb{I}\{Y_j = y_k\}$$

  that is nothing more than the mean of $X_{i,j}$.

- MLE for the variance:

$$\hat{\sigma}^2_{i,k} = \frac{1}{\sum_j \mathbb{I}\{Y_j = y_k\}} \cdot \sum_j (X_{i,j} - \hat{\mu}_{i,k})^2 \cdot \mathbb{I}\{Y_j = y_k\}$$

  that is biased.

- MVUE (Minimum Variance Unbiased Estimator):

$$\hat{\sigma}^2_{i,k} = \frac{1}{\left(\sum_j \mathbb{I}\{Y_j = y_k\}\right) - 1} \cdot \sum_j (X_{i,j} - \hat{\mu}_{i,k})^2 \cdot \mathbb{I}\{Y_j = y_k\}$$

where Y_j means the label and $X_{i,j}$ means the value of i-th feature of j-th training sample.

Also, priors $\pi_k = P\{Y = y_k\}$ have to be estimated:

$$\hat{\pi}_k = \frac{\sum_j \mathbb{I}\{Y_j = y_k\}}{n}$$

4.3.5. Implementation

As done for all the other classifiers, we used an Object Oriented approach. To the constructor, the user has to only pass the classes. Priors, means and variances are stored as dictionaries, so that `classes` can be a generic array of hashable objects. Note that no assumptions were made on the number of classes, so `GaussianNaiveBayes` class can be used for multi-class problems. Each value of `means` and `variances` is an array containing respectively the mean and the variance for the Gaussian distribution associated to class, feature pair (y_k, X_i).

The class presents two main methods: `fit` and `predict`. The first one is used to estimate the parameters of the model, the latter to make predictions. Computed probabilities can be really small numbers, and this can cause numerical instability, so we use the log-sum-exp trick that works as follows. We know that probabilities are in $[0, 1]$ but they are concentrated around 0, so, since we're working with their logs we have very negative numbers in $(-\infty, 0]$. If we subtract from them their maximum value, then we remap probabilities close to 1, avoiding underflow. To get the real value, we should multiply by e^m where m is the maximum, but since we need a normalization in $[0, 1]$:

$$p = \frac{e^{l_i - m}}{\sum_j e^{l_j - m}} = \frac{e^{l_i}}{\sum_j e^{l_j}}$$

Alongside the predicted label, also the probability distribution of the possible classes for the sample is returned.

A useful method, especially for debugging purposes, is `plot_gaussian_pdfs` that allow to print the probability distribution functions for each class and feature.

4.4. Artificial Neural Network

An artificial neural network (ANN) is a supervised ML model inspired by the way biological neural networks in the human brain process information. It consists of layers of interconnected **neurons** that work together to recognize patterns, make decisions, or perform complex tasks.

4.4.1. Structure

An ANN consists of layers of neurons connected by weighted edges. It can be represented as a function $f : X \to Y$, where X represents the input (features) and Y the output (class).

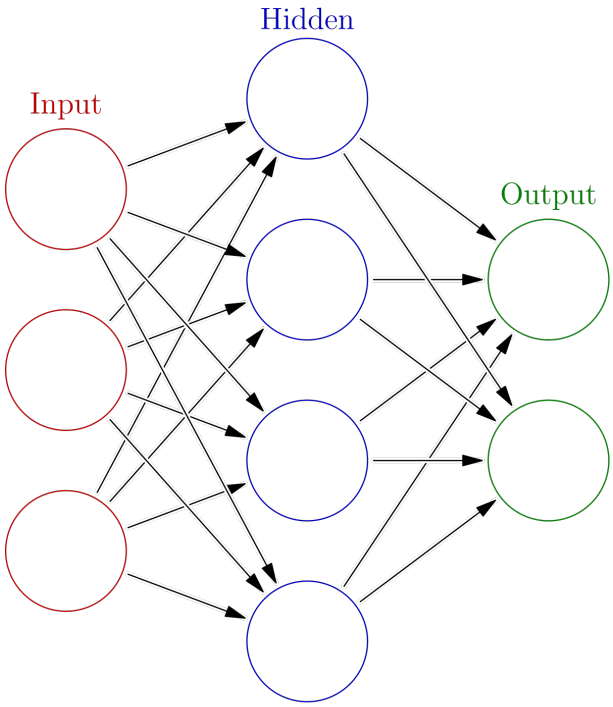

Figure 4.1: An example of multilayer neural network

- **input layer**: a set of neurons (which *cardinality* is the number of features in a given dataset) which receives raw data (features) and passes it to the next layer;
- ***many* hidden layer**: intermediate layers where computation occur. It's where the non-linear transformations are applied via **activation function** (*described later*);
- **output layer**: produces the final computed/predicted result;

Given an ANN with L layers, $\mathbf{x}^{(0)}$ the input vector, for each layer l:

$$\mathbf{x}^{(l)} = \sigma(\mathbf{z}^{(l)})$$

$$\mathbf{z}^{(l)} = W^{(l)}\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)}$$

where:

- $\mathbf{W}^{(l)}$ is the weight matrix of layer l
- $\mathbf{b}^{(l)}$ is the bias vector
- $\mathbf{z}^{(l)}$ is the weighted sum of values of layer $l-1$
- σ is the activation function
- $\mathbf{x}^{(l)}$ is the output after activation

4.4.2. Activation function

The **activation function** is used for introducing *non-linearity* into the model, allowing the network to learn and represent complex patterns in the data. Without this non-linearity feature, a neural network would behave like a *linear regression* model, no matter how many layers it has. Some common functions are provided:

1. **sigmoid function** [usually used in the *output layer*, for *binary classification tasks*]:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

2. **hyperbolic tangent (tanh) function** [usually used in the *hidden layer*]:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

3. **ReLU** (Rectified Linear Unit) [common in *deep networks*]:

$$ReLU(x) = \max\{0, x\}$$

4. **softmax function** [usually used in the *output layer*, for *non-binary classification tasks*]:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

4.4.3. Learning and Optimization

ANNs learn by adjusting *weights* and *biases* using an optimization technique called **gradient descent** (or variants like **stochastic gradient descent**, **RMSProp**, ...), described in the backpropagation algorithm.

The *objective* is to minimize a **loss function** $J(\mathbf{y}, \hat{\mathbf{y}})$, where $\mathbf{y}$ is the *expected output* and $\hat{\mathbf{y}}$ is the *predicted output*. Some common loss functions are provided:

- **mean squared error (MSE)** [good for *regression tasks*]:

$$J_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- **cross-entropy loss** [good for *classification tasks*]:

$$J_{x-entropy} = -\frac{1}{N}\sum_{i}^{N} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

- **loss** + L_2**-regularization** [used for avoiding the **over-fitting** problem]:

$$J = J_0 + \frac{\lambda}{2N}\sum_{i=1}^{N} w_i^2$$

  where J_0 is the chosen loss function (*MSE, X-entropy, ...*)

4.4.4. Backpropagation algorithm

Backpropagation is a form of **reverse-mode automatic differentiation** used to compute the **gradient** of the loss function with respect to each **weight** and **bias**:

1. **forward pass**: computes the network's output;
2. **loss computation**: evaluates how far the prediction is from the target;
3. **backward pass**: computes gradient of the loss function w.r.t. each parameter using the chain rule:

$$\frac{\partial J}{\partial W^{(l)}} = \frac{\partial J}{\partial x^{(l)}} \cdot \frac{\partial x^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}}$$
$$\frac{\partial J}{\partial W^{(l)}} = \frac{\partial J}{\partial x^{(l)}} \cdot \frac{\partial x^{(l)}}{\partial z^{(l)}} \cdot \frac{\partial z^{(l)}}{\partial W^{(l)}}$$

4. **parameters update**: adjusts weights and biases using **gradient descent** (or its variants):

$$W^{(k+1)} \leftarrow W^{(k)} - \eta \frac{\partial J}{\partial W^{(k)}}$$

$$\mathbf{b}^{(k+1)} \leftarrow \mathbf{b}^{(k)} - \eta \frac{\partial J}{\partial \mathbf{b}^{(k)}}$$

where η is the **learning rate**.

4.4.5. RMSProp (Root Mean Squared Propagation algorithm)

RMSProp is the proposed algorithm for minimizing the loss function and optimizing the weights and biases in the experiment: unlike the other algorithms, this was the best one that gave the relationship between time complexity and optimization of the various evaluation metrics (*accuracy, recall, precision, f1-score, roc-auc*).

The pseudocode of the algorithm is provided:

Algorithm 4.1 RMSProp

Require: J: loss function
$epochs$: number of iterations
$batch\_size$: size of mini-batches
η: learning rate
γ: decay rate
ϵ: smoothing parameter

(Initialize weights $W^{(0)}$ and biases $\mathbf{b}^{(0)}$ if not already done)
Initialize the cumulated square gradient $\mathbb{E}[\nabla J^2]^{(0)} = 0$
▷ ∇J w.r.t. both weights and biases
k $\leftarrow$ 1
for k $\leq$ epochs+1 **do**
  $(X_{batch}, y_{batch}) \leftarrow \text{Random}(X_{train}, y_{train}, batch\_size)$
  $g \leftarrow \nabla J(X_{batch}, y_{batch}, W^{(k-1)}, \mathbf{b}^{(k-1)})$
  $\mathbb{E}[g^2]^{(k)} = \gamma \mathbb{E}[g^2]^{(k-1)} + (1-\gamma) g^2$ ▷ g w.r.t. both weights and biases
  $W^{(k)} \leftarrow W^{(k-1)} - \frac{\eta}{\sqrt{\mathbb{E}[g^2]^{(k)} + \epsilon}} g$ ▷ Here, g w.r.t. weights
  $\mathbf{b}^{(k)} \leftarrow \mathbf{b}^{(k-1)} - \frac{\eta}{\sqrt{\mathbb{E}[g^2]^{(k)} + \epsilon}} g$ ▷ Here, g w.r.t. biases
  k $\leftarrow$ k + 1
end for
return $W, \mathbf{b}$

4.4.6. Implementation

As done for all the other classifiers, we used an Object Oriented approach.

The constructor initializes the activation function (`act_func`) to use in all layers (default `jnp.tanh`) but the last one and the activation function (`out_act_func`) to use in the output layer (default `jax.nn.sigmoid`); it initializes the `layers_size` (array of weights and biases alternated) and which type of optimization algorithm to use during the training phase (`optimizer`). The main functions used in the experiment are:

- `initialize_parameters`: to initialize randomly the values of weights and to set biases to zero;
- `regularized_loss`: the cost function used in the experiment, which uses the loss function and the penalization term of the regularization passed as parameters;
- `RMSprop`: the optimization algorithm used in the experiment, with parameters the *loss function*, *epochs* (number of iterations), *batch size*, *learning rate*, *decay rate* and *epsilon* (for avoiding division by zero)
- `fit`: used for training the neural network using the optimization algorithm set for adjusting the weights and biases (`params`) on the training samples' set X and labels `y`
- `predict`: used to predict the hard labels and probabilities of belonging class of the given set of samples X)

4.5. Logistic regression

Logistic regression is a statistical baseline supervised ML algorithm for binary classification problems, where the goal is to predict the probability of an instance/sample belonging to a particular class.

This method of *classification* (and not *regression* as written in the name of the algorithm) has a very close relationship with neural networks: indeed, an artificial neural network can be viewed as a series of logistic regression classifiers stacked on top of each other.

4.5.1. Components

Like *Naive Bayes*, logistic regression is a **probabilistic classifier** that uses these 3 components:

1. A **classification function** that computes $\hat{y}$ (the estimated/predicted class), via $p(y|x)$. In general, the logistic regression uses the **logistic (or sigmoid) function** to map predicted values to probabilities. The sigmoid function is defined as:

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}$$

where z is a linear combination of the input features. For a given input vector $[X_1, X_2, ..., X_n]$, the linear combination is:

$$z = w_0 + \sum_{i=1}^{n} w_i X_i = w_0 + w_1 X_1 + w_2 X_2 + ... + w_n X_n$$

Here, w_0 is the *intercept (bias)* term, and $w_1, w_2, .., w_n$ are the *coefficients (weights)* corresponding to the input features.

The decision boundary is the **threshold** that separates the two classes. Typically, if $\hat{y} \geq 0.5$, the sample is classified as class 1, otherwise it is classified as class 0.

2. An **objective/cost function** corresponding to error on training examples that we want to minimize.

The cost function used for optimizing the model parameters is the **cross-entropy function** (already introduced in the ANN section):

$$J_{x-entropy} = -\frac{1}{N} \sum_{i}^{N} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

In addition, a **regularization** term can be added in order to overcome the **overfitting** problem.

3. an **optimization algorithm** for **minimizing** the objective function (*Stochastic Gradient Descent, RMSProp, ...*).

4.5.2. Implementation

The implementation is similar to the ANN's one, so it's not necessary to report it again: the only difference is in the constructor, which sets only the optimization algorithm to use during the training phase (`optimizer`). Moreover, it uses almost all the main functions of ANN's class.

5 | Genetic Algorithm

5.1. Overview

The first appearance of genetic algorithms (GAs) was on J. D. Bagley's thesis *"The behaviour of adaptive systems which employ genetic and correlative algorithm"*. He described them like this: *"[...] genetic algorithms are simulations of evolution, of what kind ever. In most cases, however, genetic algorithms are nothing else than probabilistic optimization methods which are based on the principles of evolutions"*.

In general, the problem GAs are used to solve is to find an $x_{\in}X$ such that an arbitrary $f : X \longrightarrow \mathbb{R}$ is maximal, i.e. $f(x_0) = \max_{x \in X} f(x)$. Depending on the actual problem, it can be quite difficult to obtain such point and it might be sufficient to have a local maximum or to be as close as possible to the global maximum. The function f is called *fitness* because it assigns values to individuals $x \in X$ so that we can compare them. Reproduction and adaption are carried out at genetic information level, so GAs do not operate on the values of the search space X, but on some coded versions of them (like strings or arrays).

Assume S to be a set of strings (or arrays) in general with underlining grammar and let X be the search space of the above optimization problem, then the function $c : X \longrightarrow S$ is called a *coding function*; conversely: $c^{-1} : S \to X$ is called a *decoding function*. In practice, coding and decoding functions have to be specified according to the needs of the actual problem.

The transition from one generation to the next consists of four basic components:

- **selection**: choose the individuals to be reproduced according to their fitness values.
- **crossover**: merge the genetic information of two individuals (i.e. their chromosomes); if implemented correctly, good parents produce good children.
- **mutation**: in GA, mutation is realized as a random deformation of the chromosome array with a certain probability. The positive effects are the preservation of genetic diversity and avoiding of local maxima.

- **sampling**: it's the procedure that computes a new generation from the previous one and its offsprings.

In general, genetic algorithms do not use any auxiliary information about the objective function such as its gradient, so they can be applied to any kind of continuous or discrete optimization problem. Moreover, since they work on the whole population and not on a single point at time (like GD), they are more robust and have more chance to find the global optimum.

5.2. Algorithmic framework

The paper doesn't describe in depth all the parts of the GA and the design choices; we have made them using what we learnt, maybe other choices could be more effective.

The starting dataset is composed of F features, so X will be the *power set* of F:

$$X = \mathcal{P}(F) = \{A \,|\, A \subseteq F\}$$

Chromosomes are represented as arrays of length $|F|$ ($b = \{0,1\}^{|F|}$) where i-th gene represents whether feature f_i is considered by the individual. Coding and decoding functions come naturally:

$$c(A) = (b_1, b_2, \ldots, b_{|F|}), \quad \text{where } b_i = \begin{cases} 1, & \text{if } f_i \in A, \\ 0, & \text{otherwise.} \end{cases}$$

$$c^{-1}(b) = \{f_i \in F \mid b_i = 1\}$$

Authors suggested to take as fitness function the accuracy ϕ.

At each timestamp t, we'll consider a fixed size population with m individuals:

$$\mathcal{B}_t = (b_{1,t}, b_{2,t}, \ldots, b_{m,t})$$

The algorithm developed in this notebook is the following:

Algorithm 5.1 Genetic Algorithm (GA)

Require: ϕ_m: min fitness to reach
m: population size
F: initial set of features
k: total number of iterations
p_m: probability of mutation
p_c: probability of crossover
n: number of crossover points
X_{train}: matrix with training samples
y_{train}: vector with training labels
X_{test}: matrix with testing samples
y_{test}: vector with testing labels

$t \leftarrow 0$
Compute initial population $\mathcal{B}_0 = (b_{1,0}, \ldots, b_{m,0})$
Compute fitness for each individual $b_{i,0}$
Compute maximum fitness ϕ_0 for $\mathcal{B}_0$
while $\phi_t < \phi_m$ **and** $t < k$ **do**
    for $i \leftarrow 0$ to $m - 1$ **do**
        Select an individual $b_{i,t+1}$ from $\mathcal{B}_t$
    end for
    for $i \leftarrow 0$ to $m - 2$ **step** 2 **do**
        if Random[0,1] $\leq p_c$ **then**
            Cross $b_{i,t+1}$ with $b_{i+1,t+1}$
        end if
    end for
    for $i \leftarrow 0$ to $m - 1$ **do**
        Eventually mutate $b_{i,t+1}$ with probability p_m
    end for
    for $i \leftarrow 0$ to $m - 1$ **do**
        If there are $b_{i,t+1} = 0$, insert in random positions some 1 (at least one)
    end for
    $t \leftarrow t + 1$
    Compute fitness for each individual $b_{i,t}$
    Compute maximum fitness ϕ_t for $\mathcal{B}_t$
end while
return $b_{i,t}$ with maximum ϕ and its fitness

where the sampling policy is: replace selected individuals with one of children and all the others die immediately. The genetic operations are:

- **selection**: it can be deterministic, but in most implementations it has random components. The one used in this notebook is *roulette wheel* (or *proportional selection*) where the probability to choose a certain individual is proportional to its fitness:

$$P\{b_{i,t} \text{ is selected}\} = \frac{\phi(b_{i,t})}{\sum_{k=1}^{m} \phi(b_{k,t})}$$

  It's called *roulette* because it resembles a roulette game where the slots aren't equally wide.

- **crossover**: it's the exchange of genes between chromosomes of the two parents. We can realize it using different techniques:
  - *single point*: the two vectors are cut at a randomly chosen position and the two tails are swapped.
  - *multiple points crossover*: instead of only one, n breaking points are randomly; then, every second section is swapped.

Algorithm 5.2 Crossover

Require: p_1: first parent
p_2: second parent
n: number of crossover points

Create two empty arrays c_1, c_2 for the children
Generate n distinct positions (P) in range $[0, m-1]$ sufficiently far away from each other and sort them
swap $\leftarrow$ false
start $\leftarrow 0$
for $c \in P \cup \{n\}$ **do**
    if swap **then**
        c_1[start: c] $\leftarrow$ p_2[start: c]
        c_2[start: c] $\leftarrow$ p_1[start: c]
    else
        c_1[start: c] $\leftarrow$ p_1[start: c]
        c_2[start: c] $\leftarrow$ p_2[start: c]
    end if
    swap $\leftarrow$ **not** swap
    start $\leftarrow c$
end for
return c_1, c_2

- **mutation**: in reality, the probability that a certain gene is mutated is almost equal for all genes, so for the i-th gene if random$[0, 1] \leq p_m$ we invert it.

5.3. Implementation

For simplicity, we developed a class for the algorithm, called `GeneticAlgorithm`. In the constructor, the user can specify all the parameters required for algorithms 5.1 and 5.2. Also, the model to be used to create individuals and its attributes must be passed. As the reader can notice from the `.py` files, all models extend an abstract class called `BaseModel`; in this way, we are sure that all of them contain and have common signature and return values of `fit` and `predict` methods that are the only ones used by the GA. Attributes should be passed as dictionary since at instantiation time, the code does a dict-unpacking (`**self.model_params`) operation. By default, `__init__()` sets accuracy (`utils.model_evaluation.accuracy`) as fitness function, but the user can specify the

one they prefer as long as it accepts two arguments (y_{pred}, y_{test} in this order). So, for example, one can use the precision, or can combine it with recall weighting more the last (`precision(x, y) + 2 * recall(x,y)`).

The exposed methods are the following:

- `fitness`: it takes the split dataset, instantiates a `BaseModel` by unpacking its parameters, fits it to training data and tests it, returning the fitness (`self.fit_funct (y_pred, y_test)`).
- `crossover`: it takes two parents, cross them on n = `self.num_crs_points` and return the children. To avoid useless split indexes (like i and $i + 1$), once i is extracted, $[i - 2, i - 1, i, i + 1, i + 2]$ cannot be selected any more.
- `spawn_threads_fitness`: a rule of thumb for features selection GA is that the population size should be at least 3 times the number of features to choose from. Fitting and predicting with a simple RF model for 100 each generation is computationally heavy if done sequentially, so we spawn a bunch of threads to parallelize the process. We first compute the feature vectors to assign to them and then we start them with `executor.map()`. We don't need to do a `.shutdown()` because it's automatically handled by `with` statement.
- `run`: the main method for GA. It follows the structure of algorithm 5.1 with some performance enhancement.
- `decode`: utility to decode a chromosome into a list of strings.
- `encode`: utility to encode a list of strings into a chromosome.

6 | Metrics

The metrics used to evaluate our models are the most common ones in ML. The aim of the paper was to classify accurately fraudulent transaction, so we define:

- True Positive (TP): number of fraudulent transactions classified as so.
- True Negative (TN): number of non-fraudulent transactions classified as so.
- False Positive (FP): number of non-fraudulent transactions classified as fraudulent.
- False Negative (FN): number of fraudulent transactions classified as non-fraudulent.

Some stakeholders, such as banks, might be more interested in reducing FN, others, like normal users, might be more interested in FP since they don't want their usual transactions to be blocked. Each model offers a trade-off between the two that can be more or less marked basing on the hyper-parameters' values: by changing some of them, in fact, we're able to increase some metrics at the cost of decreasing some others.

Using these values, the adopted metrics are:

- Accuracy (AC): the percentage of correctly classified transactions, either fraudulent or not.
$$AC = \frac{TN + TP}{TP + TN + FP + FN}$$
- Recall (RC): the rate of correctly classified frauds over the total number of frauds.
$$RC = \frac{TP}{FN + TP}$$
- Precision (PR): the rate of correctly classified frauds over the total number of transactions classified as frauds.
$$PR = \frac{TP}{FP + TP}$$

- F-Score (F1): measure how well we're classifying samples basing on RC and PR.

$$F1 = 2 \cdot \frac{PR \cdot RC}{PR + RC}$$

Finally, we computed ROC curves (Receiver Operating Characteristic) and their associated AUC score (Area Under the Curve). ROC are a visual representation of model performance across all thresholds: if the curve is under the line `fpr` = `tpr`, it means that the classifier is performing worse than chance. AUC measures how good the classification is: it ranges from 0 (worst) to 1 (best).

A dedicated `.py` file is used to compute all this metrics. It offers methods to compute a single metric, confusion matrices and Pandas DF to aggregate the results. In it, also, we have defined the features vectors used in the paper `feature_vectors` along with their corresponding masks `feature_masks`.

7 | Results

In this chapter, we report the results obtained with each classifier on the selected feature vectors. For each of them, we show:

- A table with the comparison of our values to the ones obtained in the reference paper, which are shown in grey. This table contains, also, our AUC values.
- ROC curve for each feature vector.

Used feature vectors are presented in the following table:

Feature Vector	**Variables**
v_1	V_1, V_5, V_7, V_8, V_{11}, V_{13}, V_{14}, V_{15}, V_{16}, V_{17}, V_{18}, V_{19}, V_{20}, V_{21}, V_{22}, V_{23}, V_{24}, *Amount*
v_2	V_1, V_6, V_{13}, V_{16}, V_{17}, V_{22}, V_{23}, V_{28}, *Amount*
v_3	V_2, V_{11}, V_{12}, V_{13}, V_{15}, V_{16}, V_{17}, V_{18}, V_{20}, V_{21}, V_{24}, V_{26}, *Amount*
v_4	V_2, V_7, V_{10}, V_{13}, V_{15}, V_{17}, V_{19}, V_{28}, *Amount*
v_5	*Time*, V_1, V_7, V_8, V_9, V_{11}, V_{12}, V_{14}, V_{15}, V_{22}, V_{27}, V_{28}, *Amount*
v_6	*Time*, V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8, V_9, V_{10}, V_{11}, V_{12}, V_{13}, V_{14}, V_{15}, V_{16}, V_{17}, V_{18}, V_{19}, V_{20}, V_{21}, V_{22}, V_{23}, V_{24}, V_{25}, V_{26}, V_{27}, V_{28}, *Amount*
v_7	V_1, V_2, V_3, V_4, V_5, V_6, V_7, V_8, V_9, V_{10}, V_{11}, V_{12}, V_{13}, V_{14}, V_{15}, V_{16}, V_{17}, V_{18}, V_{19}, V_{20}, V_{21}, V_{22}, V_{23}, V_{24}, V_{25}, V_{26}, V_{27}, V_{28}, *Amount*

Table 7.1: Feature vectors and their corresponding variables. Subscripts indicate variable indices.

7.0.1. Decision Trees

Feature vector	Accuracy	Recall	Precision	F1-Score	AUC
v_1	99.94	75.51	87.40	81.02	0.86
	99.92	75.22	75.22	75.22	
v_2	99.93	73.46	85.04	78.83	0.92
	99.87	68.14	60.62	64.16	
v_3	99.93	68.03	87.72	76.63	0.86
	99.90	76.10	68.80	72.26	
v_4	99.94	73.47	87.10	79.70	0.87
	99.91	76.10	72.26	74.13	
v_5	99.94	76.19	91.80	83.27	0.88
	99.89	72.56	65.07	68.61	
v_6	99.94	74.83	89.43	81.48	0.91
	96.91	76.10	71.07	73.50	
v_7	99.94	74.15	87.90	80.44	0.91
	89.91	79.64	68.70	73.77	

Table 7.2: Performance metrics for Decition Tree

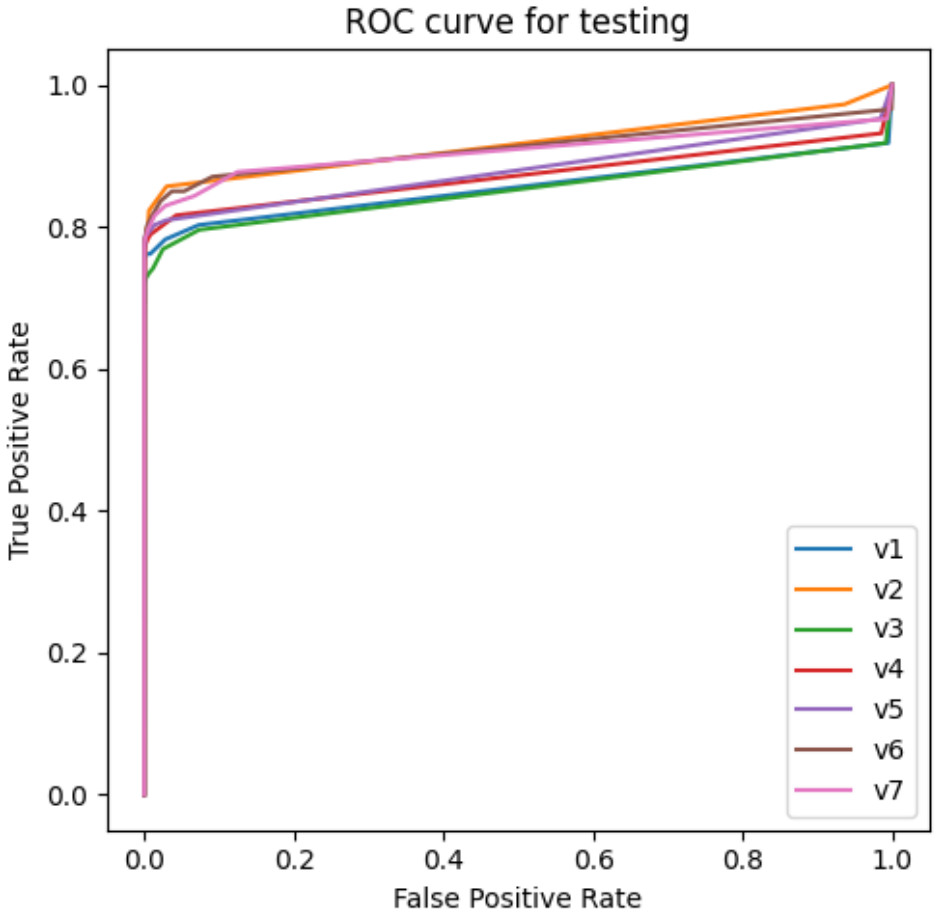

Figure 7.1: Testing ROC for Decision Tree

7.0.2. Random Forest

Feature vector	Accuracy	Recall	Precision	F1-Score	AUC
v_1	99.95	80.95	91.53	85.92	0.96
	99.94	76.99	89.69	82.85	
v_2	99.94	78.23	81.08	87.12	0.92
	99.93	76.10	82.69	79.26	
v_3	99.94	79.59	87.97	83.57	0.95
	99.94	75.22	85.85	80.18	
v_4	99.94	80.27	88.06	83.99	0.95
	99.94	77.87	83.80	80.73	
v_5	99.96	80.95	92.25	86.23	0.94
	99.98	72.56	95.34	82.41	
v_6	99.95	82.31	89.63	85.82	0.96
	87.95	77.87	92.63	84.61	
v_7	99.95	80.95	88.81	84.70	0.95
	83.78	79.64	92.78	85.71	

Table 7.3: Performance metrics for Random Forest

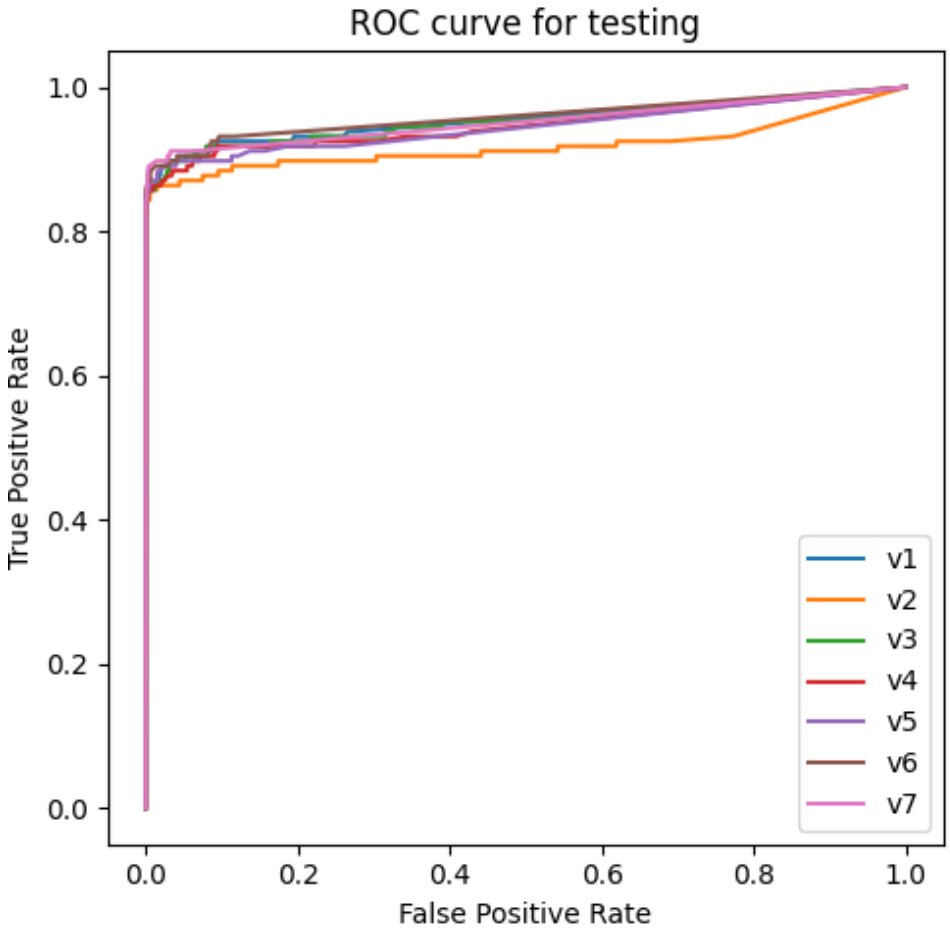

Figure 7.2: Testing ROC for Random Forest

7.0.3. Naive Bayes

Feature vector	Accuracy	Recall	Precision	F1-Score	AUC
v_1	98.10	83.67	7.15	13.17	0.96
	98.13	84.95	6.83	12.65	
v_2	98.49	70.07	7.65	13.8	0.94
	98.65	77.87	8.59	15.47	
v_3	98.69	80.95	9.85	17.56	0.95
	98.81	81.41	10.07	17.93	
v_4	98.31	75.51	7.29	13.29	0.96
	98.48	81.41	7.97	14.53	
v_5	99.38	58.50	15.61	24.64	0.96
	99.4	57.52	15.85	24.85	
v_6	99.30	64.63	14.82	24.11	0.97
	80.31	64.60	13.95	22.95	
v_7	97.84	83.67	6.32	11.76	0.96
	78.14	83.18	6.73	12.46	

Table 7.4: Performance metrics for Naive Bayes.

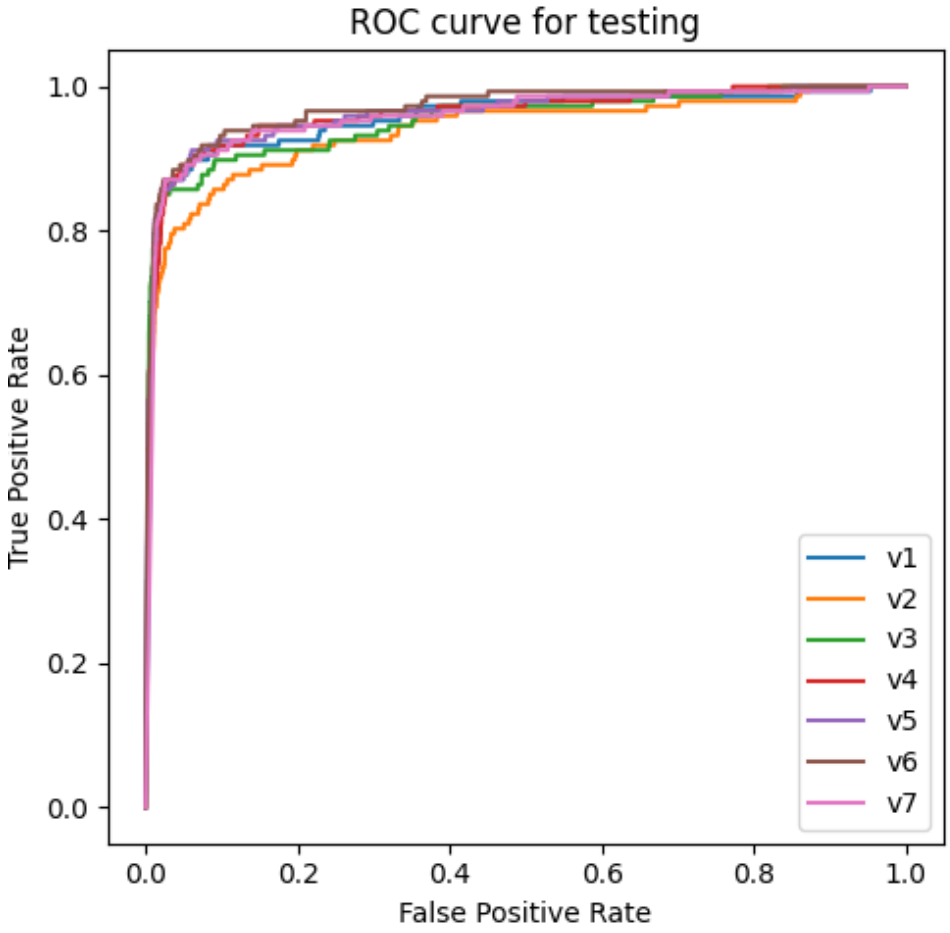

Figure 7.3: Testing ROC for Naive Bayes

7.0.4. Artificial neural network

Feature vector	Accuracy	Recall	Precision	F1-Score	AUC
v_1	99.92	67.33	83.95	74.73	0.95
	99.94	77.87	84.61	81.10	
v_2	99.91	66.34	79.76	72.43	0.81
	99.91	66.37	76.53	71.09	
v_3	99.92	76.24	80.21	78.17	0.93
	99.91	67.25	77.55	72.03	
v_4	99.92	75.25	80.85	77.95	0.85
	99.91	61.06	81.17	69.69	
v_5	99.92	83.17	75.68	79.25	0.96
	99.08	77.87	12.27	21.20	
v_6	99.93	81.19	79.61	80.39	0.98
	97.80	74.33	42.85	54.36	
v_7	99.93	82.18	80.58	81.37	0.97
	88.93	78.76	82.40	80.54	

Table 7.5: Performance metrics for Artificial Neural Network

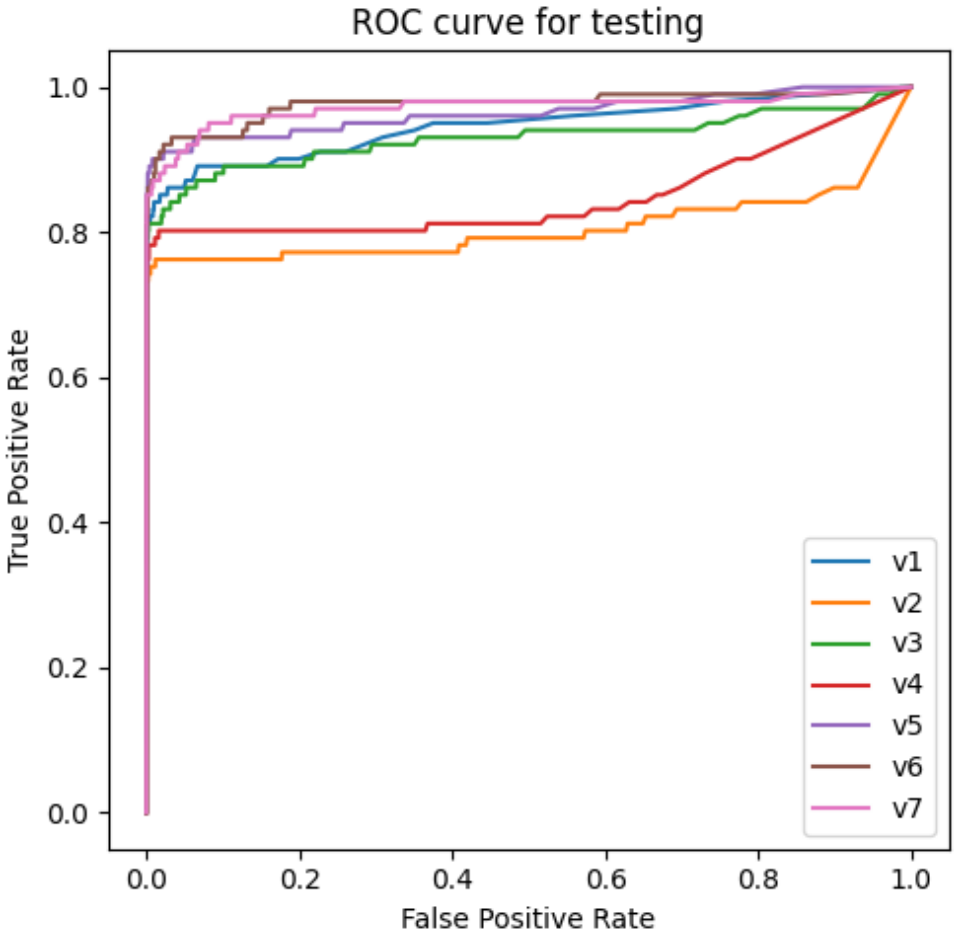

Figure 7.4: Testing ROC for Artificial Neural Network

7.0.5. Logistic Regression

Feature vector	Accuracy	Recall	Precision	F1-Score	AUC
v_1	99.91	56.43	85.07	67.86	0.92
	99.91	57.52	82.27	67.70	
v_2	99.88	35.64	87.80	50.70	0.81
	99.89	47.78	79.41	59.66	
v_3	99.90	54.46	85.94	66.67	0.90
	99.90	53.09	80.00	63.82	
v_4	99.87	30.69	86.11	45.26	0.85
	99.89	46.90	77.94	58.56	
v_5	99.92	63.37	87.67	73.56	0.96
	99.77	46.70	34.64	39.84	
v_6	99.93	68.32	89.61	77.53	0.98
	93.88	60.17	62.96	61.53	
v_7	99.93	67.33	90.67	77.27	0.98
	79.91	59.29	81.70	68.71	

Table 7.6: Performance metrics for Logistic Regression

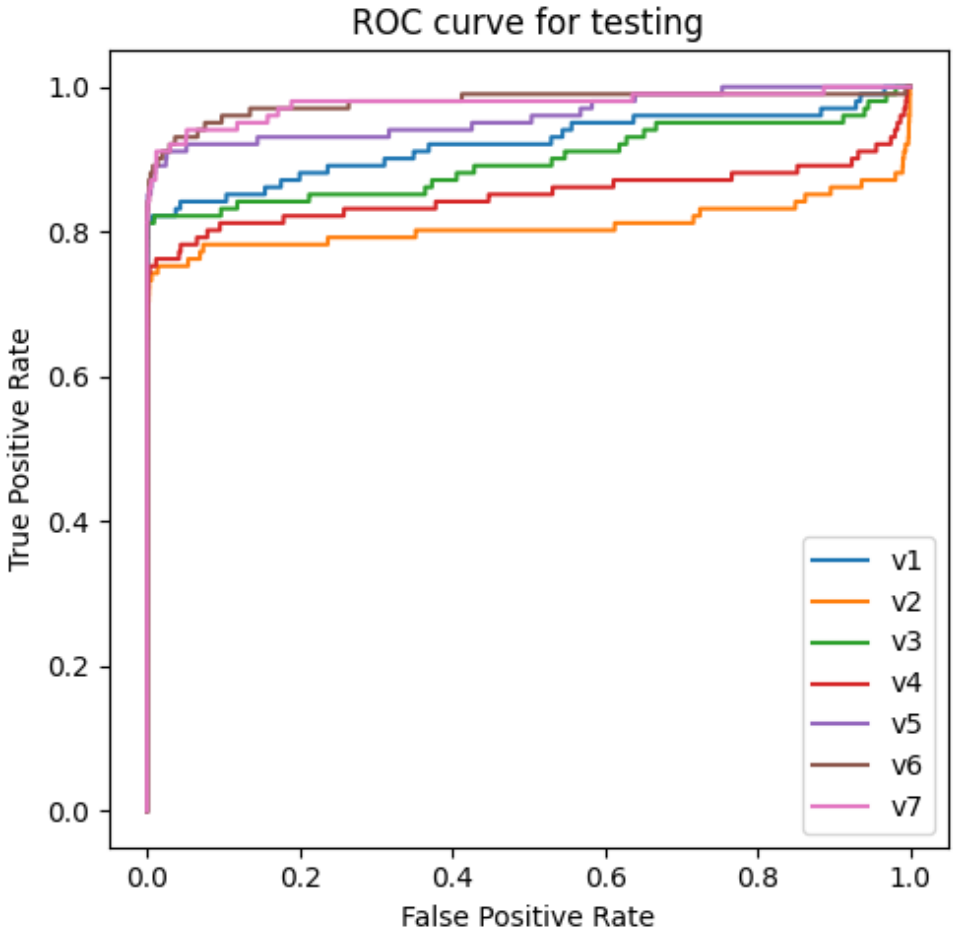

Figure 7.5: Testing ROC for Logistic Regression

7.1. Feature selection

For feature selection, the results are reported in the following table:

Model Type	Fitness Function	Feature Vector	Fitness Value
Naive Bayes	`accuracy(x, y)`	$Time$, V_4, V_9, V_{11}, V_{16}, V_{17}, V_{18}, V_{24}, V_{26}, V_{27}	0.9988
Naive Bayes	`2.0 * accuracy(x, y) + precision(x, y)`	V_4, V_5, V_6, V_{10}, V_{14}, V_{15}, V_{16}, V_{17}, V_{18}, V_{21}, V_{22}, V_{26}, V_{28}, $Amount$	2.8357
Random Forest	`accuracy(x, y)`	V_2, V_4, V_6, V_7, V_8, V_9, V_{10}, V_{12}, V_{13}, V_{14}, V_{15}, V_{16}, V_{18}, V_{19}, V_{21}, V_{23}, V_{25}, V_{27}, $Amount$	0.9996
Decision Tree	`accuracy(x, y)`	$Time$, V_1, V_2, V_4, V_5, V_6, V_7, V_9, V_{10} V_{11}, V_{12}, V_{13}, V_{14}, V_{15}, V_{16}, V_{18}, V_{20}, V_{21}, V_{22}, V_{23}, V_{24}, V_{27}	0.9995
Artificial Neural Network	`accuracy(x, y)`	$Time$, V_1, V_5, V_6, V_7, V_8, V_{10}, V_{11}, V_{13} V_{14}, V_{16}, V_{18}, V_{19}, V_{20}, V_{21}, V_{24}, V_{25}, V_{26}	0.9995
Logistic Regression	`accuracy(x, y)`	V_3, V_4, V_5, V_6, V_8, V_9, V_{10}, V_{11}, V_{12} V_{13}, V_{14}, V_{15}, V_{16}, V_{17}, V_{20}, V_{21}, V_{24}, V_{25}, V_{26}, $Amount$	0.9993

Table 7.7: Feature vectors obtained by the GA for each model.

8 | Conclusions

Before expressing our conclusions, we have to make some important considerations. First of all, researchers found $v_1, \ldots, v_5$ applying the GA with Random Forest; this means that there might be other choices that maximize accuracy for the other models. We have proven this using our implementation of GA: with accuracy as fitness function, NB reaches 99.88%, DT 99.95%, ANN 99.95% and LR 99.93% which are higher than all the others found by us and the researchers.

The primary focus of the paper was accuracy, but recall and precision are very important too. We have only 492 frauds over more than $250k$ total transactions, so a classifier that classifies all samples as non-fraud would get a fairly high accuracy (like 99.8%). Some stakeholders, such as banks, might be more interested in reducing FN, while others, like normal users, might be more interested in FP since they don't want their usual transactions to be blocked.

Another important observation is that authors have used Scikit-Learn library for all the model, while we have implemented them from scratch. Library' models are really optimized and they offer a vast variety of hyper-parameters whose tuning can positively impact on the performance. As an example, the reader can take DT: their library version exposes a lot of methods to prune them (like post-pruning) that are able to reduce the overfitting, thus increasing the performance. For the sake of simplicity, we didn't implement them.

Finally, in production or research environments, the hyper-parameters are estimated using methods like grid-search or cross-validation. We didn't implement them because in the paper it's not clear whether they have used them or not.

With $v_1, \ldots, v_5$, we observed strong metric values that align well with the results reported in the scientific paper. However, in the last two feature vectors (the full and the one with randomly selected features) we didn't see the significant drop in accuracy that the paper described. This means that while GA-based feature selection doesn't drastically enhance model performance, it does offer a clear advantage in execution speed due to the reduced data size.

Overall, the accuracy remains around 99.92%, with the exception of NB models, which struggle to reach this level and average around 98.6%.

NB didn't perform particularly well, as it can only be optimized for either high recall or high precision, but not both. In other words, it troubles in capturing the underlying structure of the data.

While RF generally outperforms DT, it also demands more memory and takes longer to train. DT, on the other hand, seems a good compromise between performance and training time: it is way faster to be used in the GA for feature selection and in eventual hyper-parameters estimation.

ANN performed as expected in the paper and takes less to train even if it has a lot of weights and biases, compared to LR which, by construction, requires more epochs and batch size for optimizing the weights and bias (so more computation time).

List of Algorithms

List of Figures

List of Tables

9 | References

- Reference paper: Ileberi, E., Sun, Y., Wang, Z. ***A machine learning based credit card fraud detection using the GA algorithm for feature selection***. J Big Data 9, 24 (2022). https://doi.org/10.1186/s40537-022-00573-8
- Credit card fraud detection dataset:

  https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud
- Notes from the course ***Numerical Analysis for Machine Learning***, A.A. 2024 - 2025, prof. Edie Miglio.
- https://towardsdatascience.com
- https://www.ibm.com/topics/decision-trees
- https://www.ibm.com/topics/random-forest
- https://www.youtube.com/@statquest
- https://www.youtube.com/@meanxai
- Daniel Jurafsky & James H. Martin. ***Speech and Language Processing***.

  https://web.stanford.edu/∼jurafsky/slp3/5.pdf
- Mitchell, T. ***Generative and discriminative classifiers: Naive Bayes and Logistic Regression***. Carnegie Mellon University (2017).
- https://scikit-learn.org/stable/modules/tree.html
- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
- https://scikit-learn.org/stable/modules/naive_bayes.html
- https://medium.com/towards-data-science/gaussian-naive-bayes-explained-a-visual-guide-with-code-examples-for-beginners-04949cef383c
- Brieman L., ***Random Forest***. Machine Learing (2001)

- Bodenhofer U., ***Genetic Algorithms: theory and applications*** (lecture notes). 3rd edition (2003 - 2004), pags: 17 - 31.