Mathematics in Machine Learning Final Project

# Bank Marketing Dataset Analysis

Federico Germinario, Andrea Ghiglione

09/09/2022

# Contents

# 1 Problem Overview

## 1.1 Introduction

In this project we work with a Bank Marketing Data Set related with marketing camapaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls and more than one contact to the same client was required. In particular, it is a binary classification problem where the goal is to predict whether a client will subscribe a term deposit. The Data Set can be found here, whereas the code can be found on GitHub.

## 1.2 Data Meet & Greet

We were provided 4 datasets as comma-separated values files:

- **bank-additional-full:** it is a dataset with 41188 records, 19 features and the target variable.
- **bank-additional:** it is a randomly subset of the above dataset, in particular 10
- **bank-full:** it is a dataset with 45211 records, 16 features and the target variable.
- **bank:** it is a randomly selected subset of the above dataset, in particular 10% of the records are present.

Smaller dataset version are available for testing more computationally demanding models, such as *SVM*. However, using Google Colaboratory, computational challenges were lighter and we managed to use the *bank-full* dataset. We did not choose the *bank-additional-full* since there were 3 extra features not particularly relevant, and there were less records.

## 1.3 Software & Tools

For this project we worked with **Python**, one of the most popular programming languages and widely used in Data Science and Machine Learning fields. We exploited free computational resources on **Google Colaboratory** which also allowed us to enrich the analysis with *LaTeX, HTML* and images.

We used a set of libraries which are summarized below:

- **NumPy:** Numerical Python, library which handles n-dimensional arrays through vectorization, indexing and broadcasting concepts. Extremely user-friendly and strong computing tool.

- **Pandas:** Library useful for data analysis and manipulation of data in tabular format.
- **Matplotlib:** Library for creating animated and interactive visualizations in Python. Extremely useful for communicate results through visual perception.
- **Seaborn:** Library for data visualization based on Matplotlib. More suitable for visualizing data with certain plots.
- **Scikit-learn:** Library tailored for Machine Learning and an efficient tool for predictive data analysis.
- **SciPy:** Library for algorithms and mathematical tools for the Python language. Extremely useful for optimization, algebraic operations, statistics, etc.
- **Imblearn:** Imbalanced-learn, library which provides tools for dealing with classification for imbalanced dataset.
- **OpenCV:** Real-time optimized Computer Vision library, useful for loading images.
- **Category_encoders:** Library with a set of sklearn-like set encoders to handle categorical features with different techniques.
- **Statsmodels:** Python module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration.
- **Tqdm:** Library to show a smart progress meter on iterables.

## 1.4 Exploratory Data Analysis

The dataset we worked with is composed of 16 features; in particular there are 6 categorical attributes, 3 binary attributes and 7 numeric attributes. These attributes describe a series of conditions of the client involved in the bank telemarketing campaign. The categorical features are about the job type of the client, the marital status, the education level, the contact information, the month of last contact, and the outcome of the previous marketing campaign. The binary features described whether the client has credit in default, if he has housing loan and if he has personal loan. The numeric features provides information about client's age, yearly balance in euros, day of last contact, number of contacts during this campaign, number of days that passed by after last contact of a previous campaign and the number of contact performed before this campaign. The target variable is a binary variable which corresponds to the fact whether the client subscribed a term deposit or not. All these information are reported with more details in Table 1

| Feature | Description | Type |
|---|---|---|
| Age | Age of the client; it is greater or equal than 18 since the dataset is referred to adult people. | Numeric |
| Job | Job type, it consists of a job category among 12 professional figures, including the unknown class. | Categorical |
| Marital | It refers to the marital status of the person and it can be single, married or divorced. | Categorical |
| Education | It provides information about the education level of the subject. It can be primary, secondary, tertiary or unknown if no information is present. | Categorical |
| Default | Binary feature about the credit in default situation of the client. It can be either yes or no. | Binary |
| Balance | Client yearly balance in euros. | Numeric |
| Housing | If the client has an housing loan. | Binary |
| Loan | If the client has a personal loan | Binary |
| Contact | Contact information's type, it can be cellular, telephone or unknown. | Categorical |
| Day | Last contact day of the month. | Numeric |
| Month | Last contact month of the year; it is a string representing the month. | Categorical |
| Duration | Last contact duration in seconds. | Numeric |
| Campaign | Number of contacts performed during this campaign and for this client, including the last contact. | Numeric |
| P_days | Number of days that passed by after the client was last contacted from a previous campaign (-1 means no previous contact) | Numeric |
| Previous | Number of contacts performed before this campaign and for this client | Numeric |
| P_outcome | Outcome of the previous marketing campaign, it can be success, failure, other or unknown. | Categorical |
| **y** | **Target variable; if the client has subscribed a term deposit.** | **Binary** |

Table 1: Dataset's features and target variable description

### 1.4.1 Categorical and Binary Features

Through data visualization we can better understand the relation between each feature and the target variable. For categorical and binary features we chose the stacked bar plot because they encode in a compact and clear way the relation between each feature's category and the amount of clients who subscribed a long term deposit. Some of these visualization are reported in Figure 1 ; all other plots can be found on the official code.

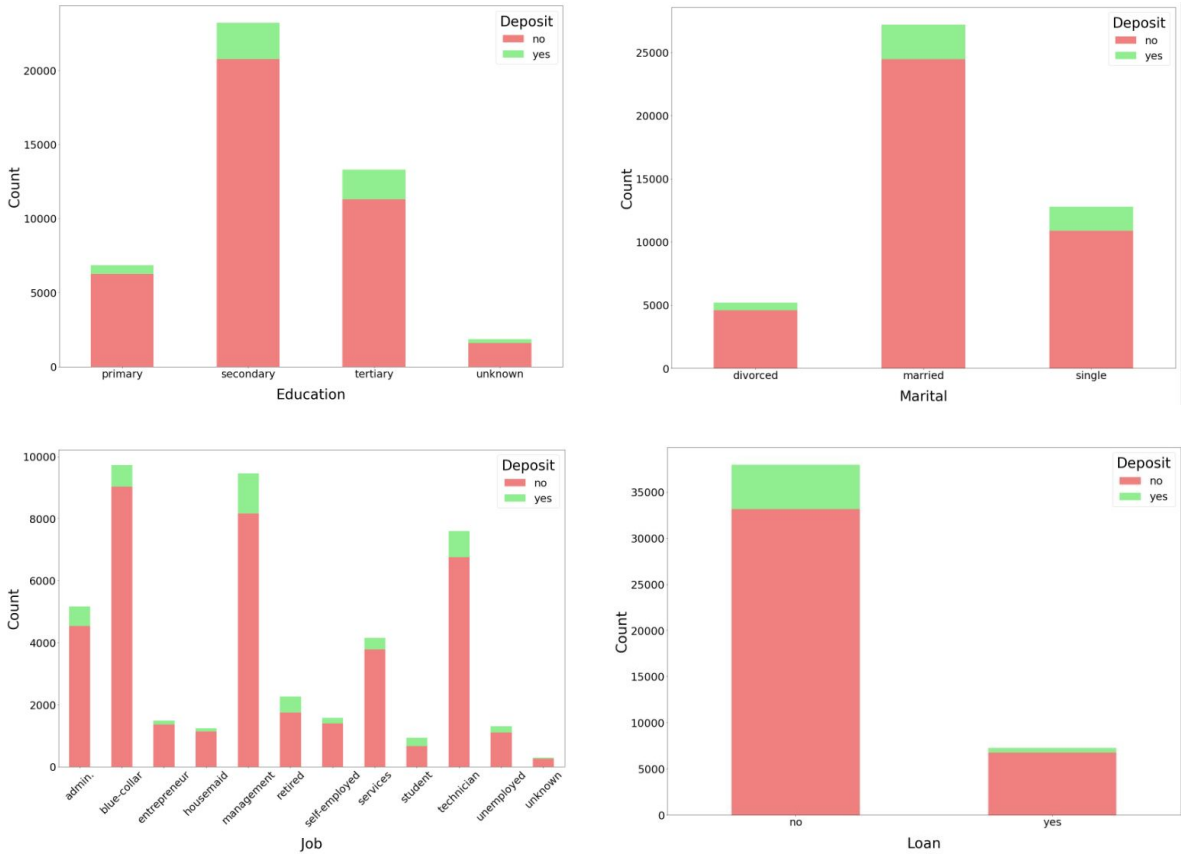


Figure 1: Subset of categorical and binary features and their relation with the target variable

From these plots we can see that the percentage of people who subscribed a long term deposit is generally low, but we could expect that since it is the result of bank telemarketing which will capture a limited amount of the people called. In particular, analyzing the **job** we can see that the majority of people who subscribed are students, retired and unemployed people. The **marital** status instead shows that single people are the most likely to subscribe a deposit. Looking at **education** tertiary and unknown are on the top, however this is not ideal since we understand that people with a degree subscribed a lot, but also people who we don't know about their education. The people

with credit in **default** almost double people without it. Furthermore, both people without **housing** loan and general **loan** subscribed much more than people with them. Looking at the **contact** it does not make a lot of difference if people gave a cellular or a telephone because who did that subscribed much more with respect to people who this information is unknown (probably not provided). The last contact **month** of the year shows that people contacted in March are the one who subscribed the most; this is interesting since other top positions are about winter months for the most. Finally, the last contact **day** of the month does not carry a lot of information.

### 1.4.2 Numeric Features

In order to analyze numeric features we used the violin plots since they provide density information for a range of real values that it is extremely easy to see and to compare. In addition, we plot the information about the quartiles as dashed lines. The plots are reported in Figure 2

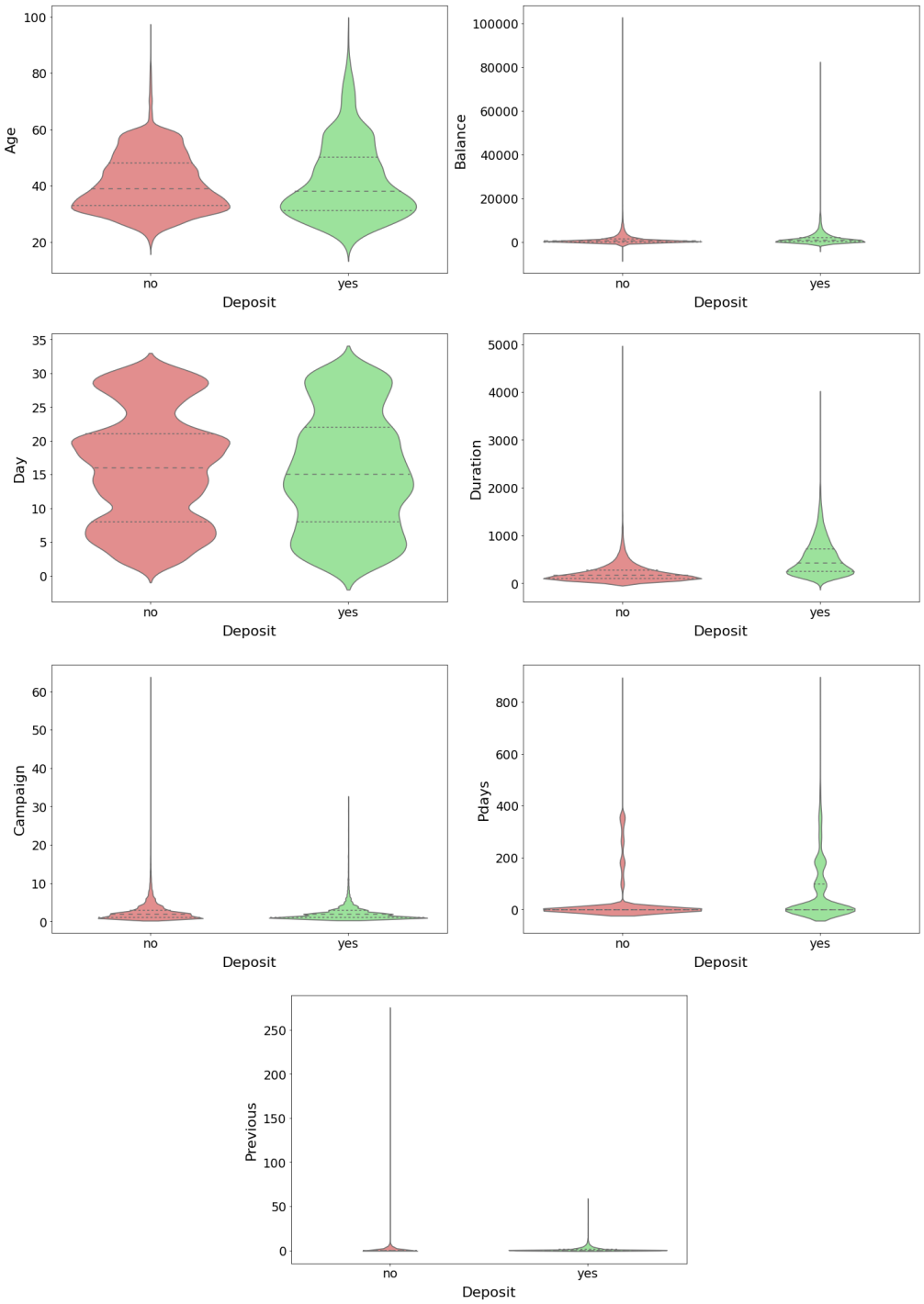


Figure 2: Violin plots of numeric features with respect to the target variable.

# 2 Data Preprocessing

## 2.1 Target variable's mapping

As a very first step, we mapped the target variable from the *yes/no* domain to *1/0* domain, hence making it an integer value for the sake of readability.

## 2.2 Dataset split

Then, we created a training set and a test set by splitting the original dataset into two portions. The test set consists of 20% of original records, while the training set is the other 80%. We made sure to have a good split by setting the *stratify* parameter based on the target variable; doing that we ensured to have two sets with the same percentage of records belonging to either class. In addition, we also set a *seed* to work with a repeatable pipeline which allowed us to compare the results in a meaningful way.

## 2.3 Missing values inspection

A key aspect to consider when doing data preprocessing is to handle *missing values*. They could be there for many reasons, such as incomplete clients' answers or as a consequence of the data gathering process. Missing values are generally handled with techniques such as *backward fill, forward fill, replacement with a metric, replacement with random values* or simply *dropping* records with some missing attributes. However, the dataset we are working on has no missing values; it has some *unknown* values which we considered as a special class. This condition is present only in the *job*, *education* and *p_outcome* for specific reasons, so it would be a mistake to consider this class as *missing values*.

## 2.4 Duplicates inspection

We analyzed the duplicates in the training set since a record present more than one time, in this problem, would cause our model to think that such record has a bigger relevance. However we can clearly see that a duplicated record in a bank telemarketing dataset is only due to a mistake in data collection, hence we should remove it to build a model that will be able to model the data in a more general way. After the inspection we noticed that *no* duplicates are present.

## 2.5 Outliers

We perform an analysis on outliers since they are data points which are likely to be resulting from a measurement error and they could be misleading for the classification

model we will build. We can perform outliers detection in different ways, the ones we implemented are:

- **Z-scores:** Removing data points which z-score is either too low or too high. In particular, the z-score of a data point is computed by taking the raw data point, subtracting the mean of the data and, finally, dividing by the standard deviation of the data. We set z=[-5,5] as valid range of z-score. We compute the z-score on each numerical feature per data point, and if one of them exceed the range then we consider the whole data point as outlier.

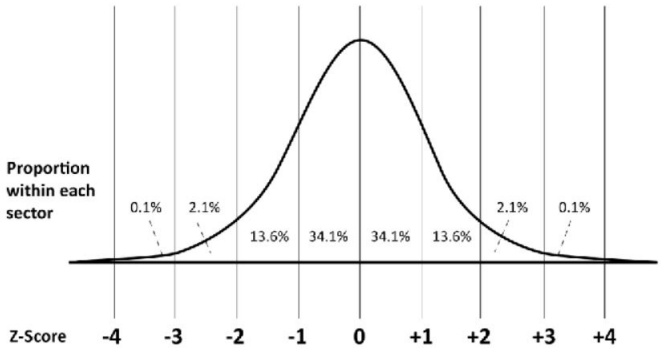


Figure 3: Proportion within each sector and respective z-scores

- **Percentiles:** Removing data points which don't lay between a range of two percentiles. In our default setting we considered non-outliers the data point which laid between the 0.5-percentile and 99.5-percentile. We computed the percentiles for every feature and we labeled a data point as outlier if one of its features' values did not fit into the feature's percentiles range.

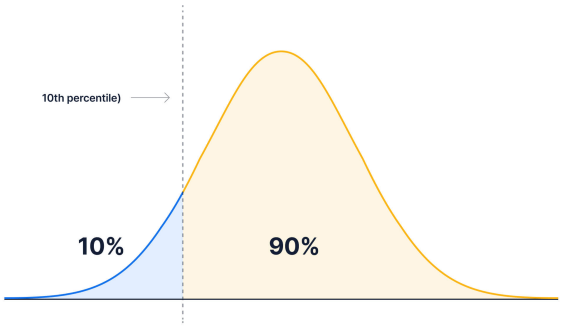


Figure 4: Example of percentile (10-percentile)

We selected the thresholds by making sure that only a small amount of records are deleted, and we payed attention to the ratio of the target variable classes that we removed. In particular, the original percentage of the positive class was 13%, while this ratio in the removed records is 27%. The outliers removal method we chose is z-scores which let us remove 856 records which is the 2% of the training set.

## 2.6 Correlation

We can now look at how the numerical features are correlated. Correlation is a statistical measure that let us grasps how much two features are related to each other through a linear relation. In order to do so, we can plot the correlation heatmap of the features. We can see that the highest correlation in absolute value among two different features


Figure 5: Heatmap of numerical features' correlation

(*pdays* and *previous*) is 0.64, which is not high enough to drop one of these features. The correlation type reported is obtained through the *Pearson* coefficient reported in Equation 1.

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \tag{1}$$

## 2.7 Scaling

We apply scaling on the numerical features since we want all the features to be in the same range and magnitude. Also, in next steps we will apply *PCA* which is an algorithm that is skewed towards high magnitude features, so we mitigate the problem by means of scaling. There are several techniques to do so, such as:

- **Standard Scaler:** This standardization consists in replacing each feature's value with its Z-score, hence subtracting the feature's mean and then dividing by the feature's standard deviation.

- **MinMax Scaler:** It consists in subtracting the feature's minimum value and then dividing by the difference between the feature's maximum value and minimum value as reported in Equation 2. The final value will be in the range [0,1]

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2}$$

## 2.8 Encoding

Since we have also categorical features, we need to encode them in order to apply some classification models. In literature, there are very many ways to encode categorical features, such as:

- **Ordinal Encoding** encoding technique that assigns to each categorical variable a unique label based on alphabetical ordering. This is the most naive encoding of categorical features and it is not very effective since it introduces an order based on the alphabet and this is something that is often wrong. For example if we have two strings, *Italy* and *USA* we would get an encoding of 0 and 1 respectively, which would introduce a non-existing relation of *USA is bigger than Italy*.

- **One Hot Encoding:** encoding technique that overcomes the problem we analyzed in label encoding by creating additional features based on the number of unique values in the categorical feature. Each of this feature's values are binary values and so 0 and 1; this is where the name one hot encoding came from. When using one hot encoding we have to pay attention to the problem called *curse of dimensionality*; this encoding may introduce a very high number of features, making our dataset sparse.

- **Count Encoding:** encoding technique that encodes each categorical feature's values with its frequency along the dataset. An encoding order is introduced as for the ordinal encoding, however here the order is carrying true information since it is the outcome of a count operation, encoding the frequency of a feature's value.

- **Target Encoding:** encoding technique that encodes each categorical feature's values with a measurement of the effect they may have on the target variable. For a binary classifier, from a mathematical point of view, it consists in calculating the conditional probability of the target to be either 0/1 given the specific categorical feature's value. We can compute different metric on the target, however the most widely used is the mean and that's why this technique is often called *mean encoding*. However, this encoding has the problem of *overfitting* and *target leakage*.

## 2.9 PCA

In machine learning the term *curse of dimensionality* refers to the problem in the context of high dimensional data, of points tending to be isolated from each other, inducing *sparsity*. In order to prevent and mitigate this problems, we can make use of algorithms of *dimensionality reduction*, whose goal is to create a mapping from the current high dimensional space to a lower dimensional one. The curse of dimensionality itself does not depend on the model chosen, but it's an intrinsic problem related to the space of the points. One, widely used, unsupervised method for *dimensionality reduction* is the *Principal Component Analysis (PCA)*, which works by building relevant features through combinations of the original features. The constructed features are built by projecting the data points onto few of the principal components. The principal components are formed as combination of the original features that explains most of the variance. We can take as many principal components depending on the amount of variance we want to be explained, this works well in case of highly correlated features. In order to do so, we take the eigenvectors of the covariance matrix that has the highest value of the corresponding eigenvalues and then considering, for example, the second principal components by taking it orthogonal with respect to the first component and that best explain the variance of the remaining subspace. We are going to repeat the process for the next components. We can define a parameter for the percentage of the variance explained. In fact we can take a number $k$ of eigenvectors, so that the sum of the corresponding $n$ eigenvalues is equal to the percentage of variance we want to be explained (i.e. 90%).

So for a set of observed $d$-dimensional data vectors $\boldsymbol{Z} = \{\boldsymbol{z}_n\}_{n=1}^N$, the $q$ principal axes $\boldsymbol{w}_j$, for $j = 1, ..., Q$ are those orthonormal axes onto which the retained variance under projection is maximal. It can be shown that the vectors $\boldsymbol{w}_j$ are given by the $q$ dominant eigenvectors (i.e. those with the largest associated eigenvalues $\lambda_j$) of the sample covariance matrix $\boldsymbol{S} = \boldsymbol{Z}^\top \boldsymbol{Z}$, such that $\boldsymbol{S}\boldsymbol{w}_j = \lambda_j \boldsymbol{w}_j$. The $q$ principal components of the observed vector $\boldsymbol{x}_n$ are given by the (latent) vector $\boldsymbol{y}_n = \boldsymbol{W}\boldsymbol{z}_n$, where $\boldsymbol{W} = [\boldsymbol{w}_1, ..., \boldsymbol{w}_q]^\top$.

The steps to perform the PCA are:

- Center the data $Z = X - \bar{X}$

- Compute the covariance matrix $S = Z^T Z$
- Compute the eigenvectors and eigenvalues
- Take the K eigenvectors with the highest value of the corresponding eigenvalues: $W = [e_1, e_2, ..., e_k]$
- Project the points on the new space: $y = ZW$

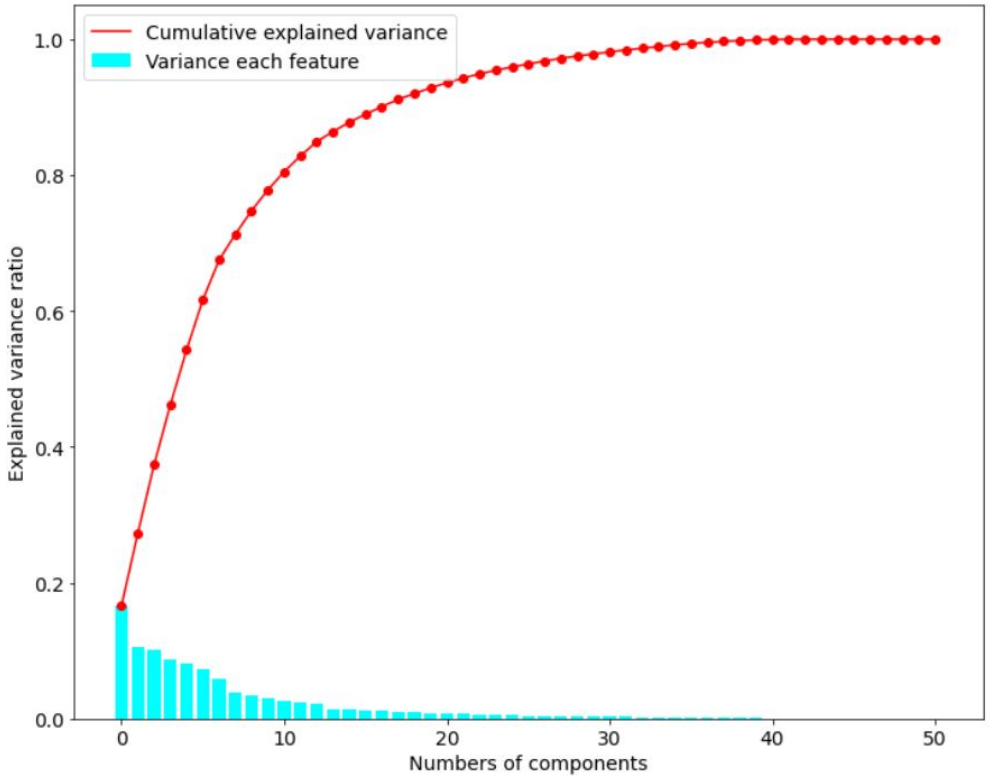


Figure 6: Explained Variance Ratio.

We can see that the number of components which are explaining a big part of the total variance are not all the components; hence we can drop some of them. In order to do that we could either specify a number of components we want to keep, or to specify the explained amount of variance we want. We chose the latter method with 95% of variance that corresponds to roughly half of the original components. The number of components considered are 24.

# 3 Classification

In this chapter we are going to work on the heart of the project, which is the classification task, first we are going to apply feature selection, through backward elimination and tackle the problem of the imbalance of the dataset and then we are going to see different Machine Learning models applied to our classification task. Most of the ML models comes from the sklearn package, but others such as the Metropolis-Hastings are built from scratch. First of all, as we previously mentioned, we applied a first split of the dataset in train and test. The split is a stratified one, in that way the resulting subsets are representative of the overall dataset. The split is 80/20 meaning 80% of the dataset is used for train while the remaining 20% is kept for testing, in order to have a good indicator for the generalization error.

## 3.1 Feature selection

We apply feature selection in order to deal with a lower dimensional feature space. By removing the redundant features present, in fact, we can reduce the high dimensionality eliminating features which do not carry any further valuable information. We will use backward elimination and logistic regression for this purpose. We are going to make use of the package *Statsmodels* for the logistic regression and we are going to implement from scratch the backward elimination script. With the logistic regression we can fit a model and obtain parameters for the different features obtaining a significance value used to understand if the values taken into considerations are significant for the classification task. Using the logistic regression as model to fit, we can apply backward elimination by following the steps:

1. Select all features to use to build the model
2. Select a significance level $\beta = 0.05$, above which the features are removed
3. Fit the logistic regression model over the selected features
4. Compute the p-values for all the features
5. Pick the maximum p-value
6. Compare it with $\beta$
    - if $p - value > \beta$ remove this feature from selected features and go to step 3
    - else keep the selected features and finish

After having performed backward elimination we are going to have only 30 features. In particular we removed the features: *month_apr, job_management, loan_no, educational_tertiary, housing_no, job_services, default_no, poutcome_other, default_yes, job_technician,*

*contact_cellular, job_unemployed, age, job_unknown, pdays, poutcome, contact_telephone, month_feb, job_blue_collar, job_self_employed, job_entrepreneur.*

We also applied PCA as a feature selection technique as we stated in the previous section, by applying PCA, instead we reached 24 generated features. We are going to evaluate our models by using the feature space generated by the PCA, a feature space without the removed features computed through the backward elimination and the original feature space. Interesting to note, that by applying PCA, we are going to lose much of the interpretability of the feature space.

## 3.2 Imbalance

As we have seen the dataset has turned to be imbalanced, having a great disproportion for the two outcome classes "subscribed term deposit (yes)" and "not subscribed term deposit (no)". In fact we have that only 13% of people has subscribed a term deposit. The problem of fitting a model with this imbalance is that we could affect the generalization, in fact in this context a dummy model classifying all instances to be *"no"* will result in holding 87% of accuracy, but effectively holding poor performance on classifying the *"yes"* class. In order to tackle this problem and reduce the impact of the class imbalance, we make use of oversampling techniques. These techniques aim at introducing (sampling) new artificial instances coming from the minority class in order to arrive to a balance between the two classes. We are going to compare two different oversampling techniques, both of which come from the *imblearn* package:

- **Random Oversampling:** this is a Naive oversampling technique, which consists in randomly sampling from the minority class, by taking with replacement instances and adding duplicates of them to the training set until the two classes are balanced.

- **Synthetic Minority Over-sampling TEchnique (SMOTE):** this is an oversampling technique which generates synthetic instances for quantitative features in the dataset. SMOTE works by looking at the k nearest neighbors of a randomly chosen instance $x_i$ belonging to the minority class. Then a random number $\alpha$ is sampled from the range (0, 1] that is used to produce the continuous features through the following equation in which $x_z$ is randomly chosen among the k neighbors.

$$x_{new} = x_i + \alpha(x_z - x_i) \tag{3}$$

The random oversampling technique could lead more easily to overfit, due to the fact that we are feeding the model with duplicated values, but the process is lightweight because it consists of a normal sampling. Differently SMOTE provides most reliable samples generated from others, but not duplicates but with more computational intensive process. In the following parts regarding the classification, we are going to

work on the original imbalanced dataset and with these two rebalanced datasets.

## 3.3 Pipeline

Before starting the training and test phase and the evaluation of the models, we would like to show a representation of the entire pipeline. We are going to see the different steps of which the overall pipeline is composed in Figure 7.

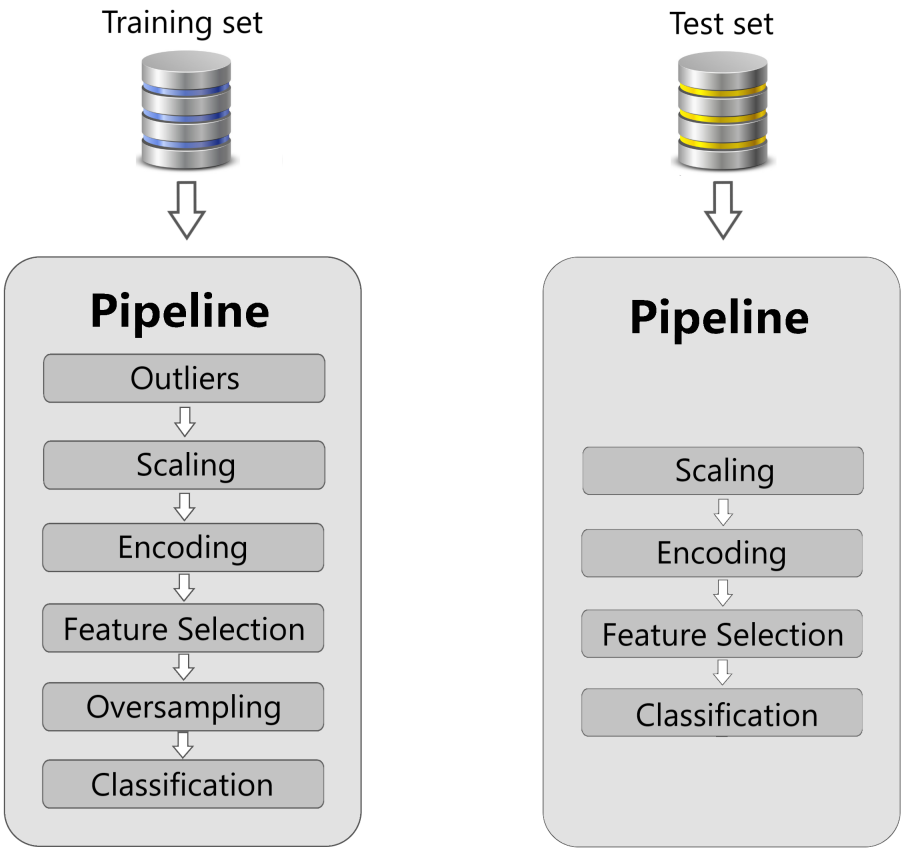


Figure 7: Training and test data is fed to this pipeline

In the path on the left we can see the steps for the training set. First we remove the outliers and scale, by fitting and transforming the numerical features of the dataset and encoding the categorial features. Then, feature selection is applied by computing the features that we want to keep and eliminate the least relevant one. For the

feature selection three different subsets are defined, using PCA, backward elimination and keeping all the original features. Afterwards we are going to apply oversampling techniques by using random oversampling and SMOTE. We are also going to use the original imbalanced dataset. At the end of these stages a classifier is fitted over the training set to get a predictive model. In the path on the right, instead, we are going to apply the scaling and encoding (transform) using the mean and variance computed with the training set and the features encoded in the training set, Then the features discarded for the training set are removed and then the test set is ready for the evaluation with the previously fitted model.

## 3.4 Metrics of evaluation

We want to give a briefly highlight of the metrics of evaluation we are going to use to asses the performance of our models. In our context, tackling a binary classification problem, we can make use of the *Confusion matrix*, which is going to provide a fair visualization of the results of the different models used. The confusion matrix consists in fact of instances correctly predicted as no deposit (no) or deposit (yes), respectively *true negatives (TN)* and *true positives (TP)*, and incorrectly ones, classified by the model as deposit *false positives (FP)* and as non deposit *false negatives (FN)*. In figure 8 is present a representation of the confusion matrix.

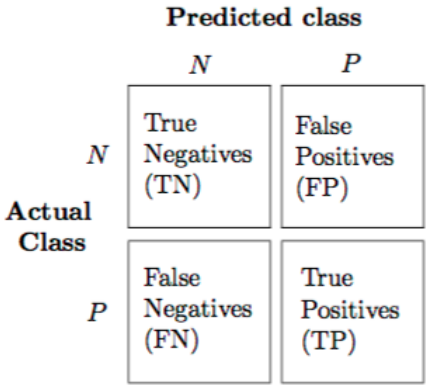


Figure 8: Representation of the confusion matrix

The confusion matrix can be used to easily compute some important metrics:

- $accuracy = \frac{TP+TN}{TP+TN+FN+FP}$
- $precision = \frac{TP}{TP+FP}$
- $recall = \frac{TP}{TP+FN}$ also called *true positive rate* or *sensitivity*
- $specificity = \frac{TN}{TN+FP}$
- $false\ positive\ rate = \frac{FP}{FP+TN}$
- $f1 - score = 2\frac{precision*recall}{precision+recall}$

The *accuracy* represents the number of correct predictions over the total number of predictions. In our case, this measure could lead to a misleading interpretation, in fact, in case of an imbalance dataset, a *dummy model* predicting always the same class will hold an high accuracy, in our case a model classifying only "no deposit" will achieve an accuracy of *87%*.

Precision, instead, quantifies the number of positive class predictions that actually belong to the positive class, while recall quantifies the number of positive class predictions made out of all positive examples in the dataset.

By computing the True Positive Rate and False Positive Rate, we can also create the *Receiving Operating Curve (ROC)*, which can give us a visual interpretation of the performance of the classifier, based on different values of the threshold. The ROC curve let us compute another useful metric of evaluation, by computing the *Area Under the Curve (AUC)*.

We are going to look more on the f1-score measure for the evaluation of the models

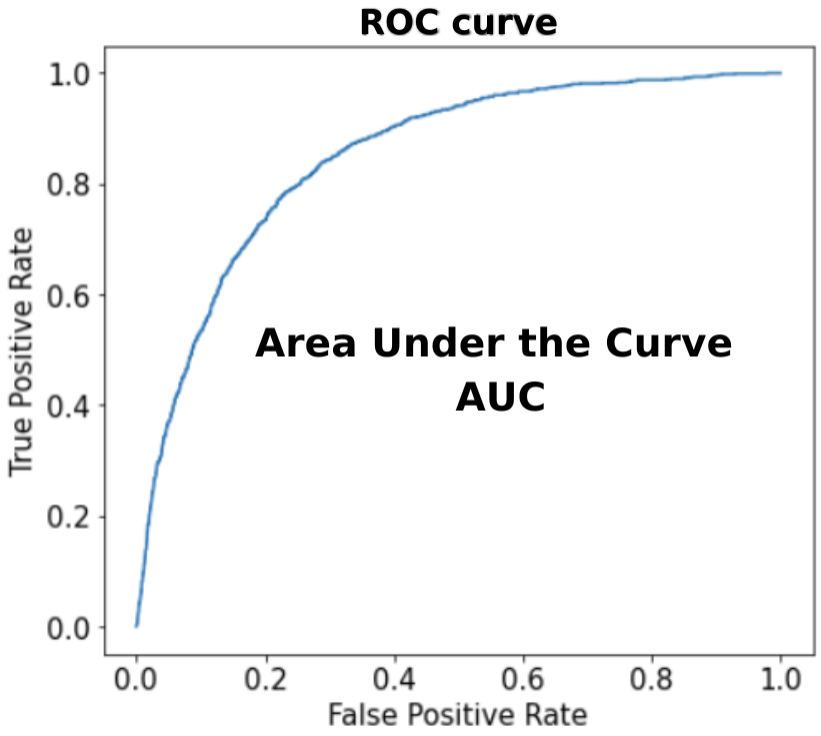


Figure 9: Example of Receiving Operating Curve and Area Under the Curve

## 3.5 Model selection

As we are going to see in the next sections, each classifier is defined by some parameters, called *hyperparameters* which control the learning process. The choice of hyperparameters called model selection would not be done extensively on all the possible combinations of the parameters due to the computational infeasibility. Model selection could not be performed over the test set, because it will affect the generalizability of the model by adding knowledge coming from the test set and could not be performed during the training, because it won't give a correct estimation of the model performance. For such reason we are going to define the *validation set*, separated both from the training and test set and for which we can evaluate the model for the different set of hyperparameters and then use the final test set to asses the overall performance over the model with the chosen parameters.

Rather than defining another set using for example a proportion of 60% training, 20% validation, 20% test, we apply *k-folds cross-validation (CV)*, which consists of splitting the the training set into k different folds, perform the training on *k - 1 folds* and evaluating the performance on the remaining fold. The training is performed k times and the average is taken to evaluate the performance over the set of hyperparameters, for all the parameters presented for each classifier on the corresponding table. The K-folds CV is implemented through the *GridsearchCV* class in the *sklearn* library. GridsearchCV let us define the set of hyperparameters, through the *param_grid* parameter and the *cv* parameter to define the number of folds to use, which in our case will correspond at the default value of 5. At the end if the *refit* parameter is set to True, the estimator with the best set of hyperparameters is retrained over the entire training set.


Figure 10: Iterations of k-fold Cross Validation

## 3.6 Classification tree

The first model we are going to test is the classification tree. Classification trees are a tree supervised learning model in which the target can only take discrete values. In these models, the features space is partitioned in a set of subregions defined through subsequent splits. At the end of the training the leaves will represent the final classes, while each node above represents a binary partition of a chosen feature. Decision Trees in general could both be used for classification and regression tasks, the procedure that a classification tree follows is to recursively split with a binary partition the set at each node level into subsets by following splitting rules on the features. Starting from the root node, the split is applied recursively until a stopping condition is met, this condition could be that all nodes contain elements belonging to the same class, or the maximum height of the tree imposed is reached or no improvements is achieved.

The algorithm for growing the classification tree need to find the splitting variables and the splitting points, that would be computationally infeasible to take into considerations all the possible combinations. For this reason the choice of parameters for the split depends on a greedy algorithm. For each node data $Q$ we need to find the splitting variable $j$ and threshold t that minimizes the overall impurity.

$$Q^- = \{X | X_j \leq t\}$$

$$Q^+ = \{X | X_j > t\}$$

In order to find the best set of parameters $(j, t)$ for the split, we consider an impurity function $H$ and try to minimize the following function with respect to the splitting parameters:

$$\underset{(j,t)}{\operatorname{argmin}}(\frac{n^+}{n} H\left(Q^+(j,t)\right) + \frac{n^-}{n} H\left(Q^-(j,t)\right))$$

With $n$ corresponding to the number of total observations and $n^+$ and $n^-$ the number of observations of the corresponding region. In order to evaluate the goodness of split,as we said, we make use of a measure of node impurity $H$. In particular as we are going to see in table, the two measures of node impurity we used are:

- GINI index:
$$H(Q) = \sum_k p_k (1 - p_k) \tag{4}$$
- Cross entropy:
$$H(Q) = -\sum_k p_k (1 - log_{p_k}) \tag{5}$$

k indicates the classes taken into consideration. Once the best local split is found, we split the data and proceed to apply the same algorithm to the successive partitioned

regions. Apart from the splitting criterion, other parameters could be defined, such as the max depth of the tree which indicates how much you accept to grow your tree in depth. The size of the tree is a parameter to be chosen based on the input data. However, if the depth of the tree is too high, that could lead to overfitting, because we are creating an over complex tree which is not able to generalize on unseen data. In order to prevent the tree from overfitting, we could set the max depth, but also the minimum number of samples to create a split on a node or we can grow the full tree and the apply *cost-complexity pruning*. We are going to use pruning and define a complexity parameter $\alpha \geq 0$ and a cost complexity measure for a single node:

$$R_\alpha(t) = R(t) + \alpha|T|$$

$\alpha$ is used to define the trade-off between the goodness of fit and the size of the tree. We are going to tune it through the cross validation together with the splitting criterion and the max features parameter which consists in the number of features to consider when looking for the best split. All the hyperparameters and the corresponding values considered during the tuning are reported in Table 13.

| Parameter | Values |
|---|---|
| Criterion | ['Gini', 'Entropy'] |
| max features | ['log2', 'sqrt'] |
| $\alpha$ | [0.005, 0.01, 0.05, 0.1] |

Figure 11: Parameters for model selection of classification tree

The 5-fold cross validation is performed over the different model's parameters and for the vanilla dataset and the oversampled version. The classification tree grants us the opportunity to easily inspect how the instances are classified and the splits performed at each node. This makes the model more interpretable and we can inspect its structure 12. Results are reported in Figure 23. Best parameters were $criterion = entropy$ , $max_features = sqrt$ and $\alpha = 0.05$

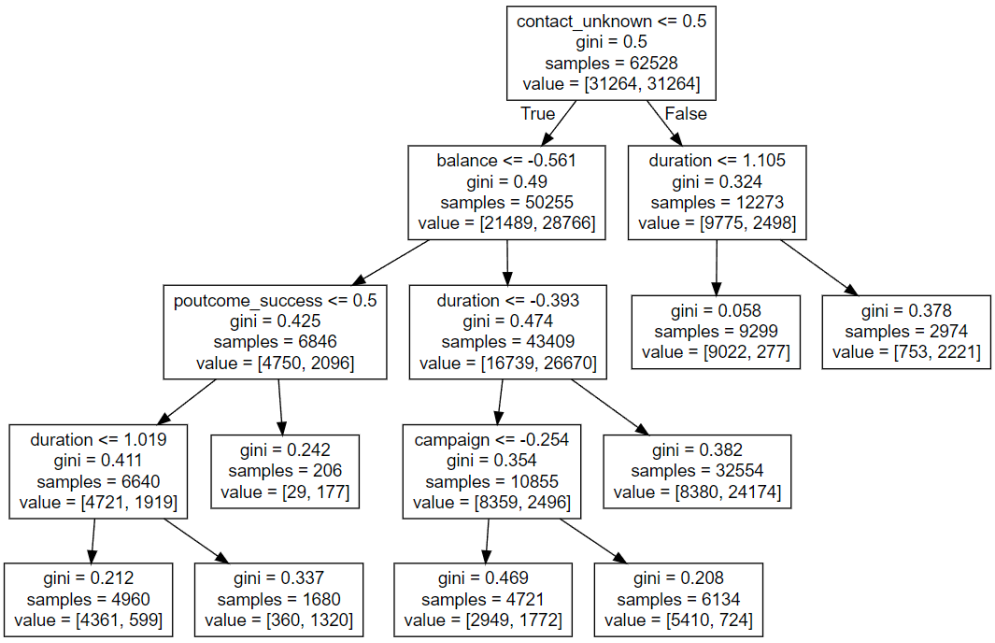


Figure 12: Tree structure of a grown tree on the dataset applying backward elimination

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.74 | 0.76 | 0.89 |
| **Precision** | 0.28 | 0.29 | 0.66 |
| **Recall** | 0.78 | 0.72 | 0.18 |
| **F1-score** | 0.41 | 0.41 | 0.28 |
| **AUC** | 0.76 | 0.74 | 0.58 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.66 | 0.64 | 0.88 |
| **Precision** | 0.22 | 0.84 | 0.57 |
| **Recall** | 0.78 | 0.84 | 0.19 |
| **F1-score** | 0.35 | 0.35 | 0.29 |
| **AUC** | 0.71 | 0.73 | 0.50 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.81 | 0.81 | 0.89 |
| **Precision** | 0.35 | 0.35 | 0.66 |
| **Recall** | 0.68 | 0.71 | 0.16 |
| **F1-score** | 0.46 | 0.46 | 0.26 |
| **AUC** | 0.76 | 0.76 | 0.57 |

Figure 13: Results obtained applying Decision Tree. From top to bottom and left to right: backward, PCA and no feature selection.

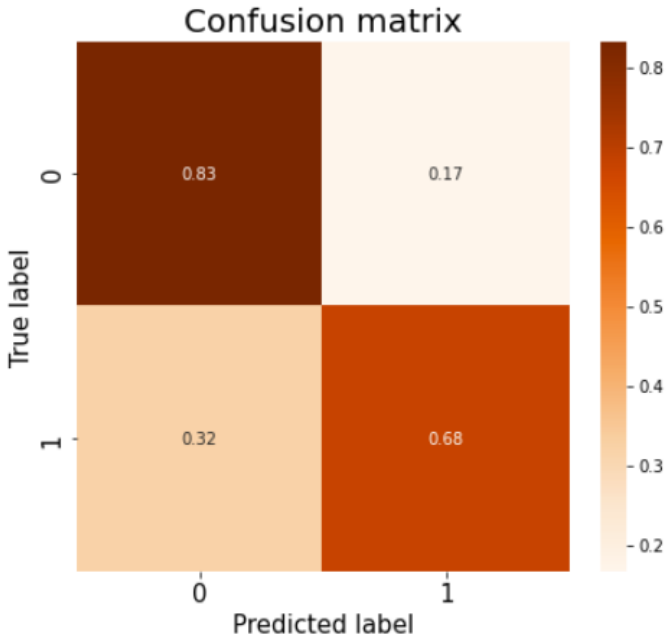


Figure 14: Confusion matrix of the Decision Tree classifier

## 3.7 Random forests

*Random forests* belong to the family of ensemble methods, whose goal is to combine different base models and aggregate their own predictions to improve generalizability of the final prediction. In particular random forests are part of the family of ensemble methods called *bagging (bootstrap aggregating)*. The aim of bagging is to build several models by fitting them on a random subset of the original training set which are selected with replacement and aggregate them by averaging their predictions. The final prediction will be formed by combining the local predictions of all the different base models through majority voting for classification or averaging for regression.

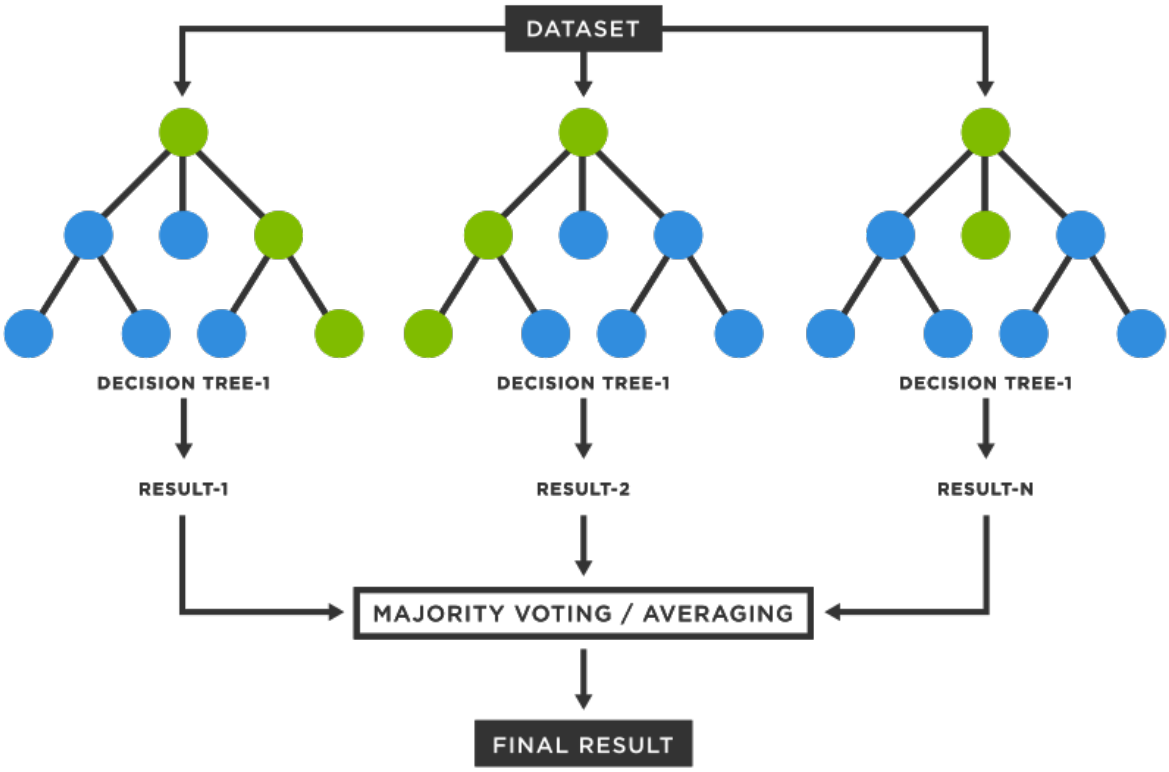


Figure 15: Representation of the random forest as ensemble of decision trees

A strong advantage of bagging consists in the capacity of reducing the overall variance of a model. This means that such methods work well with models that are able to capture complex functions and have relatively low bias such as fully developed decision trees. Random forest are built on the idea to further improve the reduction of variance,

already partly achieved by the use of bootstrap samples of bagging, by randomly select the input features for each tree.

Our parameter *max features (f)* should be $f \leq d$ with d representing the total number of features. Usually we can assign to $f$ $sqrt(d)$ or $log(d)$, we are going to tune this parameter over these two settings as reported in table 16. Of course lower the value of *f*, lower will also be the correlation among the different grown trees of the random forest.

The other hyperparameters we are going to tune are the number of estimators, meaning the number of trees composing the random forest and also, similarly to the classification trees, the criterion for each split of the tree. All the hyperparameters used for the tuning are present in Table 16. Results are reported in Figure 17. Best parameters were $n_estimators = 500$, $max\_features = sqrt$ and $criterion = entropy$.

| Parameter | Values |
|---|---|
| Criterion | ['Gini', 'Entropy'] |
| max features | ['log2', 'sqrt'] |
| N° estimators | [100, 200, 500] |

Figure 16: Parameters for model selection of random forest

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.89 | 0.90 | 0.90 |
| **Precision** | 0.54 | 0.59 | 0.65 |
| **Recall** | 0.64 | 0.53 | 0.39 |
| **F1-score** | 0.59 | 0.56 | 0.49 |
| **AUC** | 0.78 | 0.74 | 0.68 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.89 | 0.90 | 0.90 |
| **Precision** | 0.51 | 0.60 | 0.67 |
| **Recall** | 0.67 | 0.46 | 0.31 |
| **F1-score** | 0.58 | 0.52 | 0.43 |
| **AUC** | 0.79 | 0.71 | 0.65 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.90 | 0.90 | 0.91 |
| **Precision** | 0.57 | 0.61 | 0.68 |
| **Recall** | 0.62 | 0.48 | 0.39 |
| **F1-score** | 0.59 | 0.54 | 0.50 |
| **AUC** | 0.78 | 0.72 | 0.68 |

Figure 17: Results obtained applying Random Forest. From top to bottom and left to right: backward, PCA and no feature selection.

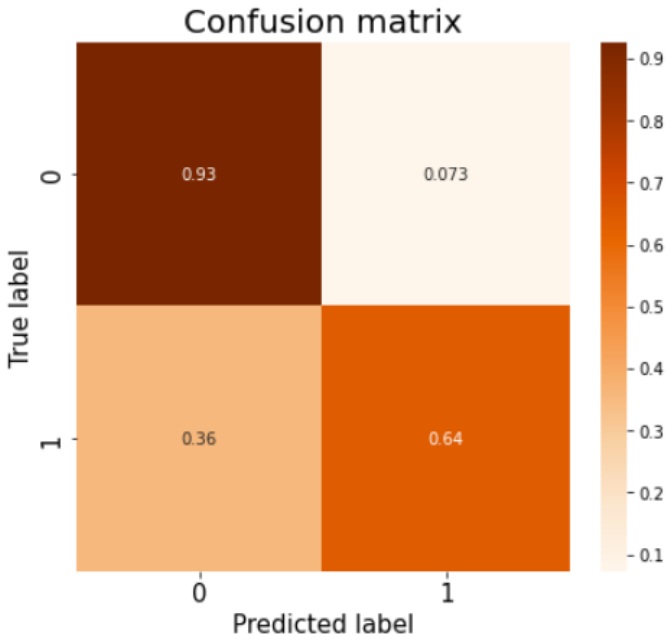


Figure 18: Confusion matrix of the Random Forest classifier

## 3.8 Support Vector Machine

The next model we are going to use is *Support Vector Machine (SVM)*.

This model relies on the idea of building an hyperplane in an high dimensional feature space which can separate our data, by maximizing the distance of this hyperplane from the nearest data point, called margin. Depending on the nature of the task, if the two classes are linearly separable, the SVM defines an unique solution and the formulation of Hard-SVM is sufficient to deal with such a problem. The Hard-SVM, in fact is defined by this optimization problem:

$$\begin{aligned} &\underset{(w,b):||w||=1}{\operatorname{argmax}} \min_{i\epsilon[m]} \quad |\langle \mathbf{w}, \mathbf{x_i} \rangle + b| \\ &s.t. \quad y_i \left( \langle \mathbf{w}, \mathbf{x_i} \rangle + b \right) > 0 \quad \forall i \end{aligned}$$

Usually data is not necessarily linearly separable, leaving us with the necessity of accepting some misclassifications. In this case we need a formulation of the SVM which is capable of accepting errors by introducing slack variables and penalizing the misclassification in the corresponding optimization problem in equation.

$$\begin{aligned} &\underset{(w,b)}{\operatorname{argmin}} \frac{\|w\|^2}{2} + C \sum_{i=1}^{m} \xi_i \\ &s.t. \quad y_i \left( \langle \mathbf{w}, \mathbf{x_i} \rangle + b \right) \geq 1 - \xi_i \quad \xi_i \geq 0, \quad \forall i \end{aligned}$$

This formulation is called Soft-SVM. The parameter $C$ present in the formula, defines a level of penalization for the slack variables. An higher level of $C$ lead to give more importance at penalizing misclassifications, while a lower value gives more importance in having a simpler model at the cost of making more errors. The parameter $C$ is called regularization parameter and it's going to be tuned as reported in Table 19

In order to enhance our previous solution we could also make use of the *kernel trick*. The aim of the *kernel trick* is to create a mapping from the original space to a feature space. In this newly defined space, we can look for a corresponding hyperplane capable of separating the two classes. This simple trick could be very powerful, because we can address the issue of having a non-linearly separable problem, by defining a mapping to a space in which this problem is linearly separable. The problem of computing effectively the transformation in such space could be extremely expensive, for such reason we can make use of *kernels*. Results are reported in Figure 20. Best parameter was $C = 0.5$.

| Parameter | Values |
|---|---|
| C | [0.5, 1, 3] |

Figure 19: Parameters for model selection of Support Vector Machine

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.85 | 0.85 | 0.91 |
| **Precision** | 0.42 | 0.43 | 0.67 |
| **Recall** | 0.74 | 0.82 | 0.38 |
| **F1-score** | 0.54 | 0.57 | 0.48 |
| **AUC** | 0.80 | 0.84 | 0.68 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.87 | 0.84 | 0.90 |
| **Precision** | 0.45 | 0.42 | 0.64 |
| **Recall** | 0.73 | 0.81 | 0.33 |
| **F1-score** | 0.56 | 0.55 | 0.44 |
| **AUC** | 0.81 | 0.83 | 0.65 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.90 | 0.85 | 0.90 |
| **Precision** | 0.56 | 0.42 | 0.69 |
| **Recall** | 0.63 | 0.87 | 0.31 |
| **F1-score** | 0.59 | 0.57 | 0.42 |
| **AUC** | 0.78 | 0.86 | 0.64 |

Figure 20: Results obtained applying SVM. From top to bottom and left to right: backward, PCA and no feature selection.

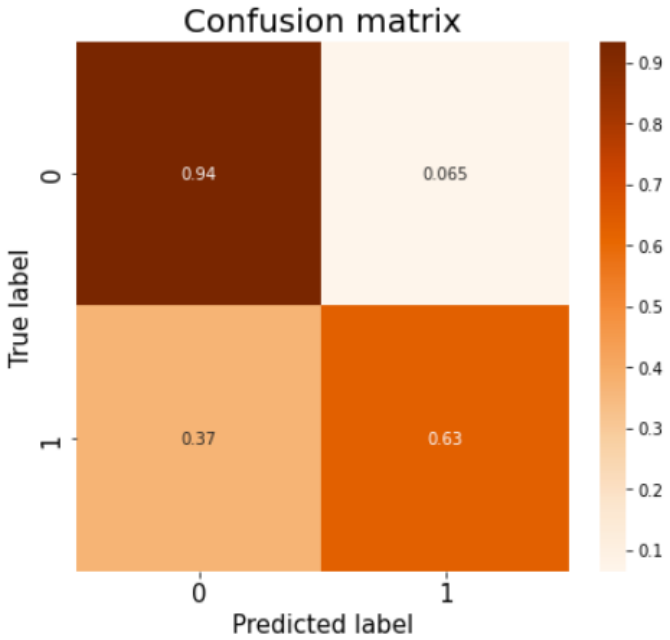


Figure 21: Confusion matrix of the SVM classifier

### 3.9 k-Nearest-Neighbor classifier

The next model we are going to evaluate is a *K-nearest-neighbors (k-NN)* classifier. The principle of the model is to assign the label of a new point by looking at the $k$ nearest training points and by performing a majority voting over their labels. k-NN does not have a proper training phase, as all the training points are only kept in memory and all the computation are made during the classification of a new point. The distance measure used to compute the nearest neighbors could be an hyperparameter to tune, in our case we choose to use the *Minkowski* distance, but it can be, in general, any metric measure. Given two vector $x$ and $y$ of dimension $d$ the distance is computed as:

$$d_p(x, y) = \left( \sum_{i=1}^{d} (x_i - y_i)^p \right)^{\frac{1}{p}} \tag{6}$$

For the default case of $p = 2$ we have the *Euclidian distance*, $p\epsilon\,[1, +\infty]$ could be an hyperparameter to use in the model selection, but we keep the Euclidean distance.

Although k-NN seems a very simple model, it still performs very well in a series of classification and regression problem, for this reason and for the simplicity of the fitting method, which could help in case the data is not all already present at training time, we chose it. In fact in case we could have new data to use for the classification, rather than refitting a classifier with k-NN we just need to update the dataset containing the training points. The hyperparameters we are going to tune are present in Table 22. $k$ determines the number of neighbors to consider in the classification, a low level of k, could lead to overfitting the model, while an high level of k could lead to underfitting. The *weights* instead, measure the amount of weight given to each training point when computing the majority voting. By choosing *uniform* we are going to give the same weight in the majority voting to all the neighbors, while with *distance* we are going to weight the points by the inverse of the distance, meaning closest points will have an higher influence than the ones further away. Results are reported in Figure 23. Best parameters found were $n\_neighbors = 10$ and $weights = uniform$.

| Parameter | Values |
|---|---|
| n_neighbors | [2, 5, 10] |
| weights | ['uniform', 'distance'] |

Figure 22: Parameters for model selection of k-nearest-neighbor

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.85 | 0.84 | 0.90 |
| **Precision** | 0.42 | 0.40 | 0.65 |
| **Recall** | 0.79 | 0.78 | 0.28 |
| **F1-score** | 0.53 | 0.57 | 0.40 |
| **AUC** | 0.82 | 0.81 | 0.63 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.82 | 0.81 | 0.90 |
| **Precision** | 0.37 | 0.36 | 0.62 |
| **Recall** | 0.77 | 0.76 | 0.26 |
| **F1-score** | 0.50 | 0.49 | 0.37 |
| **AUC** | 0.80 | 0.79 | 0.62 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.84 | 0.83 | 0.90 |
| **Precision** | 0.40 | 0.39 | 0.69 |
| **Recall** | 0.76 | 0.75 | 0.28 |
| **F1-score** | 0.53 | 0.51 | 0.39 |
| **AUC** | 0.81 | 0.80 | 0.63 |

Figure 23: Results obtained applying KNN. From top to bottom and left to right: backward, PCA and no feature selection.

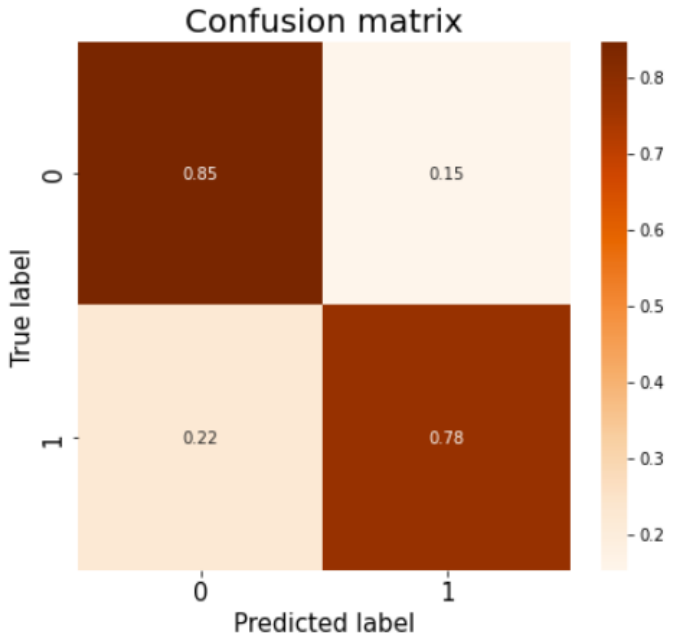


Figure 24: Confusion matrix of the KNN classifier

## 3.10 Logistic Regression

We know that some regression algorithm can be used for classification. Logistic Regression is an example of algorithm used to estimate the probability that an instance belong to a class (binary classification problem). If the predicted probability is greater than a threshold (typically 0.5) then the data point is classified as *positive class* or 1, otherwise it is classified as *negative class* or 0. So, we can build a binary classifier. In Logistic Regression a weighted sum of the inputs is computed, but instead of outputting the output as in Linear Regression, it outputs the *logistic* of this result. The logistic is a *sigmoid* function that outputs a number in the range $[0, 1]$. The logistic function is:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{7}$$

We can also have a look at the plot of this equation, in Figure 25. If the predicted probability is above 0.5 then the predicted class will be 1, otherwise it will be 0. The score (probability) is often called the *logit*. The name come from the fact that the logit function defined as $logit(p) = log(p/(1-p))$ is the inverse of the logistic function. The logit is also called the *log-odds* since it is the log of the ratio between the estimated probability for the positive class and the negative one.

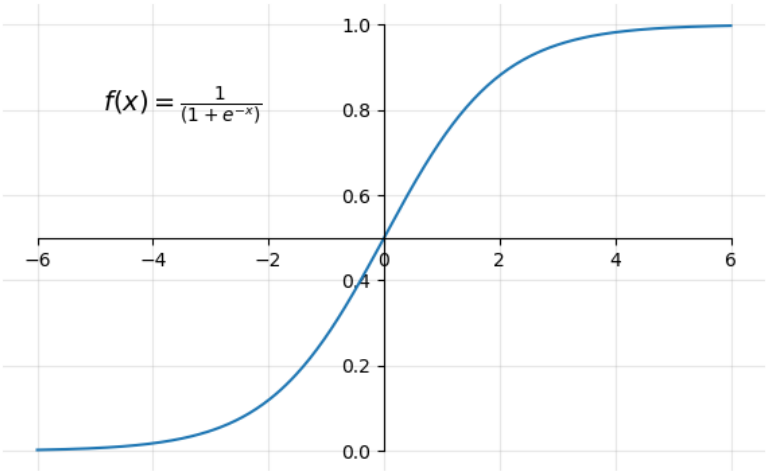


Figure 25: Logistic function

Logistic regression is a classification model that is used to model binary and multivariable outcomes.

Model selection will be performed over the hyperparameters reported in Table 26. Results are reported in Figure 27. Best parameters were $penalty = l2$, $tol = 1e-4$ and $C = 0.5$.

| Parameter | Values |
|---|---|
| penalty | ['l1', 'l2', 'elastic_net'] |
| tol | [1e-3, 1e-4, 1e-5] |
| C | [0.5, 1] |

Figure 26: Parameters for model selection of logistic regression

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.84 | 0.84 | 0.90 |
| **Precision** | 0.41 | 0.41 | 0.63 |
| **Recall** | 0.76 | 0.82 | 0.37 |
| **F1-score** | 0.53 | 0.55 | 0.46 |
| **AUC** | 0.81 | 0.83 | 0.67 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.81 | 0.81 | 0.89 |
| **Precision** | 0.36 | 0.36 | 0.58 |
| **Recall** | 0.80 | 0.81 | 0.29 |
| **F1-score** | 0.50 | 0.50 | 0.38 |
| **AUC** | 0.81 | 0.81 | 0.63 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.90 | 0.84 | 0.90 |
| **Precision** | 0.57 | 0.41 | 0.63 |
| **Recall** | 0.57 | 0.82 | 0.37 |
| **F1-score** | 0.57 | 0.55 | 0.46 |
| **AUC** | 0.76 | 0.83 | 0.67 |

Figure 27: Results obtained applying Logistic Regression. From top to bottom and left to right: backward, PCA and no feature selection.

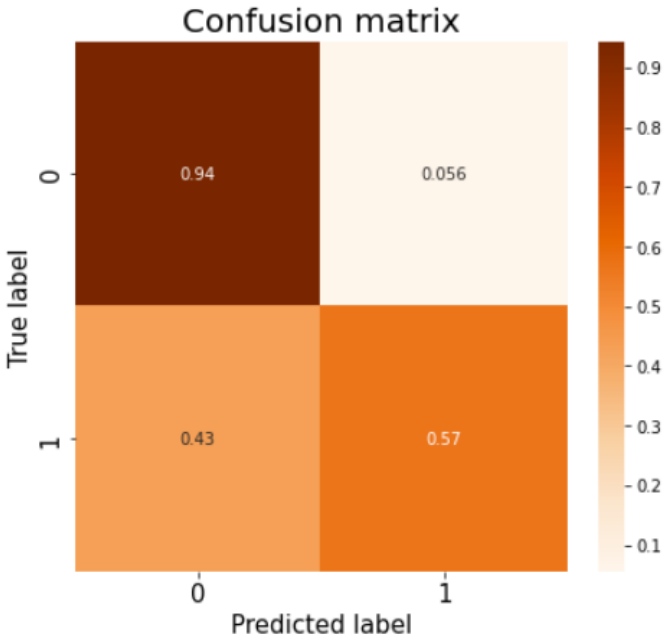


Figure 28: Confusion matrix of the Logistic Regression classifier

## 3.11 Metropolis-Hastings

We extended our research by considering a Bayesian approach. In particular, we place a *prior* on the weights define a likelihood, obtain a posterior and get predictions by averaging over all possible weights' values. The problem is that prior and likelihood are not conjugate hence we can't compute the marginal likelihood analytically and we don't know the form of the posterior. Assuming independence the likelihood is:

$$p(\mathbf{y}|\mathbf{X}, \mathbf{w}) = \prod_{n=1}^{N} p(y_n|\mathbf{x_n}, \mathbf{w}) \tag{8}$$

But there is no prior conjugate to this in the case of sigmoid probabilities defined above. A possible solution is to *sample* from the intractable posterior through *Markov Chain Monte Carlo (MCMC) sampling*. Different algorithms exist for that, but we chose to use *Metropolis-Hastings* which produces a sequence of samples $\mathbf{w}_i$ basing each proposal on the previous proposal and at each step it decides whether to accept the current proposal or not. Specifically, the algorithm consist of:

- Treat the proposal $\tilde{\mathbf{w}}_\mathbf{s}$ as a random variable conditioned on $\mathbf{w_{s-1}}$ and so we need to define $p(\tilde{\mathbf{w}}_\mathbf{s}|\mathbf{w_{s-1}}, \mathbf{\Sigma_p}) = \mathcal{N}(\mathbf{w_{s-1}}, \mathbf{\Sigma_p})$
- The choice of acceptance is given by the ratio $r = \frac{p(\tilde{\mathbf{w}}_\mathbf{s}|\mathbf{y},\mathbf{X})}{p(\mathbf{w_{s-1}}|\mathbf{y},\mathbf{X})} \cdot \frac{p(\mathbf{w_{s-1}}|\tilde{\mathbf{w}}_\mathbf{s},\mathbf{\Sigma_p})}{p(\tilde{\mathbf{w}}_\mathbf{s}|\mathbf{w_{s-1}},\mathbf{\Sigma_p})}$
- We then use the following rules: if $r \geq 1$ we accept the step, if $r < 1$ we accept with probability $r$. In this way we will eventually end by sampling from the posterior, no matter where we started.

The first ratio of $r$ is telling us that if we are moving towards a point where the posterior is higher then $r \geq 1$, else $r < 1$. We can write g since it is a ratio and so the troubling normalization constant cancels out. The second ratio instead is saying that for symmetric proposals it is 1; in case instead of skewed $p$ we want to penalize that we are doing more predictions in one direction and so we would sample more on that side (for Gaussian proposals this term is 1 since it's symmetric). We also accept proposals with probability $r$ because otherwise we would end in around the mode (doing optimization rather than sampling) not doing any other steps in other directions, hence we wouldn't capture well the true posterior distribution, so not exploring well the space. This case is reported in Figure 29 with a toy example of a weight vector $\mathbf{w}$ of 2 components.

In Metropolis-Hastings the first weight vector $\mathbf{w}_0$ is selected by hand/randomly, but we may pick values which won't let the sampling be effective (not catching well the true posterior distribution). For this reason we should set the right step size (in our case we sample from a multivariate normal and we multiply by the step size) and also the right number of samples. MH $\mathbf{w}$ can be "evaluated" looking at *trace plots* of

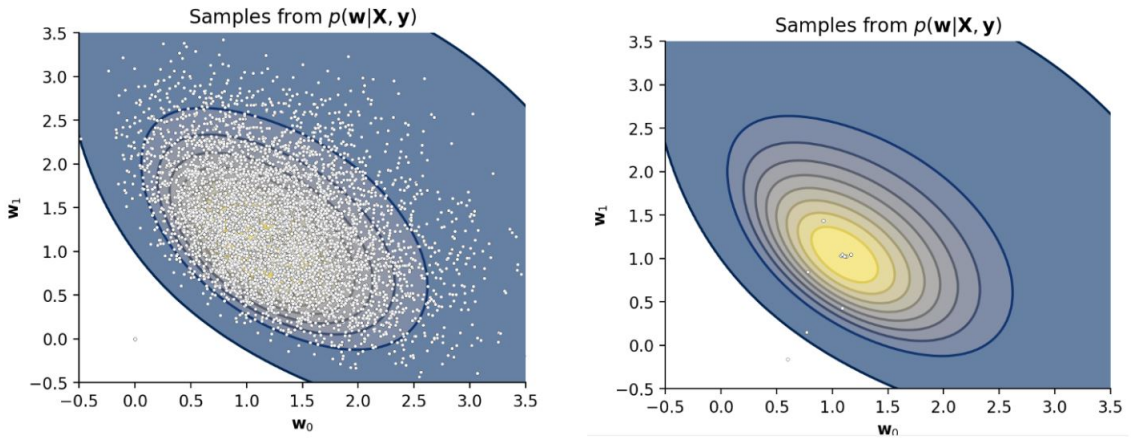


Figure 29: MH sampling with acceptance according to probability $r$ (left) and without (right)

different components of $\mathbf{w}$, also we can run different chains and plot trace plots with different plots per component. This is useful to see if the distribution is similar across chains. Also, we can use the *potential scale reduction factor* which is computed for each scalar quantity of interest, as the standard deviation of that quantity from all the chains included together, divided by the root mean square of the separate within-chain standard deviations. The key idea is that *if a set of simulations have not mixed well, the variance of all the chains mixed together should be higher than the variance of individual chains*. We typically go for a reduction factor less than 1.1

Here the parameters we tuned are the number of samples we are sampling from the intractable posterior. We have to ensure that this number is high enough since we want to model the posterior. Also, the step size is an important parameter because we want to avoid jumps which are either too big or too small in order to well-explore the space. Results are reported in Figure 32. Best parameters were $samples = 20000$ and $step\_size = 1e-3$.

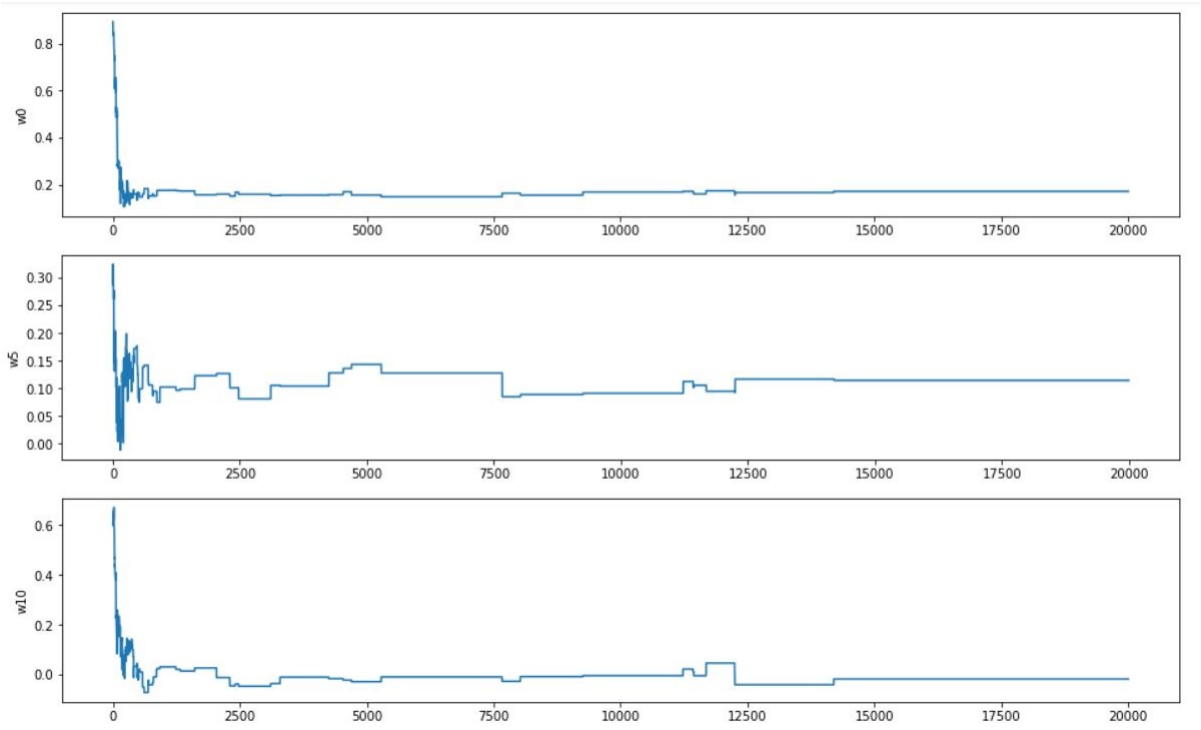


Figure 30: Trace plots of 3 weight vector's components

| Parameter | Values |
|---|---|
| samples | [10000, 20000] |
| step size | [1e-3, 1e-4] |

Figure 31: Parameters for model selection of Metropolis-Hastings

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.82 | 0.82 | 0.83 |
| **Precision** | 0.38 | 0.39 | 0.39 |
| **Recall** | 0.83 | 0.86 | 0.85 |
| **F1-score** | 0.52 | 0.53 | 0.54 |
| **AUC** | 0.82 | 0.84 | 0.84 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.75 | 0.78 | 0.79 |
| **Precision** | 0.30 | 0.33 | 0.33 |
| **Recall** | 0.90 | 0.86 | 0.84 |
| **F1-score** | 0.45 | 0.48 | 0.48 |
| **AUC** | 0.81 | 0.81 | 0.81 |

| | SMOTE | RandomOS | None |
|---|---|---|---|
| **Accuracy** | 0.80 | 0.81 | 0.80 |
| **Precision** | 0.35 | 0.37 | 0.36 |
| **Recall** | 0.86 | 0.87 | 0.89 |
| **F1-score** | 0.50 | 0.52 | 0.52 |
| **AUC** | 0.83 | 0.84 | 0.84 |

Figure 32: Results obtained applying Metropolis Hastings. From top to bottom and left to right: backward, PCA and no feature selection.

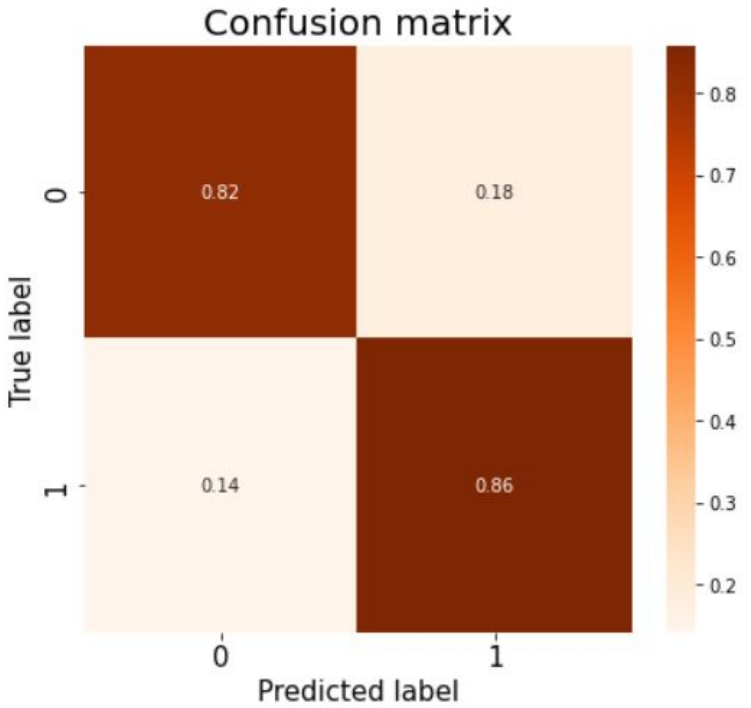


Figure 33: Confusion matrix of the Metropolis-Hastings classifier

# 4 Final considerations and conclusion

Having fitted all the different classifiers, we can give a brief overview of the results. The main metric we used for assessing the performances of a specific classifier was the *f1-score* since it is the armonic mean of *precision* and *recall*, which we preferred to *accuracy* due to imbalance problem (which however we took into account with oversampling). F1-scores showed us that the best classifiers were *Random Forest* and *SVM*, both with SMOTE application. The worst one is the *Decision Tree*, but it was something we could expect since prediction are made by taking into account only one tree, while with Random Forest we introduced the concept of majority voting. Decision tree instead, even if dealing with lower results with respect to the other classifier, we have seen that it could be useful to build a more interpretable model if used together with backward elimination or without feature selection, preserving the original feature space. We have also noted, in general, as feature selection in our context tends to lower the results of our models and that performing SMOTE oversampling hold better results with respect to Random Over sampling or not applying any rebalancing. We also worked with a Bayesian approach through *Metropolis-Hastings* which is a fascinating topic that allowed us to get predictions which took into account all possible weights' values. However, we had to handle the intractable posterior and we did that through a Markov-Chain-Monte-Carlo method (Metropolis-Hastings). Bayesian Machine Learning is a topic which is receiving increasing attention since in very many application the classification result is not enough, but the attention is on the *uncertainty* of the prediction. For example, in a medical application we would like to have a certain prediction which is not *hard*, but which has an associated probability that allow us to assess the quality of the prediction itself.