POLITECNICO DI TORINO

Master in Data Science and Engineering

Mathematics in Machine Learning

Cardiotocography classification

Professors

Prof. Francesco VACCARINO

Prof. Mauro GASPARINI

Student

Daniele GIANNUZZI

Academic Year: 2021 - 2022

Table of Contents

Chapter 1

Introduction

This work consists of a data science workflow and the exploration of its main phases for a task of binary classification.
First, in Chapter 2 there is an in-depth analysis of the *Cardiotocography dataset*. The dataset consists of measurements of fetal heart rate (FHR) and uterine contraction (UC) features on cardiotocograms classified by expert obstetricians.
In Chapter 3 the data preprocessing pipeline is analyzed: outlier removal by means of anomaly detectors such as Local Outlier Factor; scaling of the features in order to make it possible to use techniques of dimensionality reduction such as Principal Component Analysis so as to deal with the curse of dimensionality; resampling techniques to deal with an unbalanced dataset such as undersampling or oversampling. Moreover, it follows a detailed examination of K-Fold cross validation useful to split the dataset in order to tune the model and reduce the bias introduced by the data.
Chapter 4 focuses on the existing and most common metrics to evaluate a classification model. Hence, a description of the confusion metrics precedes the explanation of accuracy, precision, recall, F1-score, ROC curve and Precision-Recall curve, along with the AUC and AP scores.
Chapter 5 describes many classifiers and the rationale behind them. The classifiers considered are: Decision Tree along with its Random Forest ensemble version, Support Vector Machine and K-Nearest Neighbors. Finally, we evaluate what is the best preprocessing pipeline for each classifier, comparing then the performance of all the used classifiers.

Chapter 2

Dataset management

2.1 Data overview and description

The dataset was built in a way that 2126 fetal cardiotocograms (CTGs) were automatically processed and the respective diagnostic features measured. A summary of the 21 obtained numerical and real-valued attributes and their description are reported in Table 2.1. Moreover, we highlight that the dataset attributes do not contain null values.

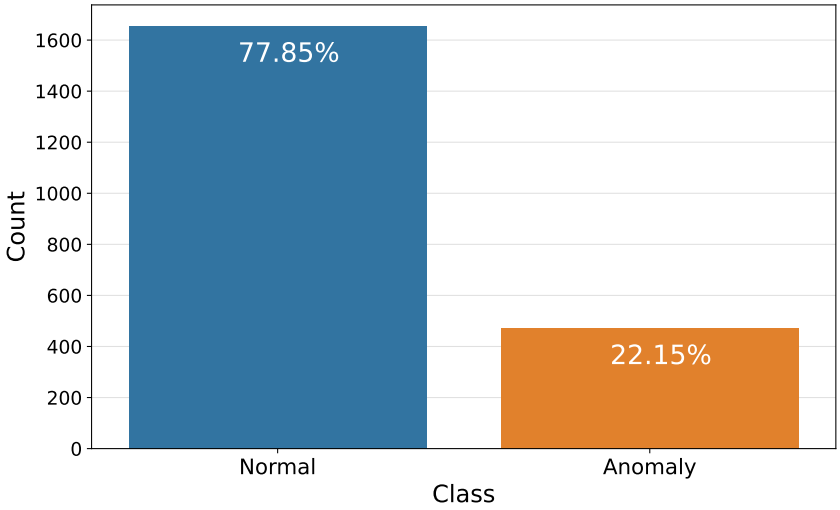

Figure 2.1: Count of number of samples per class in the dataset

The CTGs were also classified by three expert obstetricians and a consensus classification label assigned to each of them. Classification was both with respect to a morphologic pattern (A, B, C, ...) and to a fetal state: Normal (N), Suspect (S), Pathologic (P). Therefore the dataset can be used either for 10-class or 3-class experiments. In this work, we are interested in using the dataset as it is exploited in the field of anomaly detection, for which the two classes of Suspect and Pathologic are merged together into the broader

Number	Attribute	Description
1	LB	FHR baseline (beats per minute)
2	AC	# of accelerations per second
3	FM	# of fetal movements per second
4	UC	# of uterine contractions per second
5	DL	# of light decelerations per second
6	DS	# of severe decelerations per second
7	DP	# of prolongued decelerations per second
8	ASTV	percentage of time with abnormal short term variability
9	MSTV	mean value of short term variability
10	ALTV	percentage of time with abnormal long term variability
11	MLTV	mean value of long term variability
12	Width	width of FHR histogram
13	Min	minimum of FHR histogram
14	Max	maximum of FHR histogram
15	Nmax	# of histogram peaks
16	Nzeros	# of histogram zeros
17	Mode	histogram mode
18	Mean	histogram mean
19	Median	histogram median
20	Variance	histogram variance
21	Tendency	histogram tendency

Table 2.1: Attributes description for the Cardiotocography dataset

class of Anomalies.

The count of the number of samples per class present in the dataset is shown in Figure 2.1. In particular, there are 1655 samples (77.85%) for the Normal, and 471 (22.15%) samples for the Anomaly class. Therefore, the dataset is highly unbalanced.

2.2 Attributes distribution

We want to analyze how the distribution of each attribute changes given the class. In Figure 2.2 we report the results. As we can observe, the majority of the attributes distributions for both the classes are gaussian-like. However we can notice for example in case of attributes like AC, DP, ALTV how the two distributions differ in the variance which is low for one class and extremely large for the other one. For some other attribute what changes is the mean of the gaussian, as we can observe for: UC, DS, DP, ASTV, MSTV.

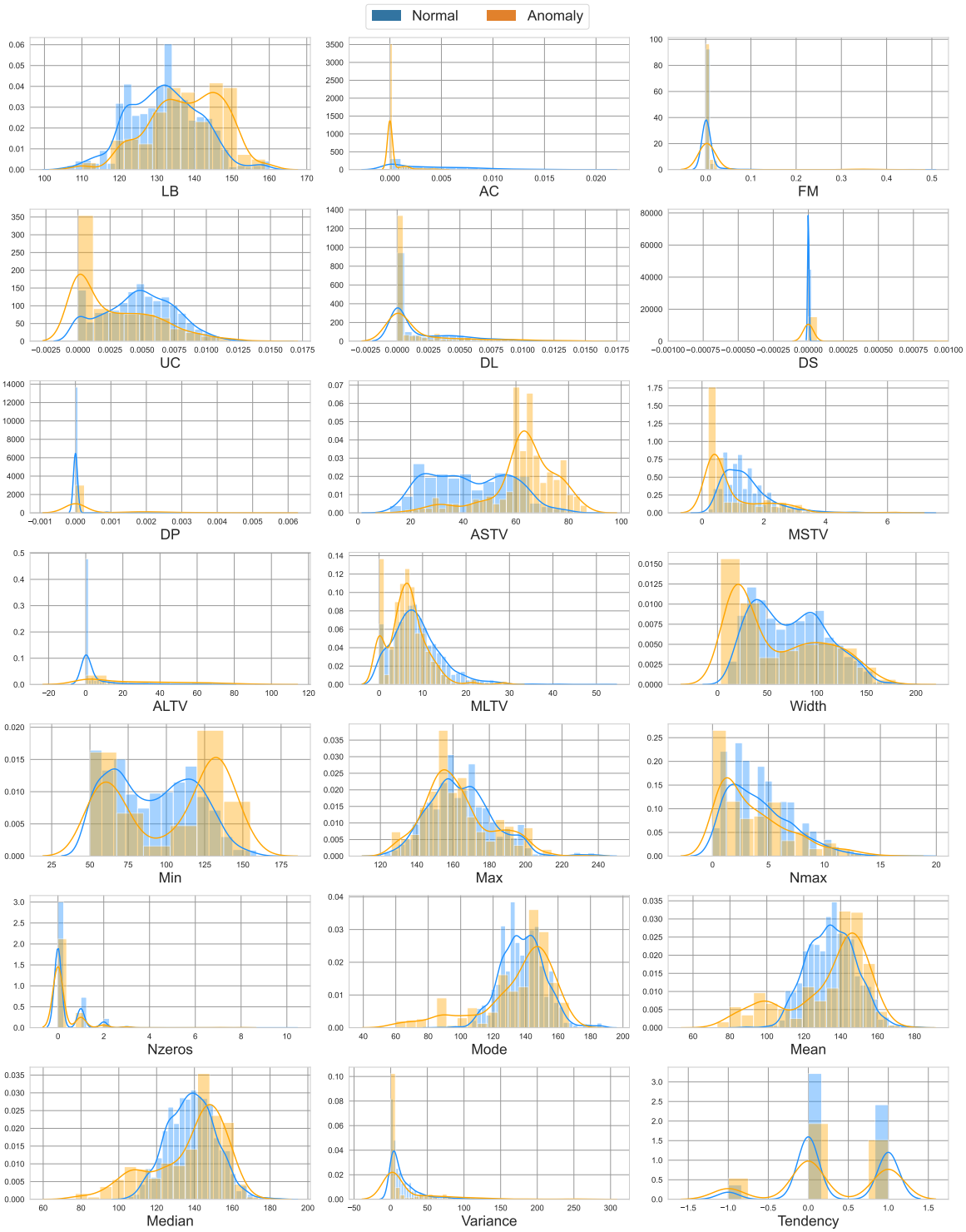

Figure 2.2: Distribution of attributes values per class

2.3 Correlation

Pearson's coefficient of correlation measures the linear relationship between a pair of variables. It falls in the value range of $[-1, +1]$. Value of -1 signifies strong negative correlation while $+1$ indicates strong positive correlation. It is obtained by taking the ratio of the covariance of two numerical variables, normalized by the product of their standard deviations.

$$\mathrm{Corr}(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \sigma_Y} \tag{2.1}$$

Covariance is a measure of the joint variability of two random variables. If the greater values of one variable mainly correspond with the greater values of the other variable, and the same holds for the lesser values (that is, the variables tend to show similar behavior), the covariance is positive and vice-versa. In the following formula, n refers to the number of samples, while μ to the mean of the underlying random variable.

$$\mathrm{Cov}(X, Y) = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \mu_X)(y_i - \mu_Y) \tag{2.2}$$

While the standard deviations of the considered random variables X and Y are computed as follows:

$$\sigma_X = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \mu_X)^2} \tag{2.3}$$

$$\sigma_Y = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (y_i - \mu_Y)^2} \tag{2.4}$$

High values of this correlation coefficient is synonym of data redundancy, so it could be helpful to remove these redundant attributes. A graphical representation of the Pearson Correlation is performed with an Heatmap, that is a data visualization technique that shows magnitude of a phenomenon as color in two dimensions. Each cell $(i; j)$ of the Heatmap represents the Person Correlation between the random variables X_i and X_j.

In Figure 2.3 we can notice how the majority of attributes are not correlated. However some positive correlation is observable among Mode, Mean and Median and this was quite imaginable since they are all related to the concept of the distribution. However, they show to have a high correlation also with respect to the LB attribute. Also the Width attribute shows to have a strong positive correlation with the attributes Max and Nmax, while it has a super strong negative correlation since it is near to -1 with the attribute Min.

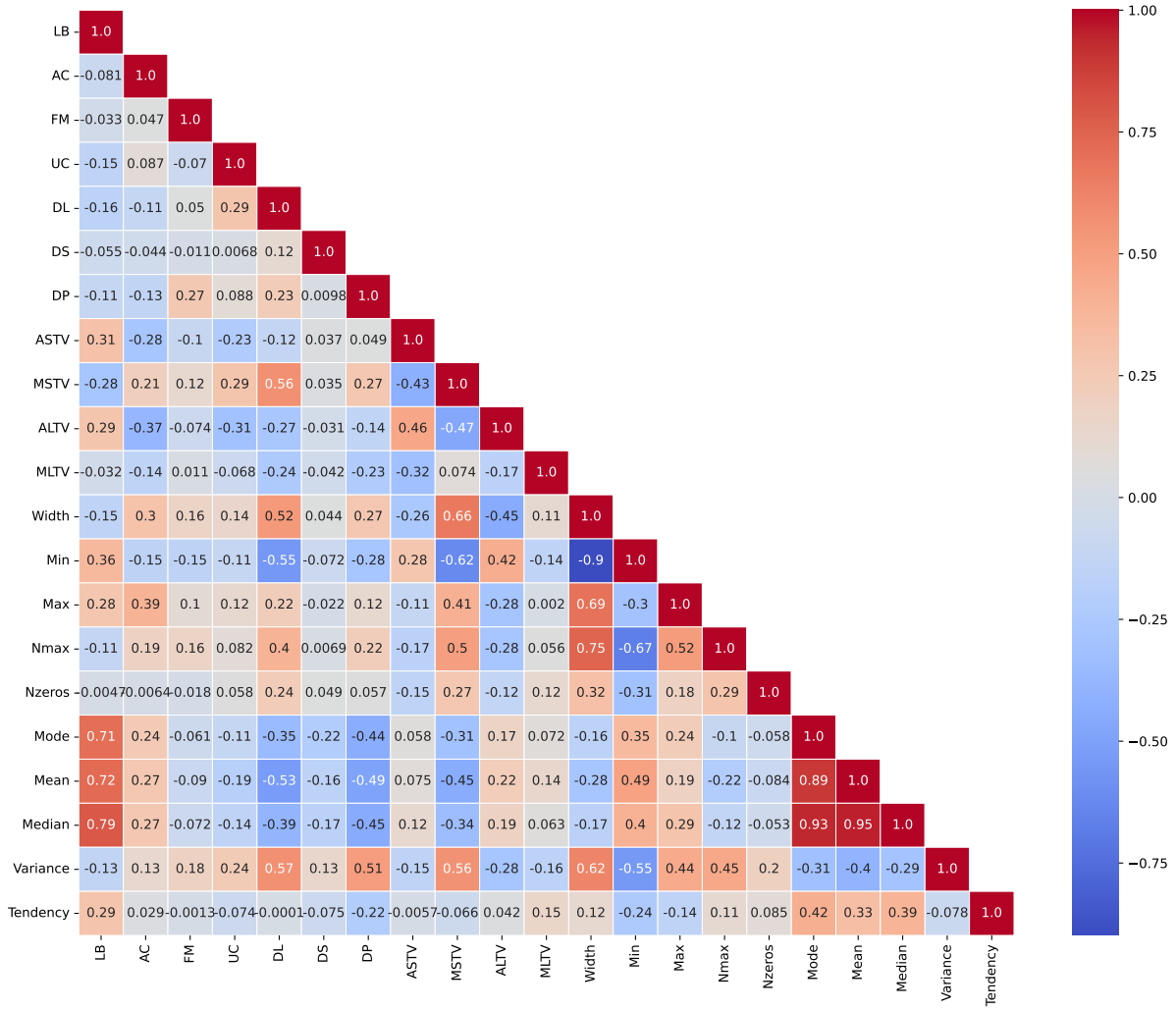

Figure 2.3: Heatmap for Pearson correlation of features

Chapter 3

Data preprocessing

3.1 Data splitting: K-Fold Cross Validation

Data splitting is the practice with which data is divided into two or more subsets. Typically, with a two-part split, one part is used to evaluate or test the data and the other to train the model. In our experiments we use 80% of dataset samples for training and the remaining 20% for testing. In order to tune the hyperparameters, and rank the models performance so as to select the best model for the task, another data splitting is necessary. We used Stratified K-Fold Cross Validation (K = 5): the original training set is partitioned into K subsets (i.e. folds). The option 'Stratified' ensures that the original ratio between positive and negative samples is preserved in each subset. Each time we take a different subset as validation set and the remaining ones as training data. We fit the model on the training data and we evaluate it on the validation data. In this way, we make an estimation of the true risk using the average error performed on the validation sets. Once obtained the best model and hyperparameters, we train again the model with both train and validation set and finally evaluate it on the test set.

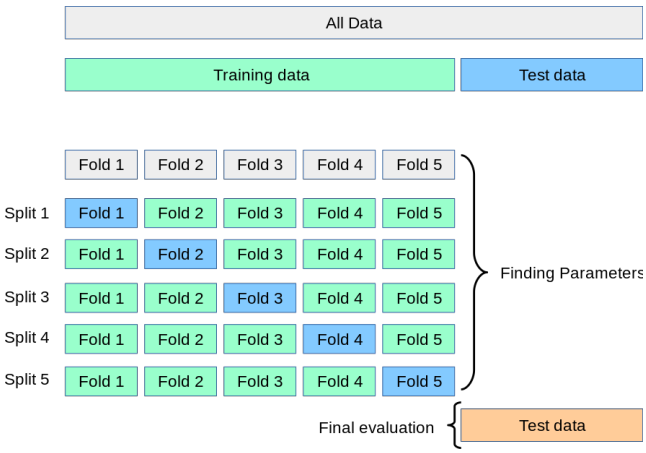

Figure 3.1: K-Fold Cross Validation

3.2 Data cleaning

Outlier detection is the process of detecting outliers in given set of data. Outliers are those data points that are significantly different from the rest of the dataset. While they can be naturally due to the data distribution, they are often abnormal observations that skew the data distribution, and arise due to inconsistent data entry, or erroneous observations. Since their presence can significantly affect the performance of a machine learning algorithm, one should evaluate if exclude them or not from the learning phase. A common practice is to train the model also without outliers and observe if its performance on the validation set increases. Many techniques exist in literature for outlier detection. We describe in the following section the Local Outlier Factor (LOF). It is important to highlight that it does not make use of labels.

3.2.1 LOF

Local outlier factor (LOF) is an algorithm used for Unsupervised outlier detection. It produces an anomaly score that represents data points which are outliers in the data set. The anomaly score of each sample is called the Local Outlier Factor. It measures the local deviation of the density of a given sample with respect to its neighbors. It is local in that the anomaly score depends on how isolated the object is with respect to the surrounding neighborhood.

To understand LOF, we should define the following concepts:

- *K-distance*: is the distance between the point, and it's K^{th} nearest neighbor.
- *K-neighbors*: denoted by $N_k(A)$ includes a set of points that lie in or on the circle of radius K-distance.
- *Reachability distance*: as in Equation 3.1 it is defined as the maximum of K-distance of o and the distance between p and o. The distance measure is problem-specific (Euclidean, Manhattan, ...). In practice, if a point p lies within the K-neighbors of o, the reachability distance will be K-distance of o, otherwise it will be the distance between p and o.

$$\mathrm{RD}\,(p, o) = \max\left(K\text{-}distance\,(o)\,,\ \mathrm{distance}\ \ (p, o)\right) \tag{3.1}$$

- *Local reachability density*: as in Equation 3.2 it is the inverse of the average reachability distance of p from its k neighbors. Intuitively according to LRD formula, more the average reachability distance (i.e., neighbors are far from the point), less density of points are present around a particular point. This tells how far a point p is from the nearest cluster of points. Low values of LRD implies that the closest cluster is far from the point.

$$\mathrm{LRD}_k(p) = \frac{1}{\left(\frac{\sum_{o \in N_k(p)} \mathrm{RD}_k(p,o)}{|N_k(p)|}\right)} \tag{3.2}$$

Then, LRD of each point is used to compare with the average LRD of its K neighbors. Then, the *Local Outlier Factor* (LOF) is the ratio of the average LRD of the K neighbors of p to the LRD of p, as in Equation 3.3.

$$LOF_k(p) = \frac{\sum_{o \in N_k(p)} \frac{\mathrm{LRD}_k(o)}{\mathrm{LRD}_k(p)}}{|N_k(p)|} = \frac{\sum_{o \in N_k(A)} \mathrm{LRD}_k(o)}{|N_k(p)|\,\mathrm{LRD}_k(p)} \tag{3.3}$$

This ratio will assume values similar to 1 if there is similar density as neighbors, lower than 1 if there is higher density than neighbors, higher than 1 if there is a lower density than neighbors. The last case is the case of an outlier. As by default we use 20 neighbors and the euclidean distance in our implementation.

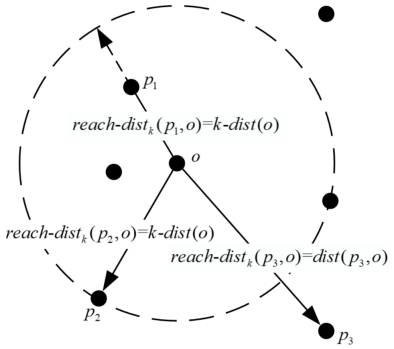

Figure 3.2: Illustration of Local Outlier Factor measures

3.3 Feature scaling: standardization

Feature scaling is one of the most important data preprocessing step in machine learning. Algorithms that compute the distance between the features are biased towards numerically larger values if the data is not scaled. Standardization or Z-Score Normalization is the transformation of features x by subtracting from mean μ and dividing by standard deviation σ. In this way, the features have zero mean and unit variance.

$$z = \frac{x - \mu}{\sigma} \tag{3.4}$$

Standardization can be helpful in those cases where the data follows a Gaussian distribution. However, this does not have to be necessarily true. Moreover, standardization does not get affected by outliers because there is no predefined range of transformed features, as it happens instead for normalization.

3.4 Dimensionality reduction

Dimensionality reduction is the process of taking data in a high dimensional space and mapping it into a new space whose dimensionality is much smaller. There are several reasons to reduce the dimensionality of the data. First, high dimensional data impose computational challenges. Moreover, in some situations high dimensionality might lead to poor generalization abilities of the learning algorithm. For example, in Nearest Neighbor classifiers the sample complexity increases exponentially with the dimension. Finally, dimensionality reduction can be used for interpretability of the data, for finding meaningful structure of the data, and for illustration purposes. Following, we describe an unsupervised technique for dimensionality reduction.

3.4.1 Principal Component Analysis

Principal component analysis (PCA) is a technique for reducing the dimensionality of such datasets, increasing interpretability but at the same time minimizing information loss. The basic idea behind PCA is to map a vector of a p dimensional space to a lower-dimensional vector in a q dimensional space (where $q < p$). It does so by creating new uncorrelated variables, the principal components, a smaller number of representative variables that collectively explain most of the variance in the original set. PCA is quite sensitive regarding the variances of the initial variables, since variables with larger variance ranges dominate over those with small ranges. Therefore, standardization prior to PCA is a critical step to perform. The first principal component Z_1 of a set of p features X is the linear combination of the features that allow to the principal component Z_1 to have the largest variance:

$$\begin{aligned} Z_1 =& a_{11}X_1 + \cdots + a_{p1}X_p \\ & \text{s.t. } \sum_{i=1}^{p} a_{i1}^2 = 1 \end{aligned} \tag{3.5}$$

The coefficients a_{ij} of the linear combinations that define the principal components are called loadings and they compose the principal component loading vector $\mathbf{a}_1 = (a_{11} \ldots, a_{p1})^T$. Moreover, the constraint is necessary otherwise we could have principal components with arbitrarily large variance.

If we consider a dataset X that is $[n \times p]$ so as to have n records and p attributes, with each attribute that has 0 as mean. Therefore the problem is to find the linear combination of the samples attributes values, as it is represented in the following equation for the first principal component, maximizing the sample variance and respecting the previous coefficient constraint:

$$z_{i1} = a_{11}x_{i1} + \cdots + a_{p1}x_{ip} \tag{3.6}$$

For each principal component we have a different loading vector. We now consider the scatter matrix: $\mathbf{A} = \mathbf{X}^{\mathrm{T}}\mathbf{X}$ The scatter matrix is a statistic that is used to make estimates of the covariance matrix. Indeed, it is similar to the covariance matrix, since the attributes

are already centered in 0, with the exception of the scaling factor that is $\frac{1}{n}$ or $\frac{1}{n-1}$. Given the scatter matrix definition, we want to solve the following optimization problem:

$$\begin{aligned} &\max_{a_1} \boldsymbol{a}_1^T \boldsymbol{A} \boldsymbol{a}_1 \\ &\text{s.t. } \boldsymbol{a}_1^T \boldsymbol{a}_1 = 1 \end{aligned} \tag{3.7}$$

If we use the Lagrangian formulation in order to solve the problem, farther setting the derivatives to 0, we find that $\boldsymbol{a}_1$ is an eigenvector of $\boldsymbol{A}$:

$$\boldsymbol{A}\boldsymbol{a_1} = \lambda_1 \boldsymbol{a_1} \tag{3.8}$$

If we use this result in the objective function, we find that:

$$\max_{a_1} \lambda_1 \tag{3.9}$$

Hence, we take the maximum eigenvalue, that represents the proportion of variance explained. Since we proposed the case of finding the first principal component, we also can explain how to find the other principal components. In practice, one has to solve the same optimization problem with a constraint that impose orthogonality with the previous component (e.g. $\boldsymbol{a_2}$ orthogonal to $\boldsymbol{a_1}$). In this way we find a new principal component that explain the maximal variance out of all linear combinations, being at the same time uncorrelated with the others. Therefore, in order to obtain q principal components, we take the q eigenvectors related to the q greatest eigenvalues. The proportion of variance that will be explained is ratio of the sum of the used eigenvalues and the sum of all the eigenvalues that is the total variance.

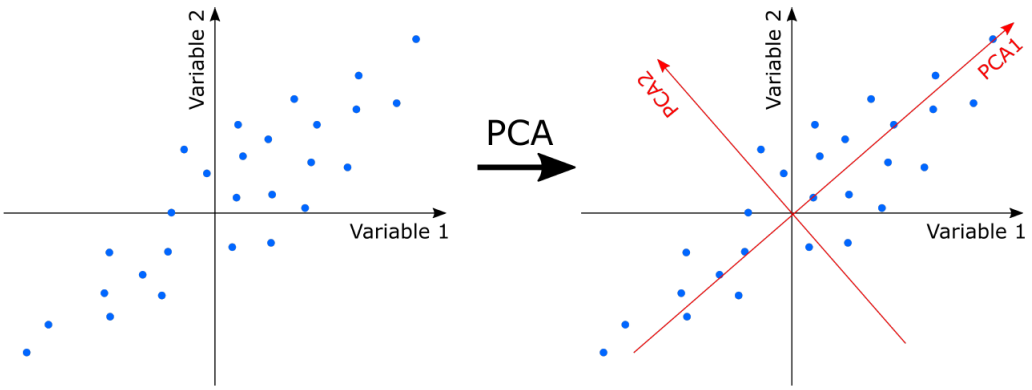

Figure 3.3: PCA illustration

The Figure 3.4 shows the variance explained by each found principal component and the cumulative one. As a rule of thumb, one would take a consistent variance percentage (e.g. 80% - 90%) to determine the number of principal components. Another possible way to proceed is to choose the number of components looking for a consistent drop in

explained variance. Since this process of using PCA is done to reduce the dimensions of the problem, if no consistent drops are present one should evaluate a trade-off between the proportion of explained variance and the number of components, hence avoiding to take all the principal components, even if the cumulative variance reach values of 100%. In this case we decided to take 9 principal components that correspond to a explained variance ratio of 85%.

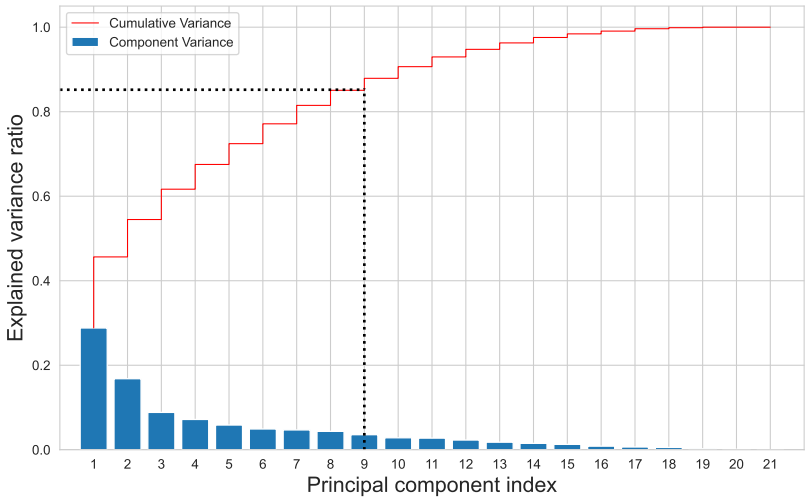

Figure 3.4: PCA: variance explained on 21 components

3.5 Resampling

When we are dealing with unbalanced datasets, as it commonly happens for many practical applications and also in this case, the learning model can be biased in predictions so as to favor the class that has the majority of samples. A technique we can use to tackle this problem regards re-balancing the dataset. In binary classification, this can be done through under-sampling the class that present the greatest number of samples or over-sampling the rare class. This can be also generalized to cases with more than two classes. Another option is combining this two techniques.

3.5.1 Undersampling

Under-sampling is a technique with which we discard some samples belonging to the majority classes. The drawback is that we loose some useful information. We can use different ways in order to perform under-sampling. Here we apply NearMiss. It is based on a KNN approach. NearMiss implements 3 different types of heuristic:

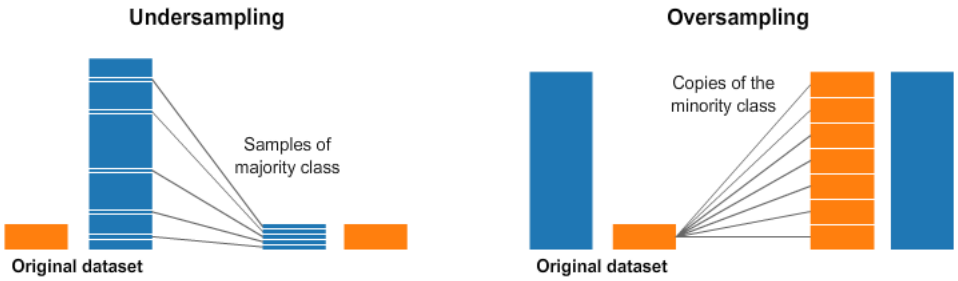

Figure 3.5: Resampling techniques: under-sampling and over-sampling

1. NearMiss-1: removes the samples of the majority class for which the average distance to the N closest samples of the minority class is the smallest. In the normal case, samples next to the boundaries will be selected. However, the drawback is that this can be altered by the presence of noise.

2. NearMiss-2: removes the samples of the majority class for which the average distance to the N farthest samples of the minority class is the smallest. NearMiss-2 does not have the same drawback of NearMiss-1 since it does not focus on the nearest samples but rather on the farthest samples. However, the drawback in this case is the presence of marginal outliers that can alter the sampling.

3. NearMiss-3: is a 2-steps algorithm. First, for each sample of the minority class, their M nearest-neighbors will be kept. Then, the samples of the majority class removed are the one for which the average distance to the N nearest-neighbors is the largest.

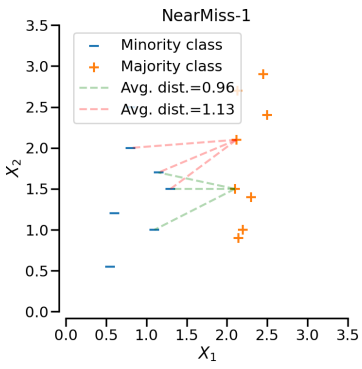

Figure 3.6: NearMiss1

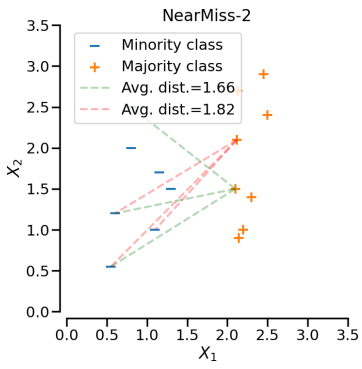

Figure 3.7: NearMiss2

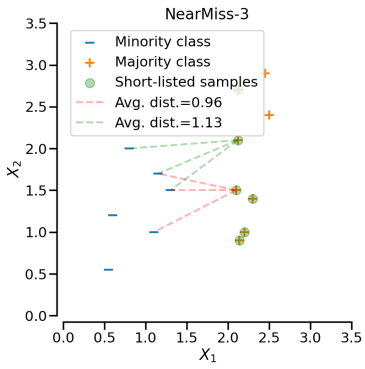

Figure 3.8: NearMiss3

Since NearMiss-3 is probably the version which will be less affected by noise due to the first step sample selection, we use it in our work, with default hyperparameters $N = 3$ and $M = 3$

3.5.2 Oversampling

Over-sampling is a technique with which we generate new data for the minority classes. The advantage of using this technique is that we do not loose information.
A technique that is well known in this field is called SMOTE [1], that stands for Synthetic Minority Oversampling TEchnique. The advantage of SMOTE is that you are not generating duplicates, but rather creating synthetic data points that are slightly different from the original data points.
The SMOTE algorithm works as follows:

1. we draw a random sample from the minority class
2. we identify its k nearest neighbors belonging to the same class
3. we take one of those neighbors and identify the vector between the current sample and the selected neighbor
4. we multiply the vector by a random number between 0 and 1
5. we add this value to the current sample to obtain the synthetic data point

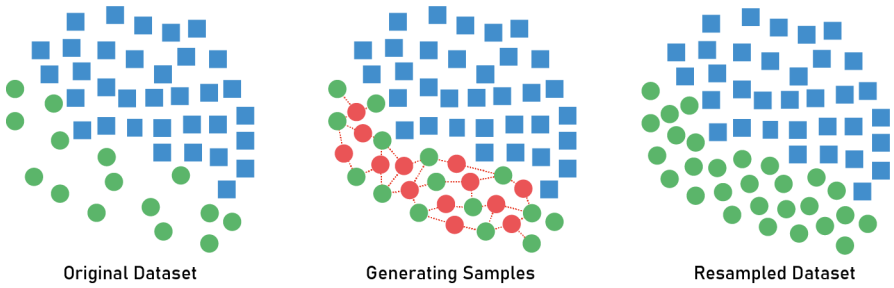

Figure 3.9: Illustration of usage of SMOTE for over-sampling

The drawback of this method is that we can generate samples that overlaps with the other classes, increasing the overlap in the distribution.

3.5.3 Combination of under-sampling and over-sampling

We can also combine under-sampling and over-sampling, in order to achieve better performance. SMOTEENN is a combination of SMOTE for over-sampling and ENN for under-sampling. ENN stands is the acronym of Edited Nearest Neighbours. In ENN [2], undersampling of the majority class is done by removing points whose class label differs from a majority of its k nearest neighbors (K=3 by default). SMOTEENN [3] applies first SMOTE until the desired proportion of minority class is met. Then ENN is used to under-sample the majority class.

In Figure 3.10 we can observe how the different re-sampling approaches used on our dataset lead to a different quantity of samples for each of the two classes.

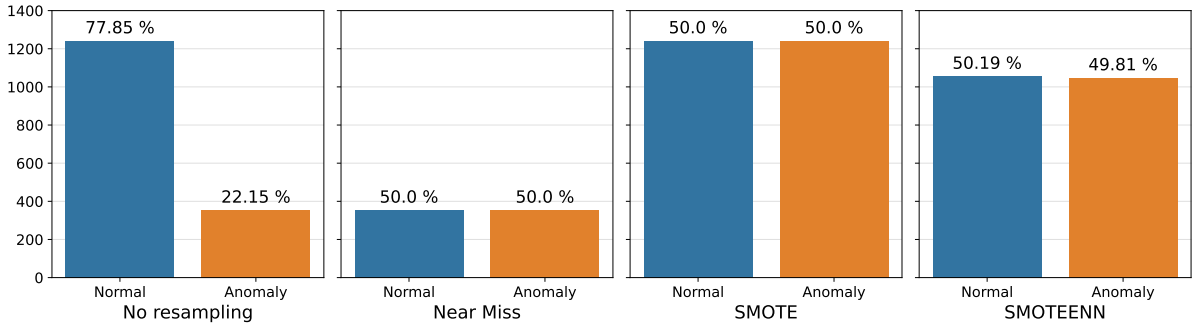

Figure 3.10: Comparison of different resampling techniques on our dataset

Chapter 4

Model evaluation

4.1 Confusion matrix and associated metrics

With model evaluation we try to measure how the model performs on data that it has never seen before. We can use different metrics depending on the task (i.e. classification or regression), the data imbalance and the aim of our study. Since we are dealing with a binary classification task, we report in Figure 4.1 some of the most common performance metrics that are used, given the confusion matrix, for which the name stems from the fact that it makes it easy to see whether the system is confusing two classes, reporting TP: True Positive; TN: True Negative; FP: False Positive; FN: False Negative.

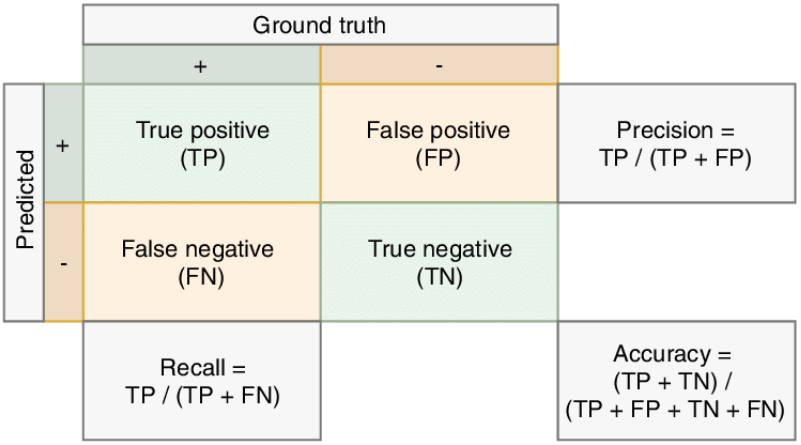

Figure 4.1: Illustration of a confusion matrix with the associated evaluation metrics

- **Accuracy**: ratio between the number of correctly classified samples and the total number of samples.

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + FN + TN} \tag{4.1}$$

- **Precision**: the ratio of the number of correctly predicted positive observations and the total number of predicted positive observations. Since Precision combines both positive and negative samples, it is class prior dependent.

$$\text{Precision } = \frac{TP}{TP + FP} \tag{4.2}$$

- **Recall**: the ratio of the number of correctly predicted positive observations and the total number of positive observations. Since Recall is only dependent on positive samples, it is independent of class priors.

$$\text{Recall } = \frac{TP}{TP + FN} \tag{4.3}$$

- **F1-score**: the harmonic mean of Precision and Recall.

$$F1 = 2 \cdot \frac{\text{Precision } \cdot \text{ Recall}}{\text{Precision } + \text{ Recall}} \tag{4.4}$$

Accuracy is a metric that performs well in balanced datasets. We should use it if both classes (positive and negative) are equally important. F1-score on the other hand is better suited for imbalanced classification, using the Harmonic Mean to penalize extreme values. In case False Negatives and False Positives are of the same importance F1-score is the proper metric to use.

4.2 ROC curve

A Receiver Operating Characteristic curve, or ROC curve, is a graphical plot that illustrates the diagnostic ability of a binary classifier system as its discrimination threshold is varied. The ROC curve is created by plotting the true positive rate $\text{TPR} = \frac{\text{TP}}{\text{P}} = \frac{\text{TP}}{\text{TP+FN}}$ against the false positive rate $\text{FPR} = \frac{\text{FP}}{\text{N}} = \frac{\text{FP}}{\text{FP+TN}}$ at various threshold settings. An advantage of the ROC curve is that it shows you sensitivity (i.e. TPR) and specificity (i.e. $TNR = 1 - FPR$) at all possible thresholds, so if you find a point that represents the right tradeoff for your use case, you can choose the threshold that goes with that point on the curve.

One way we have to express the performance on the ROC curve in terms of a single number is using its AUC. AUC stands for "Area under the Curve". It provides an aggregate measure of performance across all possible classification thresholds. That is, AUC measures the entire two-dimensional area underneath the entire ROC curve

4.3 Precision-Recall curve

The precision-recall curve shows the tradeoff between precision and recall for different threshold. A high area under the curve represents both high recall and high precision,

where high precision relates to a low false positive rate, and high recall relates to a low false negative rate. Also in this case, we can use the AUC to aggregate the curve information into a single number. In general, we prefer to use PR curves when there is a moderate to large class imbalance, as ROC curves present an optimistic picture of the model due to the use of true negatives (TN) in the FPR.

Chapter 5

Model selection

5.1 Decision tree

Decision tree is a type of model used for both classification and regression. A tree answers sequential questions which send us down a certain path of the tree given the answer. Each internal node of the Tree represents a "test" on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label (the decision taken given that particular path). The paths from root to leaf represent classification rules. For this reasons, a decision tree result is extremely easy to interpret, it can handle both numerical and categorical features and it is extremely fast. We can observe an example of a simple decision tree in Figure 5.1.

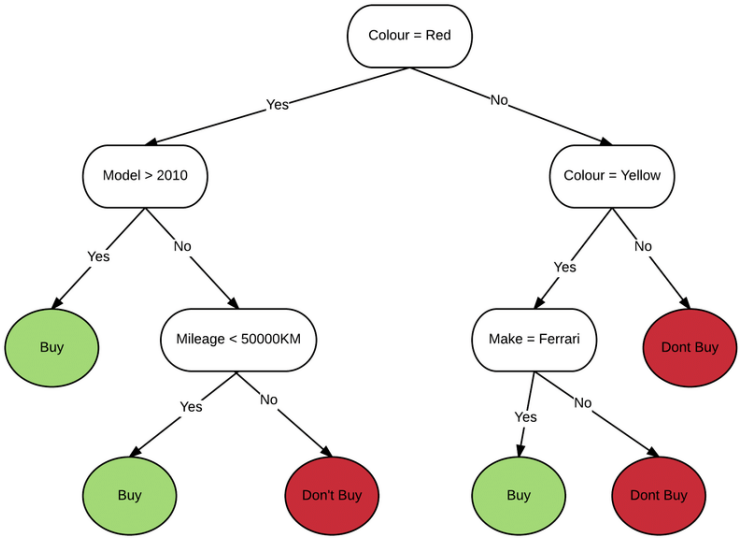

Figure 5.1: An example of a simple decision tree

The primary challenge in the decision tree implementation is to identify which attributes do we need to consider as the root node at each level. Handling this is to know as the attributes selection. Some implementations are based on heuristic approaches, using a top-down greedy search through the space of possible branches with no backtracking. A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. These implementations begin with the original set S (training set) as the root node. Since we have different attributes selection measures to identify the attribute which can be considered as the root note at each level, for each node of the tree, the algorithm iterates through the very unused attribute of the set S and calculates Entropy(H) and Information gain(IG) of this attribute. It then selects the attribute which has the smallest Entropy or Largest Information gain.

Given: $p(j \mid t)$ the relative frequency of the class j at node t; k the number of partitions (children of a node); n_i the number of records at child t over that partition; n the number of records in the node; we can define:

- Gini: it performs only binary splits. Higher value of Gini index implies higher inequality, higher heterogeneity. Its value is in the range: $[0,\ (1 - n_{classes}^{-1})]$. The highest value of this index is reached when records are equally distributed among all classes, implying the highest impurity degree, while its minimum value is 0 when all records belong to one class, implying the lowest impurity degree, after which it would have no sense to further split.

$$\text{Gini}(t) = 1 - \sum_j p(j \mid t)^2 \tag{5.1}$$

  The quality of the split is then calculated as follows:

$$\text{Gini}_{\text{split}} = \sum_{t=1}^{k} \frac{n_i}{n} \text{Gini}(t) \tag{5.2}$$

- Entropy: it is a measure of the randomness in the information being processed. The higher the entropy, the harder it is to draw any conclusions from that information. Therefore, our aim is to minimize the entropy. Its value is in the range: $[0,\ \log_2(n_{classes})]$.

$$\text{Entropy}(t) = -\sum_j p(j \mid t) \log_2 p(j \mid t) \tag{5.3}$$

  Information gain (IG) is a statistical property that measures how well a given attribute separates the training examples according to their target classification. It computes the difference between entropy before split and average entropy after split of the dataset based on given attribute values. Constructing a decision tree is about finding an attribute that returns the highest IG.

$$\text{GAIN}_{\text{split}} = \text{Entropy}(p) - \sum_{t=1}^{k} \frac{n_i}{n} \text{Entropy}(t) \tag{5.4}$$

The set S is then split by the selected attribute to produce a subset of the data. The algorithm continues to recur on each subset, considering only attributes never selected before.

5.1.1 Evaluation

We now use the decision tree, performing GridSearchCV on the hyperparameters in Table 5.1 for each possible preprocessing pipeline. The hyperparameter "max_depth" is referred to the maximum depth of the tree. If it is 'None' then nodes are expanded until all leaves are pure or until all leaves contain less than 2 samples (where 2 is another parameter that is set here by default). The problem of leaving a tree grow indefinitely is overfitting, while introducing a maximum depth can improve the generalization performance. The hyperparameter "criterion" has been already discussed in the previous section.

We highlight that since standardization is always performed in the pipeline, we avoid to report it in the following graph. In Figure 5.2 we can observe how the best performance of the decision tree is obtained using only the standardization in the pipeline. On the right we can observe the confusion matrix related to the best configuration, as well as the tuned hyperparameters that are in bold in Table 5.1.

Hyperparameter	**Values**
criterion	gini, **entropy**
max_depth	2, 5, **10**, 20, 50, 100, None

Table 5.1: Decision Tree GridSearch. In bold the values related to the best configuration

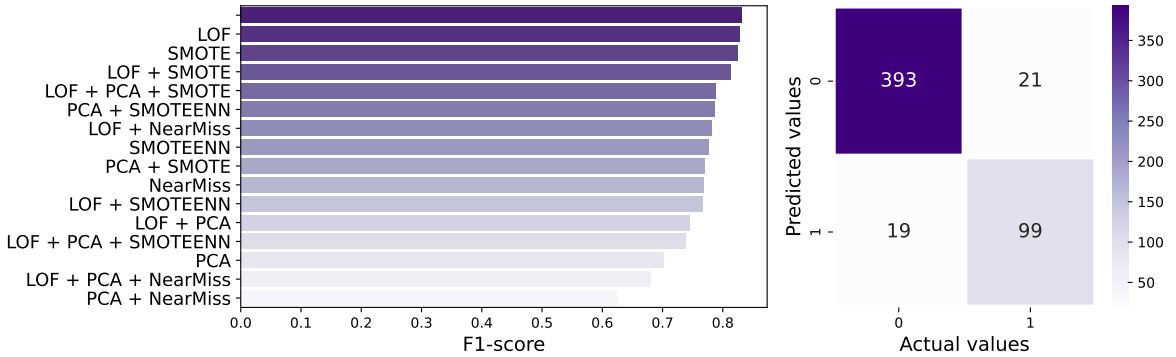

Figure 5.2: Comparison of different pipelines with a Decision Tree as classifier tuned with a GridSearchCV, along with the confusion matrix referred to the best configuration

5.2 Random Forest

Random Forest[4] is an example of ensemble learning, in which we combine multiple machine learning algorithms to obtain better predictive performance.

Two are the key concepts that give it the name random:

1. Random sampling of training dataset when building trees.

   This is based on a technique known as "bootstrap aggregation" or "bagging" that is used to create an ensemble of trees where multiple training sets are generated with replacement from the original one. In the bagging technique, a data set is divided into N samples using randomized sampling. Then, using a single learning algorithm a model is built on all samples. Later, the resultant predictions of the trees learned on different training sets are combined using voting or averaging in parallel. This is a procedure useful to reduce the variance related to a statistical learning method, since given a set of independent random variables $Z_1, ..., Z_n$, each having variance σ^2, the variance of their mean: Z_{mean} is σ^2/n.

2. Random subsets of features considered when splitting nodes.

   In a specific tree, each time we consider a split, a random sample of m predictors is chosen as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors, where generally $m \approx \sqrt{p}$. The reason is that otherwise the majority of the trees will be similar, using the main strong predictors and providing highly correlated predictions.

5.2.1 Evaluation

We now use the random forest, performing GridSearchCV on the hyperparameters in Table 5.2 for each possible preprocessing pipeline. The hyperparameter "max_depth" and "criterion" has been already discussed in the Section 5.1.1 about the decision tree. Here we tune the algorithm also on two more hyperparameters: "n_estimators" regulates the number of trees in the forest, while "max_features" refers to the random subsets of features considered when splitting nodes as previously described.

Hyperparameter	Values
n_estimators	11, 22, **51**, 101, 201
criterion	gini, **entropy**
max_depth	2, 5, 10, **20**, 50, 100, None
max_features	**sqrt**, log2

Table 5.2: Random Forest GridSearch. In bold the values related to the best configuration

We highlight that since standardization is always performed in the pipeline, we avoid to report it in the following graph. In Figure 5.3 we can observe how the best performance

of the random forest is obtained using only the standardization in the pipeline. On the right we can observe the confusion matrix related to the best configuration, as well as the tuned hyperparameters that are in bold in Table 5.2.

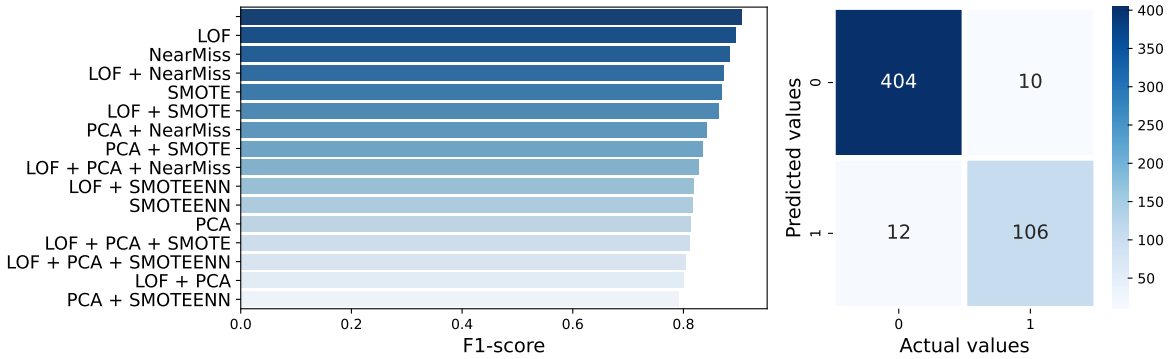

Figure 5.3: Comparison of different pipelines with a Random Forest as classifier tuned with a GridSearchCV, along with the confusion matrix referred to the best configuration

5.3 SVM

SVM or Support Vector Machine[5] is a linear model for classification and regression problems. It can solve linear and non-linear problems and work well for many practical problems. The idea of SVM is simple: the algorithm creates a line or a hyperplane which separates the data into classes.

5.3.1 Hard margin

Given a binary problem that is linearly separable, many hyperplanes can exist that separates the two classes as we can observe in Figure 5.4. The margin of a hyperplane with respect to a training set is defined to be the minimal distance between a point in the training set and the hyperplane. Hence, support vectors are those data points that stay on the margin and are closer to the hyperplane, influencing the position and orientation of the hyperplane. If a hyperplane has a large margin, then it will still separate the training set even if we slightly perturb each instance. Therefore, with Hard-SVM the aim is to find the hyperplane that leads to the largest possible margin, as Figure 5.5 shows.

If we want to mathematically describe the problem, given a training set $\{x_k, y_k\}_{k=1}^{N}$, input patterns $y_k \in \mathbb{R}$, output patterns $y_k \in \mathbb{R}$ where $y_k \in \{-1, +1\}$. One can do a rescaling of the problem such that $\min_i \left|w^T x_i + b\right| = 1$, i.e. the rescaling is done such that the point closest to the hyperplane has a distance $1/\|w\|_2$. The margin between the classes is then equal to $2/\|w\|_2$. Maximizing the margin corresponds then to minimizing $|w\|_2$. Hence, if we assume:

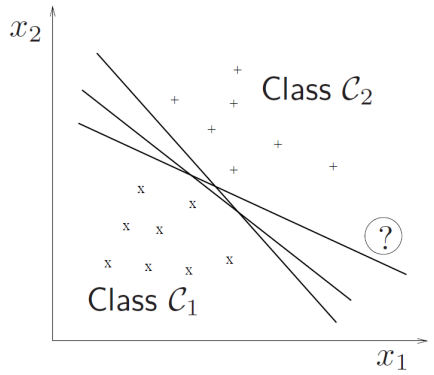

Figure 5.4: Possible hyperplanes

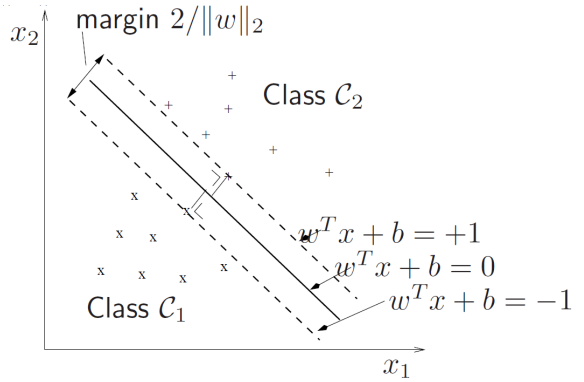

Figure 5.5: Largest margin hyperplane

$$\begin{cases} w^T x_k + b \geq +1 & , \quad \text{if} \quad y_k = +1 \\ w^T x_k + b \leq -1 & , \quad \text{if} \quad y_k = -1 \end{cases} \tag{5.5}$$

This is equivalent to Equation 5.6, where we require that all training data are correctly classified.

$$y_k \left[w^T x_k + b \right] \geq 1, \quad k = 1, \ldots, N \tag{5.6}$$

Therefore, Equation 5.7 describes how the optimization problem becomes. The objective is to maximize the margin subject to the constraint that we want to correctly classify all training data.

$$\min_{w,b} \frac{1}{2} w^T w \text{ s.t. } y_k \left[w^T x_k + b \right] \geq 1, \quad k = 1, \ldots, N \tag{5.7}$$

5.3.2 Soft margin

The Hard-SVM formulation assumes that the training set is linearly separable, which is a rather strong assumption. Indeed, for most real-life probems when taking a linear classifier not all the data points of the training set will be correctly classified as one can observe in Figure 5.6.

Soft-SVM can be viewed as a relaxation of the Hard-SVM rule that can be applied even if the training set is not linearly separable. Therefore, one modifies the inequalities with slack variables ξ_k such that the original inequalities can be violated for certain points if needed, modifying the optimization problem as follows:

$$\begin{aligned} &\min_{w,b,\xi} \mathcal{J}(w, \xi) = \frac{1}{2} w^T w + c \sum_{k=1}^{N} \xi_k \\ s.t. &\begin{cases} y_k \left[w^T x_k + b \right] \geq 1 - \xi_k, \quad k = 1, \ldots, N \\ \xi_k \geq 0, \quad k = 1, \ldots, N. \end{cases} \end{aligned} \tag{5.8}$$

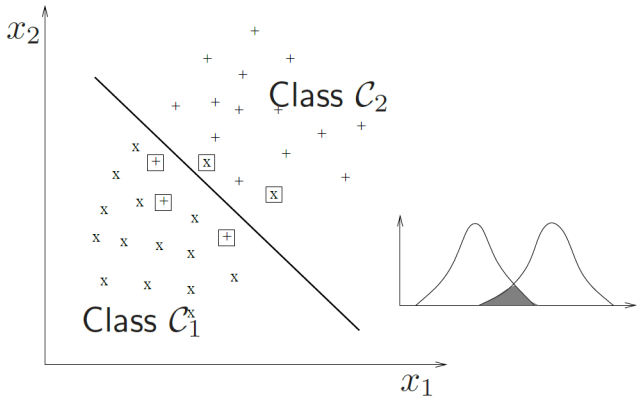

Figure 5.6: Problem of non-separable data, due to overlapping distributions

5.3.3 Kernel trick and Mercer condition

An important progress in SVM theory has been made when linear theory has been extended to nonlinear models. In order to achieve this, one maps the input data into a high dimensional feature space which can be potentially infinite dimensional. A construction of the linear separating hyperplane is done then in this high dimensional feature space, after a nonlinear mapping $\varphi(x)$ of the input data to the feature space, as we can observe in Figure 5.7. Therefore, the first constraint present in Equation 5.8 becomes:

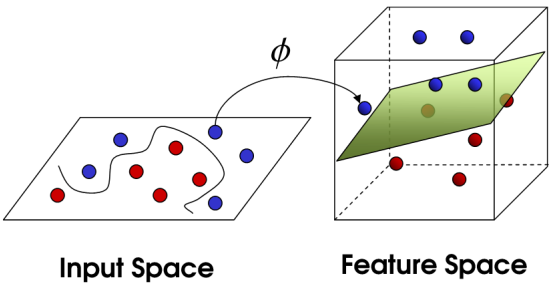

Figure 5.7: Linear separating hyperplane after mapping in high dimensional space

$$y_k \left[w^T \varphi \left(x_k \right) + b \right] \geq 1, \quad k = 1, \ldots, N \tag{5.9}$$

However, we do not need an explicit construction of the nonlinear mapping $\varphi(x)$ if we make use of the Mercer theorem, also known as kernel trick. This derives as an important consequence of the dual representation of the problem as reported in Equation 5.10. This solves the problem in Lagrangian multipliers so as the dimension of the feature space need not affect the computation.

$$\max_{\alpha_k} \mathcal{Q}(\alpha) = -\frac{1}{2}\sum_{k,l=1}^{N} y_k y_l K\left(x_k, x_l\right)\alpha_k\alpha_l + \sum_{k=1}^{N}\alpha_k. \qquad \begin{cases} \sum_{k=1}^{N}\alpha_k y_k = 0 \\ 0 \le \alpha_k \le c, k = 1, \ldots, N \end{cases} \tag{5.10}$$

We make use of the Mercer condition by choosing a positive definite[1] kernel. This is a function K such that for all $x_k, x_l \in X$:

$$K\left(x_k, x_l\right) = \varphi(x_k)^T\varphi(x_l) \tag{5.11}$$

As one does not represent the feature vectors explicitly, the number of operations required to compute the inner product by evaluating the kernel function is not necessarily proportional to the number of features. The only information used about the training examples is their kernel matrix K in the feature space.

Several choices are possible for the kernel $K(\cdot,\cdot)$:

- $K\left(x, x_k\right) = x_k^T x$ (linear SVM)
- $K\left(x, x_k\right) = \left(\gamma x_k^T x + \tau\right)^d$ (polynomial SVM of degree d and $\tau \ge 0$)
- $K\left(x, x_k\right) = \exp\left(-\gamma \left\|x - x_k\right\|_2^2\right)$ (RBF SVM)
- $K\left(x, x_k\right) = \tanh\left(\gamma x_k^T x + \tau\right)$ (sigmoid SVM)

The Mercer condition holds for all γ values in the RBF case but not for all possible choices of γ, τ in the sigmoid case. Therefore the use of an sigmoid kernel is not popular.

5.3.4 Evaluation

We now use the SVM, performing GridSearchCV on the hyperparameters in Table 5.3 for each possible preprocessing pipeline. The hyperparameter "kernel" specifies the kernel type to be used in the algorithm; "gamma" regards the kernel coefficient for 'rbf' and 'poly' kernels. In practice, it regulates the non-linearity introduced by the kernel. If "gamma"='scale' it uses $1/(n\_features \times X.var())$ as value of gamma, if 'auto', it uses $1/n\_features$. C is the regularization parameter. The strength of the regularization is inversely proportional to C. It must be strictly positive and the penalty associated is a squared l2 penalty. We highlight that since standardization is always performed in the pipeline, we avoid to report it in the following graph. In Figure 5.8 we can observe how the best performance of the SVM is obtained using only the standardization in the pipeline. On the right we can observe the confusion matrix related to the best configuration, as well as the tuned hyperparameters that are in bold in Table 5.3.

[1] A positive-definite kernel is a generalization of a positive-definite function or matrix. A positive definite matrix is a symmetric matrix where every eigenvalue is positive.

Hyperparameter	Values
kernel	linear, **rbf**, poly
gamma	**scale**, auto
C	0.1, 0.2, 0.5, 1, 2, 5, **10**, 20, 50, 100

Table 5.3: SVM GridSearch. In bold the values related to the best configuration

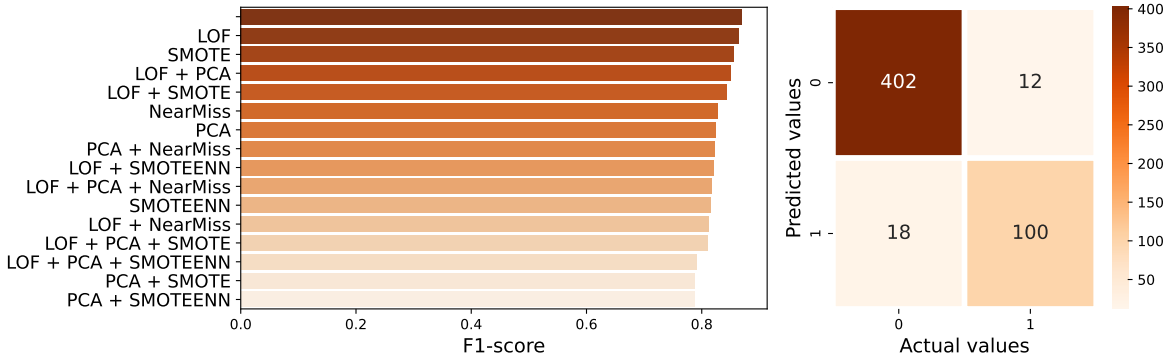

Figure 5.8: Comparison of different pipelines with a SVM as classifier tuned with a GridSearchCV, along with the confusion matrix referred to the best configuration

5.4 KNN

Nearest Neighbor algorithms are among the simplest of all machine learning algorithms. The idea is to memorize the training set and then to predict the label of any new instance on the basis of the labels of its closest neighbors in the training set. The rationale behind such a method is based on the assumption that the features that are used to describe the domain points are relevant to their labelings in a way that makes close-by points likely to have the same label.
Given $K \geq 1$ and the test sample x_0, first of all the KNN looks for the K closest samples in the training set based on a defined distance metric. Then, given the set of neighbors N_0 the algorithm estimates the conditional probability for the sample x_0 to belong to the class j as the fraction of the number of samples belonging to the class j in the neighborhood. This is reported mathematically in Equation 5.12.

$$\Pr\left(Y = j \mid X = x_0\right) = \frac{1}{K} \sum_{i \in N_0} I\left(y_i = j\right) \tag{5.12}$$

Then the class with the highest conditional probability is assigned to the sample x_0. The choice of the distance metric (e.g. Manhattan, Euclidean, Minkowski ...) and the size of the neighborhood K is fundamental for the performance of this classifier. As K grows, the decision boundary becomes smoother and more linear, while lower values of K lead to have more non-linearity and makes it easier to overfit.

5.4.1 Evaluation

We now use the KNN, performing GridSearchCV on the hyperparameters in Table 5.4 for each possible preprocessing pipeline. The hyperparameter "n_neighbors" refers to the number of neighbors to use. We already described its influence in the non-linearity of the decision boundary. The "weights" hyperparameter refers to the weight function used in prediction. If 'uniform' all points in each neighborhood are weighted equally; if 'distance' the KNN weights points by the inverse of their distance. In this case, closer neighbors of a query point will have a greater influence than neighbors which are further away. "p" is the power parameter for the Minkowski distance: $D(X,Y) = (\sum_{i=1}^{n} |x_i - y_i|^p)^{1/p}$, where X and Y are the samples for which we want to measure the distance and n is the number of features. When $p = 1$, this is equivalent to using the Manhattan distance (l1) and when $p = 2$ we are using the Euclidean distance (l2). We highlight that since standardization is always performed in the pipeline, we avoid to report it in the following graph. In Figure 5.9 we can observe how the best performance of the KNN is obtained using a pipeline made of: standardization and undersampling using the NearMiss algorithm. We can easily understand why for this classifier these steps make the classification better. Due to its nature, KNN suffers from unbalanced dataset, since it is more likely that more neighbors of a sample are belonging to the majority class. For this reason, using undersampling (NearMiss) or oversampling (SMOTE) improves the performance. Furthermore, the KNN classifier makes the assumption that similar points share similar labels. Unfortunately, in high dimensional spaces, points that are drawn from a probability distribution, tend to never be close together, so it can suffer from the curse of dimensionality. This is the reason for which using PCA with 9 principal components (85% of explained variance ratio) provides better results than using the KNN without it as a preprocessing step. Moreover, if 'K' value is low, the model can be susceptible to outliers since the prediction depends on few samples. For this reason removing outliers with LOF as a preprocessing step using also others preprocessing steps such as PCA and undersampling produced significant improvement in performance.

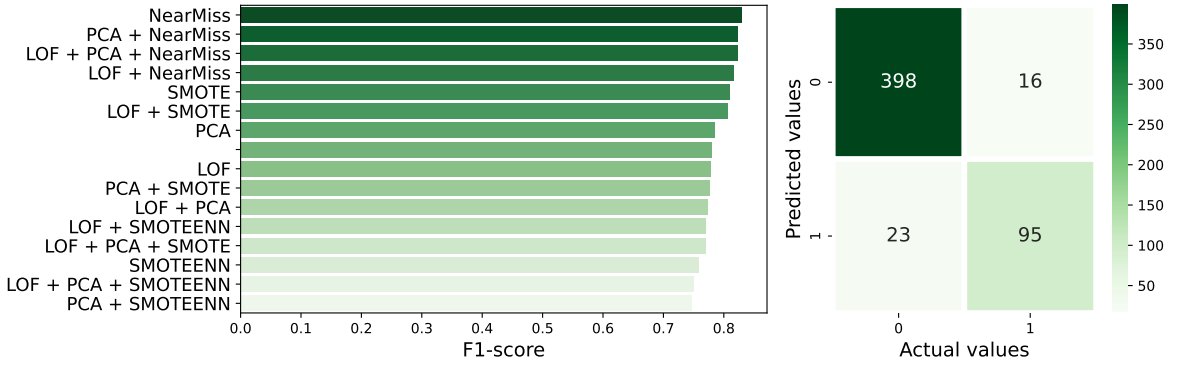

Figure 5.9: Comparison of different pipelines with a KNN as classifier tuned with a GridSearchCV, along with the confusion matrix referred to the best configuration

Hyperparameter	Values
n_neighbors	3, 5, **11**, 21, 51, 101
weights	uniform, **distance**
p	1, **2**, 3, 4

Table 5.4: KNN GridSearch. In bold the values related to the best configuration

On the right of Figure 5.9 we can observe the confusion matrix related to the best configuration found, as well as the tuned hyperparameters that are in bold in Table 5.4.

5.5 Comparison of different models

In this section we compare the different classifiers, each with the best suited pipeline. As the Figure 5.10 shows, the Random Forest is clearly outperforming the other models when considering the F1-score. The second best model in performance is the SVM with only standardization on data. However, it seems that Decision Tree and KNN have comparable performance, even if the latter uses a different pipeline, using undersampling for the reasons explained in the Section 5.4.1.

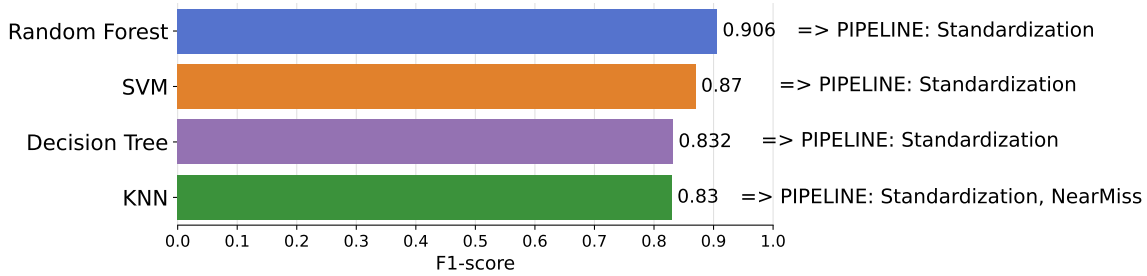

Figure 5.10: F1-score comparison of different classifier each with the best pipeline found

5.5.1 ROC curve

When we consider the ROC curve to compare the performance of our models, we evaluate the entire range of FPR and TPR since we are probably interested in a performance that allows us to decide the preferred trade-off.
In Figure 5.12 we can observe how the models perform for different thresholds. They all perform better than a random classifier, that is what we expect as a minimum requirement. Moreover, we can observe how the Random Forest outperforms all the other models for all the possible thresholds. On the other side, also the SVM appears to be a good model to use even if its performance is slightly lower than the RF one. This conclusion is also clear looking at the AUC for both the models: 0.989 for the RF and 0.980 for the SVM. For the same reason we should derive that the KNN with AUC=0.946 outperforms the Decision

Tree with AUC=0.906, but this is not true, because depending on the threshold we are interested in, we could prefer the decision tree, approximately in the FPR range that goes from 0.05 to 0.12.

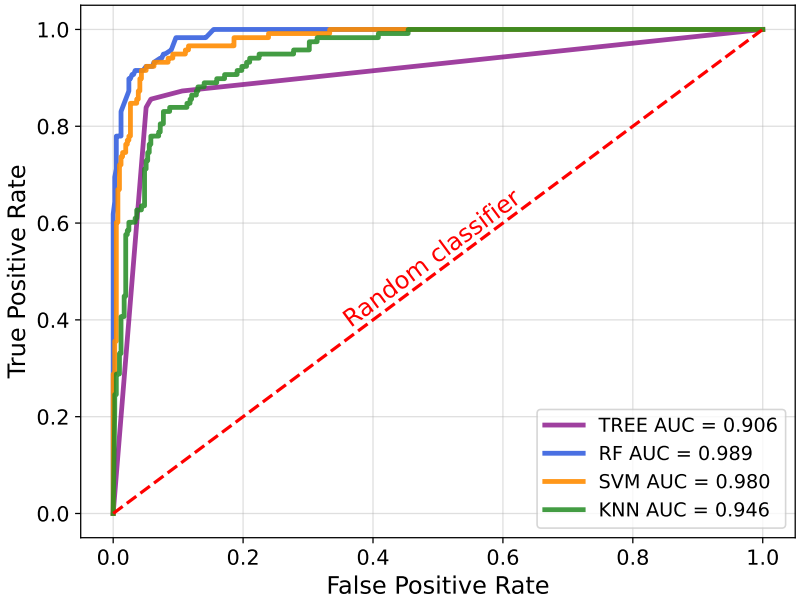

Figure 5.11: ROC curve comparison for the best configuration of each classifier

5.5.2 Precision-Recall curve

If we evaluate the performance of the models on the Precision-Recall curve we can derive approximately the same results. Firstly, all the models curves are above the curve of the random classifier. Secondly, the only difference we can observe is that the Average Precision of the Decision Tree is much lower than the KNN one even if this is not clear if one looks at the PR AUC (of which the AP is an approximation). Indeed it seems that the Decision Tree is better than the KNN for a Recall that goes from 0 to 0.9. However, we should consider that the thresholds that are generated considering the scores (i.e. probabilities) produced by the decision tree, are lower in number. This is the reason for which there is that evident knee in its curve. Therefore, we would also have less choice in the evaluation of the Precision-Recall trade-off in choosing a proper threshold. However this is not the case, since the Random Forest with his AP=0.966 outperforms also in this case all the other models, and it results to be the best choice in any case and for every possible threshold and trade-off.

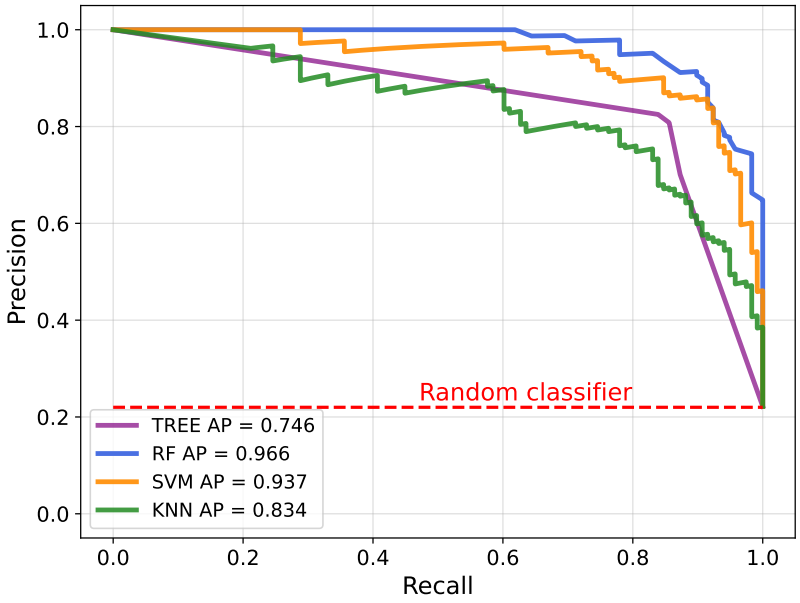

Figure 5.12: PR curve comparison for the best configuration of each classifier

Bibliography

[1] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. «SMOTE: synthetic minority over-sampling technique». In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357 (cit. on p. 14).

[2] Dennis L Wilson. «Asymptotic properties of nearest neighbor rules using edited data». In: *IEEE Transactions on Systems, Man, and Cybernetics* 3 (1972), pp. 408–421 (cit. on p. 14).

[3] Gustavo EAPA Batista, Ronaldo C Prati, and Maria Carolina Monard. «A study of the behavior of several methods for balancing machine learning training data». In: *ACM SIGKDD explorations newsletter* 6.1 (2004), pp. 20–29 (cit. on p. 14).

[4] Leo Breiman. «Random forests». In: *Machine learning* 45.1 (2001), pp. 5–32 (cit. on p. 22).

[5] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014 (cit. on p. 23).