POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Numerical Analysis for Machine Learning - Project

Authors: **Enrico Simionato - 10698193**

Alberto Sandri - 10698469

Group Number: **34**

Academic Year: 2022-2023

Contents

Contents **i**

1 Introduction to the problem **1**

2 Dataset inspection **3**

3 Pre-processing **4**

3.1 Normalization . . . 4

3.2 Train/test split . . . 4

3.3 Undersampling . . . 4

4 Classifiers **5**

4.1 Logistic Regression . . . 5

4.2 Decision Trees . . . 6

4.3 Random Forest . . . 8

4.4 Naive Bayes . . . 9

4.4.1 Bayes Theorem and theoretical foundations . . . 9

4.4.2 Fitting of the model . . . 10

4.4.3 Computing the prediction . . . 10

4.5 Artificial Neural Network . . . 11

4.5.1 Cost functions . . . 13

4.5.2 Optimization methods . . . 14

4.5.3 Regularization . . . 16

4.5.4 Implementation choices and parameters . . . 17

5 Metrics **19**

6 Results **21**

7 Conclusion **29**

8 References **31**

1 | Introduction to the problem

Nowadays more and more payments are made by means of credit cards and no more by using physical money. States around the world are moving towards favouring card payments and discouraging physical payment methods in order to make transactions more traceable. Moreover, online payments and online exchanges are one of the most used ways for sending and receiving money. In this context, the number of frauds affecting this form of payment increased significantly in the last few years, mainly following the growth of online transactions. Therefore, it is of paramount importance to tackle this problem by using systems that are able to detect when a fraud is happening. One possible solution is to build machine learning models, trained on the available data and capable of discovering the underlying patterns among the frauds.

Relying on machine learning methods in this environment could be a good approach in order to derive automatic methods for dealing with card fraud detection problems. It is assumed that there is not any practical and known law that rules card frauds happening, even if it is reasonable that card frauds have something in common with each other. So, using machine learning models, which can deal with large amounts of data and that are able of finding connections that are not visible to the human eye, could be a good way to go.

Several approaches have been applied to find a good solution for the problem, in particular, we will refer to the research conducted by Emmanuel Ileberi, Yanxia Sun and Zenghui Wang explained in the paper **A machine learning based credit card fraud detection using the GA algorithm for feature selection**.

In this paper, the authors suggest to first applying a genetic algorithm to the entire dataset containing the transactions, labelled as fraud and not fraud, in order to select which features to use to build the models. In this way, the classifiers will focus on a subset of promising data, avoiding some features that could have a negative impact on the performance, speeding up also the training phase since the algorithms need to process less data.

After this first step, the researchers obtained five feature vectors that were used to build models using the framework Scikit-Learn. The classifiers they used are:

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

The theoretical foundations, the structure and way of working of the methods are explained in chapter 4.

In this project, we will start from the feature vectors selected by researchers and we will implement the classifiers from scratch, using just some basic libraries, and then compare the obtained results with the ones reported in the paper, showing that the 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. It is composed of 284807 samples of which just 492 are frauds and every transaction has 30 features. There is a clear unbalance in the dataset, in fact only 0,1727% of the samples are frauds and this makes it harder for a model to accurately determine when a fraud verifies.

Due to confidentiality, the features are anonymized so they are called `Time`, `V1`, `V2`, ..., `V28`, `Amount`. The last column of the dataset is `Class`, which represents the target to predict, and contains 1 to denote a fraud and 0 otherwise. The dataset was already pre-processed, so there are no null or missing values and it was also performed PCA on `V1`, `V2`, ..., `V28`, in order to maximize the variance of the features and minimize the covariance between them.

Further insights and plots are analyzed in the Notebook.

3 | Pre-processing

3.1. Normalization

To normalize the data the min-max scaling method is applied to each feature f, in this way all the data are rescaled in a bounded interval.

$$f_{scaled} = \frac{f - min(f)}{max(f) - min(f)}$$

This is done by using the function `min_max()`.

3.2. Train/test split

The data are ordered by increasing values of the feature `Time` that contains the seconds elapsed between each transaction and the first transaction in the dataset. To avoid that the order could impact the predictions we have shuffled the data. This operation can be performed using `train_split()` which also divides the data into train, validation and test datasets, given the percentages of data that should be present in each one. By default, the training dataset contains 70% of the data and 30% are in the test dataset. The validation has been used during the training phase of the classifiers to find good values of the hyperparameters.

3.3. Undersampling

The last step is to perform undersampling to handle the high imbalance of the dataset. Building a classifier on the entire dataset would lower the accuracy since it would be more difficult for a model to learn when a fraudulent transaction is happening with respect to a normal one and so it could easily get stuck in a minimum where it classifies all the transactions as normal. For this purpose we wrote the function `undersample()` that keeps all the frauds in the input dataset and, by setting the `ratio` parameter, allows to decide how many normal transactions have to be kept for each fraud.

4 | Classifiers

In this chapter, we explain how the binary classifiers employed in the research work and how we have implemented them from scratch using Python on Google Colab. The whole code is contained in the Notebook `Project_10698469_10698193.ipynb` and it is explained through the documentation, further explanations are given by comments.

Even if some methods have been implemented defining some functions and others using object-oriented programming, all classifiers go through two phases: in the first one the model is built using the train set, instead in the second one the model is used to predict the labels for a test set. During both phases, the metrics are computed in order to monitor the statistics of the created models.

4.1. Logistic Regression

Logistic regression is a widely used binary classifier. Considering the input sample $\mathbf{x}$, the predicted value y_{pred} associated with the sample is the sigmoid function computed in the scalar product between $\mathbf{x}$ and the weight vector $\mathbf{w}$ plus a bias b. The outcome y_{pred} is bounded between 0 and 1, so in this case it can be interpreted as the probability that a sample corresponds to a fraudulent transaction. If the output value is greater or equal to 0.5, then the input sample is classified as `fraud`, otherwise as a `normal` transaction.

$$z = w_0x_0 + w_1x_1 + ... + w_nx_n + b$$

$$y_{pred} = \frac{1}{1+e^{-z}}$$

$$label = \begin{cases} 1, & \text{if } y_{pred} \geq 0.5 \\ 0, & \text{otherwise} \end{cases}$$

The logistic regression process consists in fitting the weights and bias to minimize the loss on the dataset. In this case, the used cost function is the cross-entropy, which can be written in the following way since there are only two possible labels y:

$$J(\mathbf{w}, b) = -\frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} \alpha y_i \log y_{pred,i} + \beta(1 - y_i) \log(1 - y_{pred,i})$$

The cost function is weighted differently for each class through α and β. Since our focus is to be able to detect fraudulent transactions which are very few compared to the number of normal transactions by imposing $\alpha > \beta$ we can penalize more the errors made by wrongly classifying a fraud, increasing the recall. To minimize $J(\mathbf{w}, b)$ there is no closed-form solution, so we used the stochastic gradient descent (SGD) method with mini-batch and linear decay of the learning rate:

$$\lambda_k = \max\left(\lambda_{\min}, \lambda_{\max}(1 - \frac{k}{K})\right)$$

where k is the number of the current epoch and K is the number of epochs to reach $\lambda_{\min}$. We decided to use SGD with mini-batch to speed up the process and to reduce overfitting by picking randomly only a few samples at each iteration. We tried also more sophisticated optimization methods without seeing big improvements, so we chose to leave the SGD since it was performing quite well.

In the Notebook, using the function `SGD()` it is possible to train the model by choosing the number of epochs, $\lambda_{\min}, \lambda_{\max}, K$ and the size of the batch. This function calls `cross_entropy()` to compute the cost function. In the end, the predictions are computed through `predict_LR()`. These last two functions both rely on `sigmoid()`, which simply computes the logistic function.

4.2. Decision Trees

Decision trees (DT) are a type of classifier that can be used both for regression and classification tasks. As the name suggests the model consists of a tree made of one root node, many decision nodes with two children each and many leaf nodes.

Initially, the whole training set is considered in the root node. Then the tree is built by evaluating all possible splitting conditions at each node, so evaluating all the thresholds for all the features and finding the one that divides the values in the best way. The best branching is evaluated according to the information gain, which gives a measure of the reduction of impurity obtained by doing a certain split. The impurity quantifies the heterogeneity of the labels among the samples, so it is minimum when in a node all samples have the same labels, instead is maximum when they are in the same quantity.

The information gain (IG) can be computed using two different methods: one is the entropy (E) and the other is the Gini index (G). IG is computed as the difference between the entropy or Gini index evaluated before the branch and the one assumed after the branch. The best possible branch is the one that maximizes the information gain since it would minimize the entropy or Gini index of the children, which can assume an optimal value of 0.

$$IG = E(parent) - (w_{left} \cdot E(child_{left}) + w_{right} \cdot E(child_{right}))$$

$$E(node) = - \sum_{l \in labels} p(l) \cdot \log_2(p(l))$$

$$IG = G(parent) - (w_{left} \cdot G(child_{left}) + w_{right} \cdot G(child_{right}))$$

$$G(node) = 1 - \sum_{l \in labels} p(l)^2$$

$p(l) = \frac{\#samples\ with\ label\ l}{total\ samples}$, $w_{left} = \frac{\#samples\ on\ left\ child}{total\ samples}$, $w_{right} = \frac{\#samples\ on\ right\ child}{total\ samples}$

When the number of remaining samples on a node is less than a threshold decided a priori or all the values have the same label, then a leaf node is created and it is set as its value the label that has the most samples belonging to it in that node.

Once built, the tree is traversed for each test sample starting from the root until a leaf is reached. When the leaf node is reached its label is assigned as the class of the sample. Each decision node stores a feature and a threshold, while traversing the tree the value of the sample in the feature associated with the node is compared with the threshold of the node: if the value is lower the next node will be the left child, otherwise the right one.

To implement DT we opted for an object-oriented approach by building two classes: `Node` and `DecisionTree`. The Node simply stores the feature, threshold, left and right children and the label, this last attribute is stored only in leaf nodes. DecisionTree instead has two main functions: `train()` to build a decision tree model and `predict()` to get the predictions using the built model. There are also other auxiliary functions that are explained directly in the Notebook.

When creating a tree is possible to set some parameters that influence how the tree is built:

- maximum depth of the tree;
- number of features to evaluate while branching to build a tree;

- maximum number of thresholds to evaluate while branching to build a tree;
- minimum number of samples in a node to branch;
- the way to compute the information gain by choosing between `entropy` and `gini index`.

The first four parameters need to be tuned and have a huge impact on the time required to build a tree and to make predictions. In fact, using the vanilla process, so by evaluating at each node all the features and all the possible thresholds, it takes many minutes to build just one tree, instead with the parameters we found it takes much less time. We observed an important decrease in the computation time by setting some of these thresholds, without seeing big differences in the evaluation performances. Moreover setting a limit on the depth of the tree and a minimum number of samples to split to avoid creating a lot of leaves with just one sample, allows for avoiding overfitting. This is also reached thanks to the maximum number of features and samples that introduces stochasticity in the process since these values are picked randomly.

Actually, in our case, the choice of the mode to compute the information gain did not prove to have a great impact on the final result, but the entropy mode, which uses the logarithm could be computationally slower.

4.3. Random Forest

Random forest (RF) classifier heavily relies on DT, in fact as the name suggests a random forest model is composed of a set of decision trees. For each new sample, the prediction is made by evaluating it for each tree and then taking as a label the most common value. In our code, this is done using `predict()`.

With the function `train()` it is possible to build the model by choosing the number of trees that will compose the forest, setting also all the tunable hyperparameters of the DT defined before. Each tree is trained using a bootstrapped dataset, so using a dataset with the same dimensions as the original one but by sampling randomly it using replacement, to avoid using the same data for every tree.

Obviously, if the number of DTs increases also the computation time to train the model will increase.

4.4. Naive Bayes

Naive Bayes (NB) classifiers are a family of classifiers with theoretical foundations in the Bayes theorem and are one of the simplest ones. This method tries to approximate the distribution of the features for each possible class of the problem and uses these probability distributions in order to predict the outcome of a given sample.

4.4.1. Bayes Theorem and theoretical foundations

Given a set of random variables y, x_1, x_2, ..., x_n, Bayes theorem can be expressed as:

$$p(y|x_1, x_2, ..., x_n) = \frac{p(y) \cdot p(x_1, x_2, ..., x_n|y)}{p(x_1, x_2, ..., x_n)}$$

Assuming the independence of the random variables x_1, x_2, ..., x_n the previous probability can be written as:

$$p(y|x_1, x_2, ..., x_n) = \frac{p(y) \cdot \prod_{i=1}^{n} p(x_i|y)}{\prod_{i=1}^{n} p(x_i|y)} \propto p(y) \cdot \prod_{i=1}^{n} p(x_i|y)$$

In this context p(y) is called prior probability.

Let's then be x_1, x_2, ..., x_n the features of each sample in a classification problem and y the label representing the class to which the sample belongs. In this scenario the last relation allows us to find the probability of a sample to belong to a class given the distributions of the features of the samples belonging to that class and the probability of a sample to belong to it.

These distributions are in general unknown but can be approximated. Here, as it is done in the paper, it is used a Gaussian naive Bayes classifier which considers every feature x_i of the dataset to have a Gaussian distribution with mean the mean of the feature and with standard deviation the standard deviation of the feature. The means and the standard deviations can be estimated considering the empirical counterpart evaluated over the training set.

1. Mean of the feature of index i

$$\overline{X_i} = \frac{1}{n_{samples}} \sum_{j=1}^{n_{samples}} x_{ji}$$

2. Variance of the feature of index i

$$var_i = \frac{1}{n_{samples} - 1} \sum_{j=1}^{n_{samples}} (x_{ji} - \overline{X_i})^2$$

   This is the unbiased version of the variance, the other estimator of the variance is the one with $n_{samples}$ instead $n_{samples} - 1$ in the denominator.

3. Standard deviation of the feature of index i

$$\sigma_i = \sqrt{var_i}$$

In our code, the implementation of the method is done through the class `GaussianNaiveBayes`. This class has the method for creating a new model given some data and for using it for prediction purposes.

In the last part of the code, in section **Additional demonstration about the distribution of the features** we empirically prove that the Gaussian distribution is a very good fit for the distribution of the features.

4.4.2. Fitting of the model

The definition, also called fitting, of the model is done inside the constructor of the class `GaussianNaiveBayes`. In the `__init__()` function, the training set, obtained from the original one, is split into two sets of samples based on the class to which each sample belongs. Using the functions provided by `numpy`, means and standard deviations of all the given features for the samples belonging to both classes are separately computed.

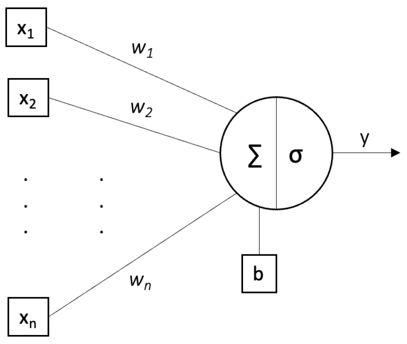
4.4.3. Computing the prediction

An initial remark has to be done: the method can be applied under the assumption of independence of the features of the dataset; in this case, the dataset has independent features since principal component analysis has been applied to it. It can be seen, in the **data inspection** section, that the correlation between the features is almost zero, as it is required, for all the couple of features, except for the `Amount` and `Time`. Besides this fact, their correlations are small. It is possible to perform again PCA in order to have a perfectly independent set of features. The prediction of the class of a given sample is implemented by means of the function `predict()`. The classification is performed simply by applying the theoretical results previously discussed and so by computing $p(y|x_1, x_2, ..., x_n)$ for both the two possible outcomes of y. The class with the highest probability will be the prediction of the sample. Another function has been defined in order to obtain these probabilities and it is `get_probability_positive()`, which returns the probability of a sample belonging to the positive class, namely to be a fraud in this case.

4.5. Artificial Neural Network

Artificial neural networks (ANN) are computational models composed of nodes, called neurons. Neurons are connected with each other through weighted links. These structures take inspiration from the biological shape of the brain and work in a similar way. This section explains some theoretical concepts about this topic, underlining how the method has been implemented in our code. The development of the algorithm is done through a set of Python functions which will be cited and briefly discussed.

The structure of a neuron is the following:



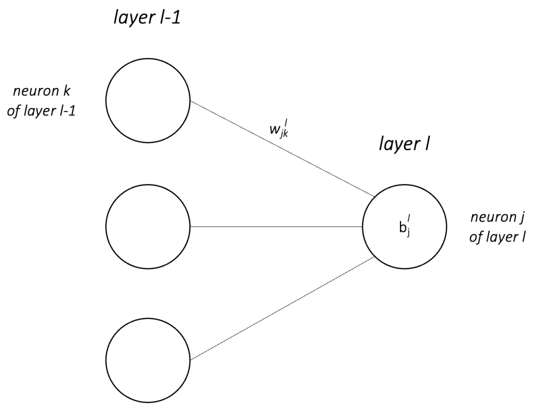
Therefore, a neuron is defined by its weight vector $\mathbf{w}$, its bias b and an activation function σ. $x_1, x_2, ..., x_n$ are the inputs of the neuron and y is the output. Inputs can also be seen as a vector $\mathbf{x}$.

The output of the neuron is computed as:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \sigma(\sum_{i=0}^{n} w_i \cdot x_i + b)$$

In general, a neural network can have many neurons divided into many layers. Usually, it is considered to have one input layer containing as many neurons as the number of the inputs and one output layer containing as many neurons as the dimension of the output. Between the input and output layers, there can be one or more layers, called hidden layers. Since we consider fully-connected neural networks every neuron of one layer is connected to all the neurons of the next layer, except for the last layer which has no outgoing arcs. Moreover, we consider feedforward neural networks in which loops inside the structure cannot be present.

In the following, the basics of the mathematical foundations of a neural network will be given.



1. b^l_j bias of the neuron j in layer l,

   $\mathbf{b}^l$ vector of the biases of layer l

2. w^l_{jk} weight of the arc connecting neuron k of layer $l-1$ to neuron j of layer l,

   W^l matrix of the weights of the arcs connecting layer $l-1$ and layer l

3. a^l_j activation of the neuron j in layer l

   $$a^l_j = \sigma(\sum_{k=0}^{size_{l-1}-1} w^l_{jk} a^{l-1}_k + b^l_j)$$

   with $size_{l-1}$ number of neurons of the layer $l-1$,

   $\mathbf{a}^l$ activation vector

   $$\mathbf{a}^l = \sigma(\mathbf{W}^l \cdot \mathbf{a}^{l-1} + \mathbf{b}^l)$$

These operations are computed for each layer from the input layer until the last layer, generating the output of the network.

Artificial neural networks are used for approximating a wide range of functions thanks to the theoretical results given by the universal approximation theorem.

The set of parameters of the neural network, namely the matrices of the weights and the vectors of the biases for each layer, is created by the function `initialize_parameters()` which takes as input the sizes, in terms of number of neurons, of each layer. This function also initializes the parameters: for the weights, it is adopted a random initialization with the values sampled from a standard Gaussian distribution; the biases instead are initialized to zero.

The computation done by the artificial neural network in our code is provided by the function `ANN()`, which takes as arguments the input of the neural network and its parameters and returns the output of the calculation.

Let's consider the classification of fraudulent transactions, the problem is solved as a supervised learning problem, with an ANN. It is given a set of transactions with their features and labelled as fraud or not fraud. The input of the artificial neural network is composed of the features of a transaction and on the output it gives the class to which the transaction belongs, or better, the probability to belong to a class. The problem of determining the mapping between the features and the class coincides, in the ANN case, to determine the values of the parameters that allow the best classification.

As in the other previously discussed classification methods, also in this one, the dataset is split into three parts, training set, validation set and test set.

4.5.1. Cost functions

The computation of the correct set of parameters is done through the solution of a minimization problem of a function that gives the measure of the discrepancy between the predictions and the correct labels. This function is called cost function.

Considering $n_{samples}$ the number of samples used as input, the cost functions we defined in the code are:

1. **Cross entropy for a binary classification problem**

$$J(\mathbf{W}, \mathbf{b}) = -\frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} \alpha y(\mathbf{x}_i) \log a_i^L + \beta(1 - y(\mathbf{x}_i)) \log (1 - a_i^L)$$

   With $y(\mathbf{x}_i)$ true label and a_i^L the computed prediction, considering L the last layer of the neural network. α and β are weights that give more or less importance to the error made in predicting a fraudulent transaction or a not fraudulent one. We define this cost function inside the Python function `cross_entropy()`.

2. **Cross entropy for a general classification problem**

$$J(\mathbf{W}, \mathbf{b}) = -\frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} \sum_{j=0}^{n_{outputs}-1} \alpha_j y_j(\mathbf{x}_i) \log a_{ji}^L$$

   With $\mathbf{y}(\mathbf{x}_i)$ true label (one-hot representation) and $\mathbf{a}_i^L$ the computed prediction, considering L the last layer of the neural network. α_j are weights that give more or less importance to the error made in predicting a fraudulent transaction or a not fraudulent one. We define this cost function inside the Python function `cross_entropy_general()`.

3. **Mean squared error**

$$J(\mathbf{W}, \mathbf{b}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} \frac{1}{2} \| y(\mathbf{x}_i) - a_i^L \|^2$$

   With $y(\mathbf{x}_i)$ true label and a_i^L the computed prediction, considering L the last layer of the neural network We define this cost function inside the Python function `MSE()`.

4. **Accuracy**

   Is the percentage of correctly classified samples. We define this cost function inside the Python function `accuracy()`.

4.5.2. Optimization methods

In order to minimize the cost function with respect to the parameters, since the parameters are a lot, an iterative method is the only solution. The procedure of applying the minimization method for finding the best approximation of the mapping between input and outputs is called training of the neural network. The following methods find their basics in the idea of updating the parameters in the inverse gradient direction in order to go in the direction that locally minimizes the cost function. The methods we used for finding the parameters that minimize the cost function are:

1. **Stochastic gradient descent (SGD)**

   Here we consider the stochastic gradient descent with mini-batch and adaptive learning rate, depending on the epoch of execution k. The recursive update of the parameter in the method is:

$$\mathbf{g}(\mathbf{x}^{(k)}) = \frac{1}{|I_k|} \sum_{i_k \in I_k} \nabla J_{i_k}(\mathbf{x}^{(k)})$$
$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \gamma^{(k)} \mathbf{g}(\mathbf{x}^{(k)})$$

with I_k set of randomly extracted indexes of samples and with the gradient computed with respect to the weights and biases of the network, these ones contained in the variable $\mathbf{x}$. Our implementation of the method is the function `SGD()`.

2. **Nesterov acceleration method (NAG)**

This method is similar to the stochastic gradient descent but at each iteration, the update is done taking into account the sum of the previous directions of the gradient. Here we consider the Nesterov acceleration method with mini-batch and adaptive learning rate, depending on the epoch of execution k. The recursive update of the parameter in the method is:

$$\mathbf{v}^{(k)} = \alpha \mathbf{v}^{(k-1)} + \gamma^{(k)} \frac{1}{|I_k|} \sum_{i_k \in I_k} \nabla J_{i_k}(\mathbf{x}^{(k)} - \alpha \mathbf{v}^{(k-1)})$$
$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{v}^{(k)}$$

with I_k set of randomly extracted indexes of samples and with the gradient computed with respect to the weights and biases of the network, these ones contained in the variable $\mathbf{x}$. Our implementation of the method is the function `NAG()`.

3. **RMSprop**

Also, this method works on the top of the stochastic gradient descent but at each iteration, the update is done taking into account the weighted sum of the previous gradients squared. In this way, the learning rate is automatically updated during execution. The recursive update of the parameter in the method is:

$$\mathbf{g}(\mathbf{x}^{(k)}) = \frac{1}{|I_k|} \sum_{i_k \in I_k} \nabla J_{i_k}(\mathbf{x}^{(k)})$$

$$\mathbf{r}^{(k+1)} = \rho \mathbf{r}^{(k)} + (1-\rho) \mathbf{g}(\mathbf{x}^{(k)}) \odot \mathbf{g}(\mathbf{x}^{(k)})$$

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \frac{\lambda}{\delta + \sqrt{\mathbf{r}^{(k+1)}}} \odot \mathbf{g}(\mathbf{x}^{(k)})$$

with I_k set of randomly extracted indexes of samples and with the gradient computed with respect to the weights and biases of the network, these ones contained in the variable $\mathbf{x}$. ρ is the rate of decay of the history of the gradients since past values of the gradient become less important when time passes, δ is a parameter used to avoid division by zero and λ is a sort of initial value for the learning rate. Our implementation of the method is the function `RMSprop()`.

The learning rate $\gamma^{(k)}$ is computed using a linear decay, already explained in the logistic regression classifier:

$$\gamma^{(k)} = \max\left(\gamma_{\min}, \gamma_{\max}(1 - \frac{k}{K})\right)$$

where k is the number of the current epoch and K is the number of epochs to reach $\lambda_{\min}$.

Within the optimization methods the gradients are computed using the automatic differentiation library `JAX`. `JAX` is also used for speeding up the code through the just-in-time compilation of the functions.

The chosen minimization method is applied to the neural network using as input the data contained in the training set.

During the training phase, the loss function has been computed also on the validation set in order to evaluate whether the model of the neural network is generalizing in a good way the input-output mapping.

The history of training, namely the values assumed by the cost function during the training step are stored, by means of the function `dump()`, and then plotted by the function `plot_history()`. `dump()` computes the values of the different cost functions in order to make a comparison between them possible. For speeding up the computations some cost function computations can be commented.

4.5.3. Regularization

In order to evaluate if a better result can be achieved we also use a regularization method, implemented through the functions `MSW()` and the `cost_regularization()`. This function is used as a cost function in the training phase in order to penalize neural networks with big weights since they are a symptom of overfitting. The mean square weight is defined as:

$$\mathrm{MSW}(\mathbf{W}) = \frac{1}{n_{weights}} \sum_{i=1}^{n_{weights}} w_i^2$$

Therefore the new cost function will be:

$$J_{regularization}(\mathbf{W}, \mathbf{b}) = J(\mathbf{W}, \mathbf{b}) + \beta \, \mathrm{MSW}(\mathbf{W})$$

where β is the regularization parameter.

4.5.4. Implementation choices and parameters

We choose to implement two slightly different neural networks: one with one scalar output representing the probability of the input vector to represent a fraudulent transaction and one with two scalar outputs representing the probability of the input vector to represent a fraudulent transaction and to represent a not fraudulent transaction respectively.

The implementation choices we adopted for the first neural network are the following:

1. Activation function of the neural network:
    - **hyperbolic tangent** on all the layers except for the last one
    - **sigmoid** on the last layer
2. Cost function: **cross entropy for a binary classification problem with regularization**
3. Weights for the class losses: **[2.8, 1]**
4. Layers size: **[input size, 30, 20, 1]**
5. Optimization method: **RMSprop**
    - Number of epochs: **2000**
    - Mini-batch size: **256**
6. **Small regularization weight**

The implementation choices we adopted for the second neural network are the following:

1. Activation function of the neural network:
    - **hyperbolic tangent** on all the layers except for the last one
    - **softmax** on the last layer
2. Cost function: **cross entropy for a general classification problem with regularization**

3. Weights for the class losses: **[3.2, 1]**
4. Layers size: **[input size, 30, 30, 10, 2]**
5. Optimization method: **RMSprop**
    - Number of epochs: **2500**
    - Mini-batch size: **256**
6. **Small regularization weight**

The set of hyperparameters and the type of design we chose has been driven by some heuristics and consideration but is also the result of many trials in guessing their best values. The set of parameters we found allows the neural network to reach good results in terms of the principal metrics we adopted. Modifying the parameters slightly better results might be found.

In the end, we decided to use as activation function `tanh` since only a few parameters were needed and not too many epochs were required to find a good model.

Although we tried other optimization methods, RMSprop was the best one in terms of results, requiring fewer epochs but a higher number of samples in the mini-batch. In the other methods, the tuning of the parameters was more difficult.

Regularization is not too useful in this case. Too high penalization makes higher the precision but lowers a lot the recall until making too small the weights. We use the small value of the penalization in order to avoid overfitting, it is not necessary but increases a bit the results.

5 | Metrics

In order to compare the results with the aforementioned paper, we decided to use the same performance metrics. In this context of card fraud detection, the aim is to classify accurately the fraudulent transactions using a binary classifier, so we define:

- true positive (TP) as the number of frauds correctly classified as frauds;
- true negative (TN) as the number of normal transactions correctly classified as normal;
- false positive (FP) as the number of normal transactions wrongly classified as frauds;
- false negative (FN) as the number of frauds wrongly classified as normal transactions.

Using these values the adopted metrics are:

- accuracy (AC) that is the percentage of accurately classified transactions, fraudulent and not fraudulent;
- recall (RC) that tells how many transactions are accurately classified as frauds on the total of fraudulent transactions;
- precision (PR) that tells how many transactions are accurately classified as frauds on the total of transactions that are classified as frauds;
- F1-Score (F1) that is a measure of the correctness of the classification based on RC and PR.

Expressing the metrics using their formulas:

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

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

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

$$F1_{score} = 2 \cdot \frac{PR \cdot RC}{PR + RC}$$

To compute all these values we implemented the function `metrics()`.

During the training phase, we noticed that by changing some hyperparameters it is possible to raise some values of these metrics to the detriment of others. Hence, the optimal values depend on what the research is focused on.

We computed also the receiver operating characteristic (ROC) curve that allows visualizing the performance of a model at different thresholds, with the associated area under the ROC curve (AUC). The range of values of AUC is between 0 and 1 and the higher the value of AUC, the better the classification model.

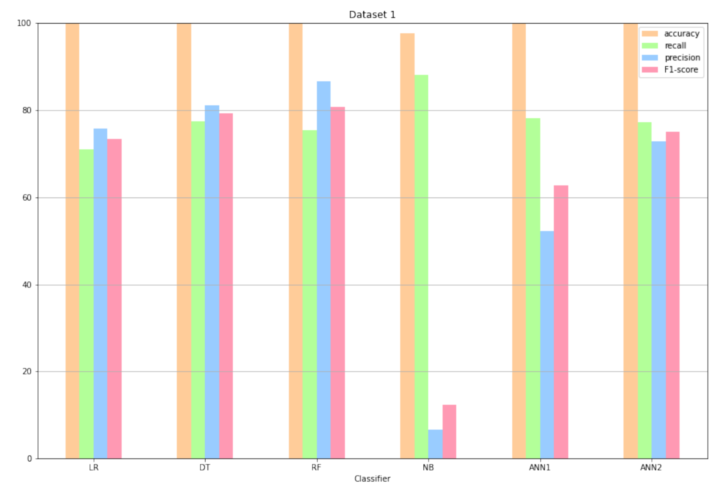
6 | Results

In this chapter, we report the results obtained with each classifier grouped by the datasets associated with the selected feature vectors. For each dataset, we show:

- a bar plot with the values of accuracy, recall, precision and F1-score obtained by every classifier on the test set;
- a table with the comparison of the previous values with respect to the ones obtained in the reference paper, which are shown in grey;
- ROC curve for each classifier with the associated value of the AUC.

In addition to the five datasets, also the full feature vector (dataset 6) and the feature vector generated with a random approach (dataset 7) are considered.

`ANN1` refers to the ANN with one output, instead `ANN2` to the ANN with 2 outputs.



Model	Accuracy	Recall	Precision	F1-Score
RF	99.93	75.48	86.67	80.69
	99.94	76.99	89.69	82.85
DT	99.93	77.42	81.08	79.21
	99.92	75.22	75.22	75.22
ANN1	99.83	78.22	52.32	62.70
ANN2	99.91	77.23	72.90	75.00
	99.94	77.87	84.61	81.10
NB	97.77	88.12	6.62	12.31
	98.13	84.95	6.83	12.65
LR	99.91	70.97	75.86	73.33
	99.91	57.52	82.27	67.70

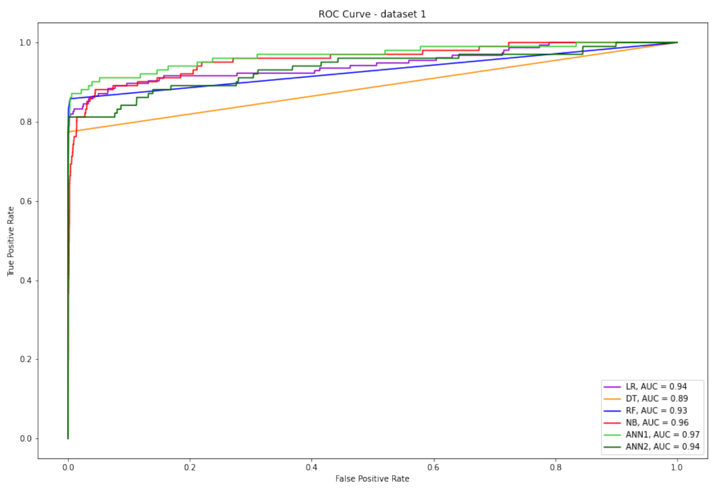
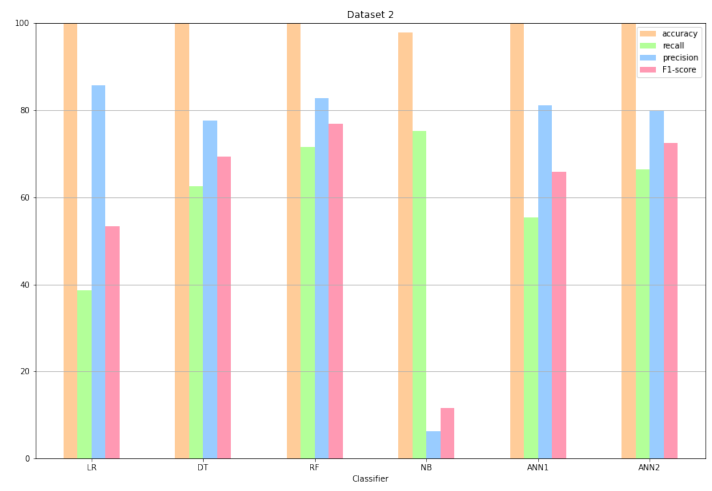


Figure 6.1: Results for dataset 1.



Model	Accuracy	Recall	Precision	F1-Score
RF	99.92	71.61	82.84	76.82
	99.93	76.10	82.69	79.26
DT	99.90	62.58	77.60	69.29
	99.87	68.14	60.62	64.16
ANN1	99.90	55.45	81.16	65.88
ANN2	99.91	66.34	79.76	72.43
	99.91	66.37	76.53	71.09
NB	97.95	75.25	6.25	11.54
	98.65	77.87	8.59	15.47
LR	99.88	38.71	85.71	53.33
	99.89	47.78	79.41	59.66

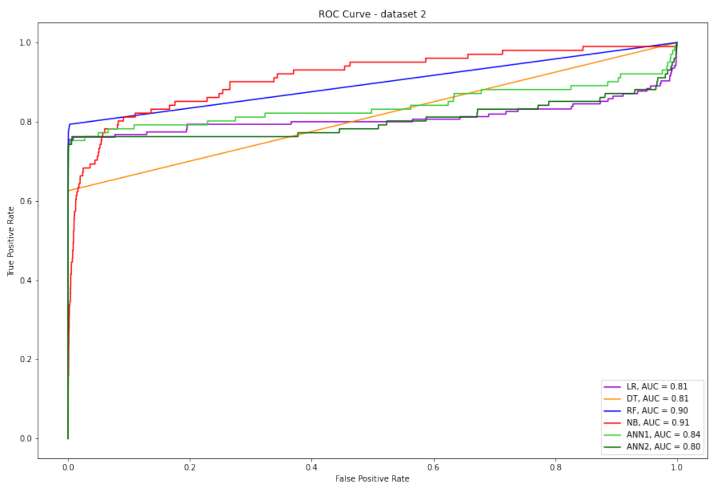
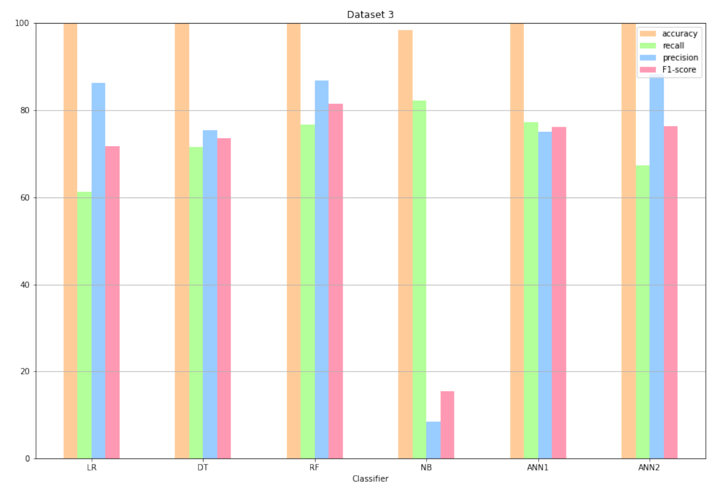


Figure 6.2: Results for dataset 2.



Model	Accuracy	Recall	Precision	F1-Score
RF	99.94	76.77	86.86	81.51
	99.94	75.22	85.85	80.18
DT	99.91	71.61	75.51	73.51
	99.90	76.10	68.80	72.26
ANN1	99.91	77.23	75.00	76.10
ANN2	99.93	67.33	88.31	76.40
	99.91	67.25	77.55	72.03
NB	98.40	82.18	8.49	15.38
	98.81	81.41	10.07	17.93
LR	99.91	61.29	86.36	71.70
	99.90	53.09	80.00	63.82

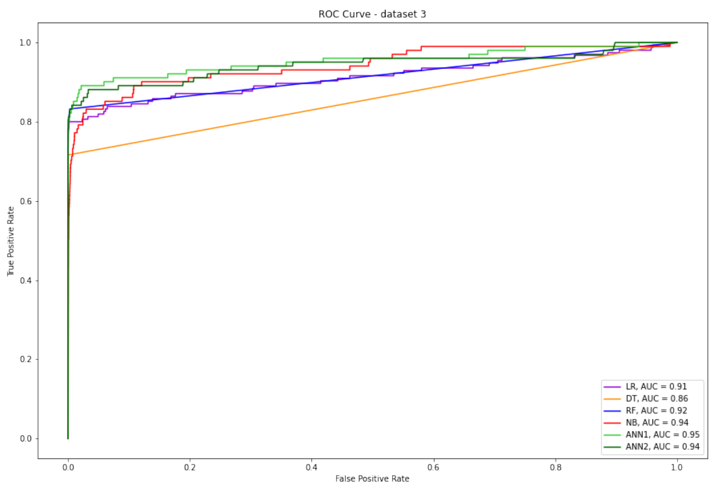
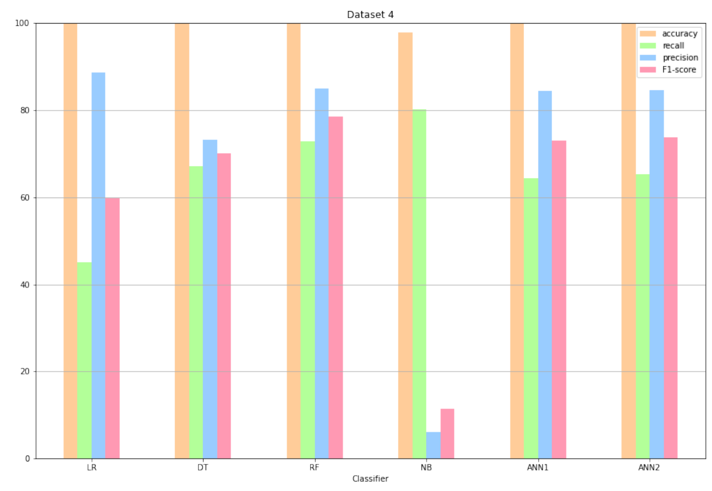


Figure 6.3: Results for dataset 3.



Model	Accuracy	Recall	Precision	F1-Score
RF	99.93	72.90	84.96	78.47
	99.94	77.87	83.80	80.73
DT	99.90	67.10	73.24	70.03
	99.91	76.10	72.26	74.13
ANN1	99.92	64.36	84.42	73.03
ANN2	99.92	65.35	84.62	73.74
	99.91	61.06	81.17	69.69
NB	97.78	80.20	6.12	11.38
	98.48	81.41	7.97	14.53
LR	99.89	45.16	88.61	59.83
	99.89	46.90	77.94	58.56

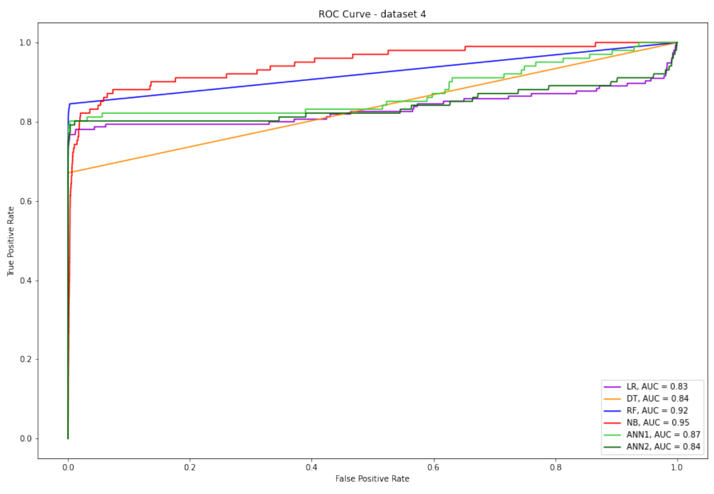
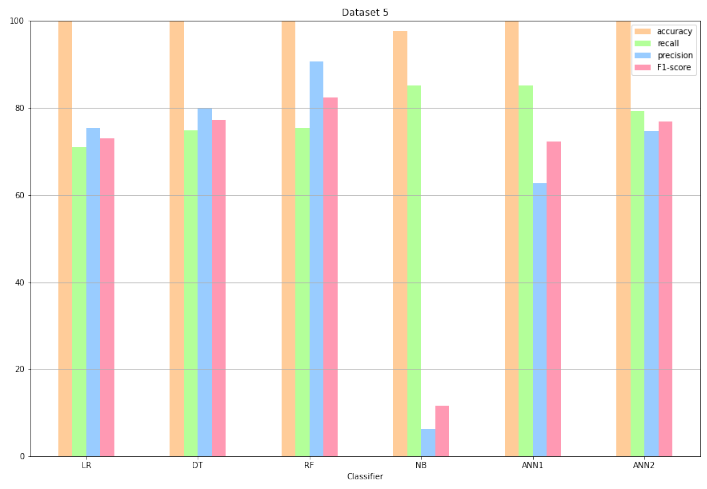


Figure 6.4: Results for dataset 4.



Model	Accuracy	Recall	Precision	F1-Score
RF	99.94	75.48	90.70	82.39
	99.98	72.56	95.34	82.41
DT	99.92	74.84	80.00	77.33
	99.89	72.56	65.07	68.61
ANN1	99.88	85.15	62.77	72.27
ANN2	99.92	79.21	74.77	76.92
	99.08	77.87	12.27	21.20
NB	97.69	85.15	6.20	11.57
	99.44	57.52	15.85	24.85
LR	99.91	70.97	75.34	73.09
	99.77	46.90	34.64	39.84

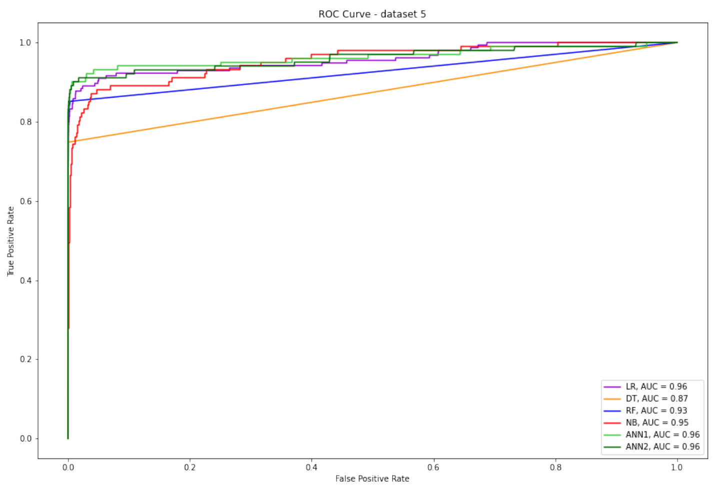
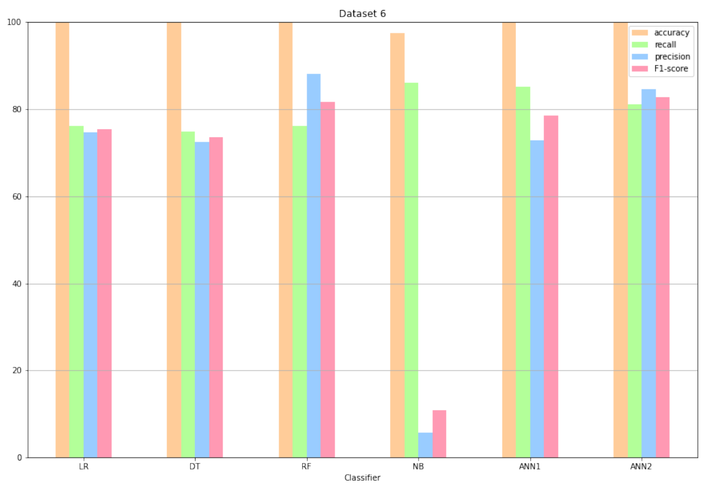


Figure 6.5: Results for dataset 5.



Model	Accuracy	Recall	Precision	F1-Score
RF	99.94	76.13	88.06	81.66
	87.95	77.87	92.63	84.61
DT	99.90	74.84	72.50	73.65
	96.91	76.10	71.07	73.50
ANN1	99.92	85.15	72.88	78.54
ANN2	99.94	81.19	84.54	82.83
	97.80	74.33	42.85	54.36
NB	97.49	86.14	5.78	10.83
	80.31	64.60	13.95	22.95
LR	99.91	76.13	74.68	75.40
	93.88	60.17	62.96	61.53

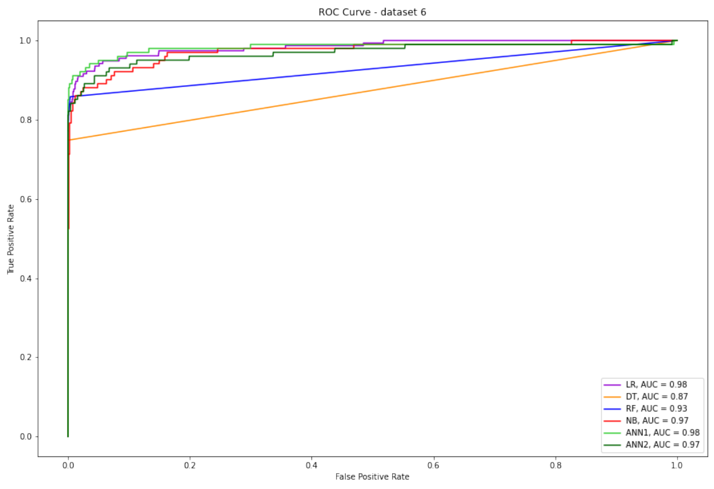
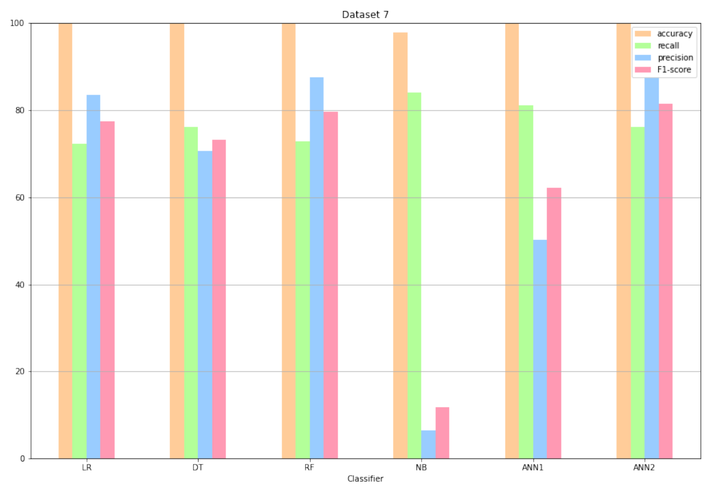


Figure 6.6: Results for dataset 6.



Model	Accuracy	Recall	Precision	F1-Score
RF	99.93	72.90	87.60	79.58
	83.78	79.64	92.78	85.71
DT	99.90	76.13	70.66	73.29
	89.91	79.64	68.70	73.77
ANN1	99.82	81.19	50.31	62.12
ANN2	99.94	76.24	87.50	81.48
	88.93	78.76	82.40	80.54
NB	97.78	84.16	6.37	11.85
	78.14	83.18	6.73	12.46
LR	99.92	72.26	83.58	77.51
	79.91	59.29	81.70	68.71

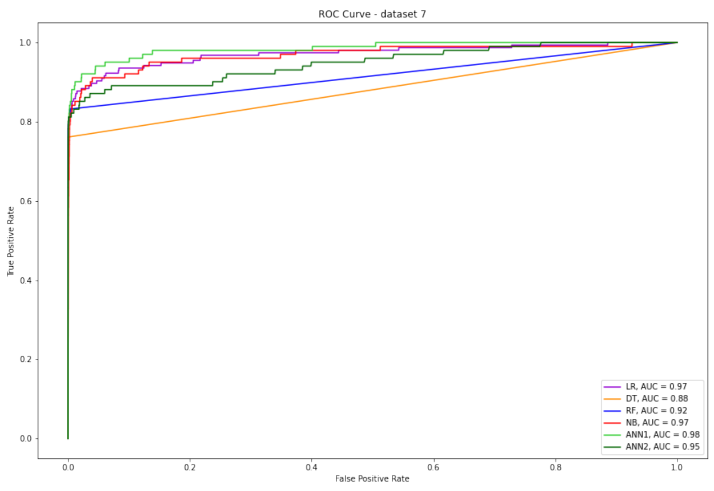


Figure 6.7: Results for dataset 7.

7 | Conclusion

In the first five datasets, which use the GA-selected features, we found good values of all the metrics, comparable with the ones pointed out by the scientific paper. In the last two datasets, namely the entire one and the one with randomly selected features, we did not notice a significant drop in accuracy like the one highlighted in the paper. We end up saying that the selection of the features using GA does not improve so much the goodness of the generated models but we agree that the methods are faster in terms of execution time since we use a smaller amount of data.

In general, the accuracy is around 99.9%, only naive Bayes models struggle to reach this threshold and have an average accuracy of 98%. Overall naive Bayes didn't perform well since we can only optimize the model for having a high recall or a high precision but not both. It's evident from the bar plots that this model has problems fitting the underlying structure of the data. Concerning the other models it is true that RF performs almost always better than DT, but it requires also more memory and training time than DT. DT seems to be a good compromise among all the datasets since it takes little time to train, has just a few hyperparameters to tune and can create simpler models. Considering the ANN, instead, it reaches good performances but it is very difficult to train since there are a lot of hyperparameters to tune and just by slightly changing a hyperparameter is possible to end up with a model with completely different performances. Hence, probably this is not the best model to face this problem. Among the two alternatives of ANN, the one with two outputs appears to be the best one since in the majority of the cases it has better values of the metrics. Considering LR, it performs quite well, is very fast to train and all hyperparameters regard the optimization method to use, but there is often a trade-off between recall and precision making it not the best model for this aim. This last is a general problem encountered in almost all models, particularly evident in ANN, in fact increasing one of the two metrics the other would decrease making it difficult to find a balance between the two.

ROC curves are similar to the ones obtained by the researchers and in the majority of the cases the associated AUC is higher than 90% meaning that the model is able to classify the data.

In these observations we have to take in mind that we got these results by using the same hyperparameters to train all the models of a specific classifier, so by tuning each model separately we could get even higher values of the metrics. Moreover, in some cases, we have decided to keep certain hyperparameters that give slightly worse performances but at the same time decrease a lot the computation time. This was done just to make the reader able to execute the Notebook in a small amount of time, approximately 12 minutes. In practical applications, there is not this time constraint and it is preferred to achieve small improvements in the metrics paying with some additional training time. We also remind that the results were obtained with implementations of the algorithms from scratch, instead, the researchers used libraries that could contain optimizations.

The researcher's focus was primarily on the accuracy, but recall is very important too. In fact, since there is a huge imbalance between the two classes, it is not difficult to obtain a high accuracy because a classifier that labels all samples as normal transactions would get an accuracy higher than 99.8%. Moreover, in the field of payment transactions, we think that it is important to have a model capable of identifying accurately when a fraud is happening, so to have a higher recall with respect to precision, since it is better to have some fake alarm about a possible fraudulent transaction than not warning about a real fraud at all.

8 | 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 2022/23**, professor Edie Miglio
- **https://towardsdatascience.com**
- **https://www.youtube.com/@AssemblyAI**
- **https://www.ibm.com/topics/decision-trees**
- **https://www.ibm.com/topics/random-forest**