Chapter 1

Introduction

Cyber Physical Systems (CPS) are nowadays widely used in different application domains, such as smart-homes, smart-cities, hospitals, etc... They are mainly composed of two entities: a cyber part consisting in a computing and networking component, and a physical part consisting in different controllers and sensors. The existence of a connected cyber part implies its susceptibility to multiple cyber threats. The malfunctioning of these systems, due to a cyber threat, can cause severe impacts on the real life and the safety of the community, for example a blackout or water contamination. That is why many algorithms have been designed for the security monitoring of those systems, in particular the anomaly and attack detection.

Nowadays, machine and deep learning algorithms are used to detect those anomalies and intrusions. But, in majority, they rely only on the cyber part of the systems and on the data describing their behaviour, ignoring their physical models. The idea behind this work is to employ a hybrid machine learning algorithm, in particular neural networks, to detect anomalies and attacks in CPS considering its physical model.

1.1 Physic guided machine learning in literature

As mentioned before, the aim of the work is to fuse the black-box and theory-based models together to get better predictions. However this is not the first time such a fusion is examined. In the literature various approaches of the fusion of neural networks with theory-based models were presented. Those approaches can be divided into two types given what aspect of the algorithm they're changing: those that modify in first place the input to take into consideration the physical constraints, and those that modify the structure of the neural network.

–>here comes some more explanations–<

Chapter 2

Case study

In order to focus on the implementation of the hybrid machine learning algorithm, a CPS, with ready to use datasets, was chosen from a list provided in [1]: the **power system** [2], which network diagram was represented on figure 2-1. The system is composed of two power generators who are alimenting the whole system. Intelligent Electronic Devices (IEDs) R1 to R4 and the breakers BR1 to BR4 can be found connected directly to those generators. Each IED switches its corresponding breaker when a fault is detected, valid or fake. The communication between the IEDs and the Substation Switch is done wirelessly. On the other hand the Substation Switch is connected with the Primary Domain Controller (PDC) and the Control Room.

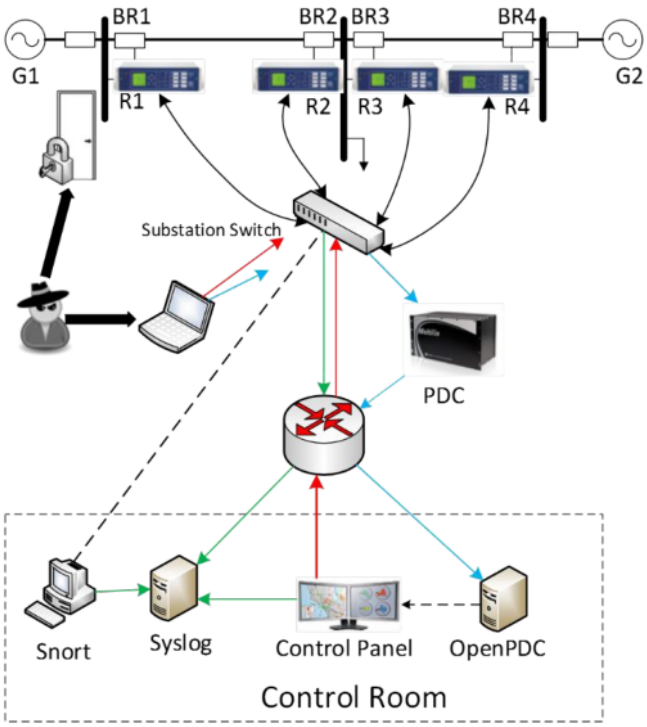


Figure 2-1: Power system network diagram [2]

The operation of this power system can be described following 6 main scenarios:

- normal behaviour,
- short-circuit,
- line maintenance,
- remotely opening the breakers (attack),
- disruption of fault protection system (attack),
- fault imitation (attack).

Each of those scenarios can be divided into several sub-scenarios concerning different entities of the system or/and the failure range. Every scenario was labelled with a number between 1 and 41. In this way **37 scenarios** are obtained, divided and numbered as follows:

- 1 no events scenario, its number it is 41,
- 8 natural fault scenarios, its number ranges are 1-6 (short-circuit) and 13-14 (line maintenance),
- 28 attack scenarios, its number ranges are 7-12 (fault imitation), 15-20 (remotely opening the breakers), 20-30 and 35-40 (disruption of fault protection system).

The reason for dropping the numbers between 31 and 34 in the naming process of scenarios is not known.

The datasets provided in [1] represent **78377 events**, in which one of those scenarios was reproduced in the system. They have been grouped by scenario into 3 datasets: binary (attack or normal operation), three-class (attack, normal fault and no events) and multiclass (differentiating all 37 scenarios). Each of these 3 datasets is composed of 15 .arff or .csv files comporting in average 141 events for each of 37 scenarios. The exact number of events per file for each scheme is illustrated on figure 2-2. For the 3 class dataset **55663 attack**, **18309 natural fault** and **4405 normal operation** events were found. The distribution of these schemes throughout the files is shown on figure 2-3.

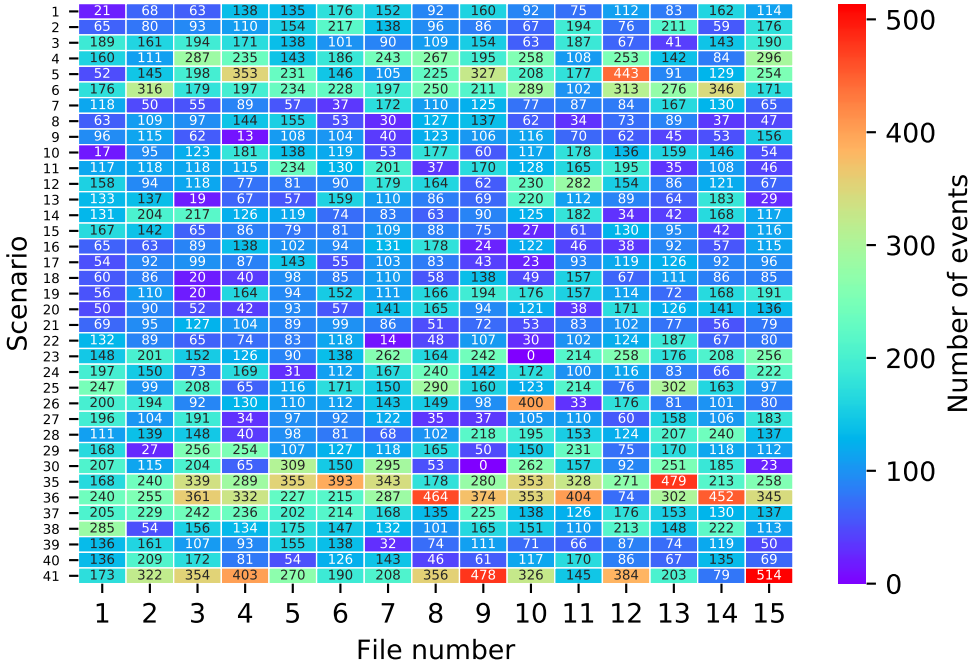


Figure 2-2: Scenarios distribution throughout all 15 files

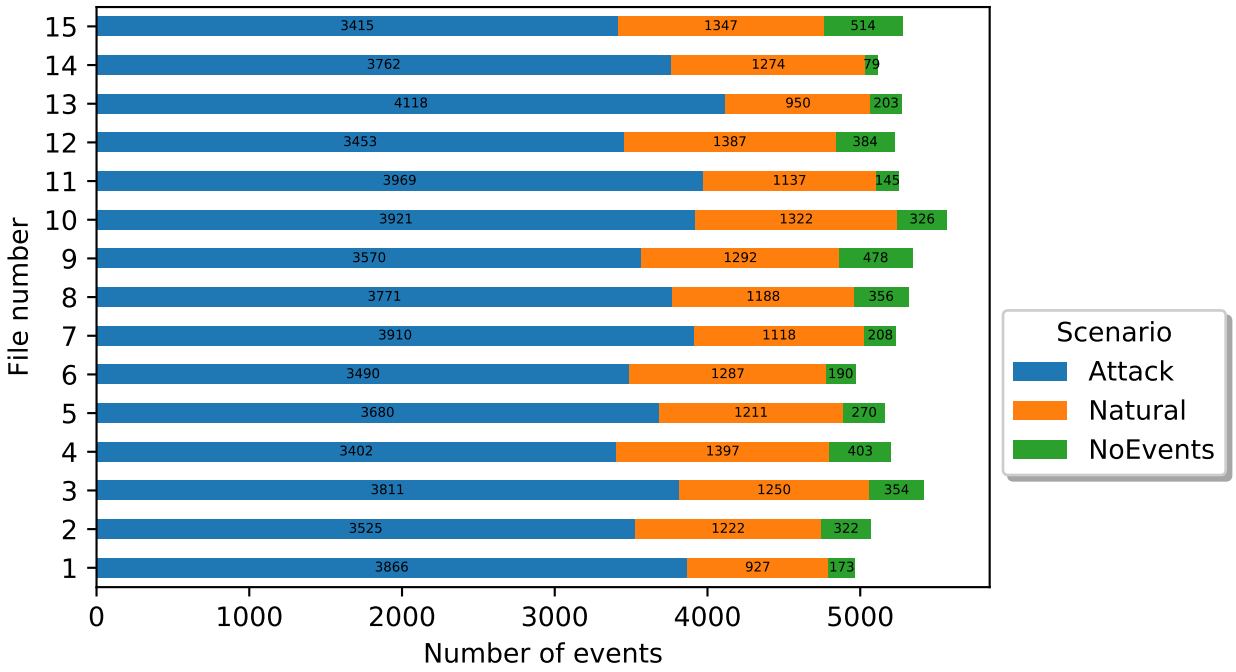


Figure 2-3: Scenarios distribution throughout the 3-class dataset files

Figure 2-3 shows also this distribution for the binary datasets. It is sufficient to add the number of natural (orange) and normal operation (green) events.

The scenarios are not equally distributed in the case of the 37 schemes dataset, it is especially shown by the standard deviation of 61, which is an important value compared to some scenarios

counting less than 100 events. On the other hand, in the case of 3-class scenarios, the distribution is even more not equal compared to the 37 schemes dataset. The **mean standard deviation among all files is equal to 1767**, which is an enormous result given that some scenarios count only around 100 events.

Every electrical grid around the world uses a **3-phased** electric power. Such a grid is composed of three alternating current generators combined. Those generators pass the current in three conductors. That way three conductors are obtained, and each of them conducts a phase of current named A, B and C respectively. The current phases have the same frequency, but a difference of phase of 1/3 of a cycle between each of them. In addition to that, each current has a corresponding voltage, with the same frequency and phases differences [3].

In order to simplify the analysis of three-phase power systems, symmetrical components transformation is used, for both voltage and current. This transformation is defined as:

$$\begin{bmatrix} V_0 \\ V_1 \\ V_2 \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 1 & 1 & 1 \\ 1 & a & a^2 \\ 1 & a^2 & a \end{bmatrix} \begin{bmatrix} V_A \\ V_B \\ V_C \end{bmatrix}, \tag{2.1}$$

where $\mathbf{V_0}$, $\mathbf{V_1}$, $\mathbf{V_2}$ are called respectively **zero sequence**, **positive sequence** and **negative sequence**, $a = e^{i\frac{2\pi}{3}}$ and V_A, V_B, V_C the A-C voltage phases [4]. As each sequence is a weighted sum of sinusoidal functions (A-C phases), it can be on its own written as one sinusoidal function after mathematical transformations.

Each phase is a **sinusoidal** function. Its equation form is $y = A \cdot \sin(\omega t + \theta)$, where A is the amplitude, ω the angular frequency and θ the initial phase. Two terms will be used in what follows: the **magnitude** which is the absolute value of the amplitude and the **angle** which refers to initial phase. These two variables were illustrated on figure 2-4.

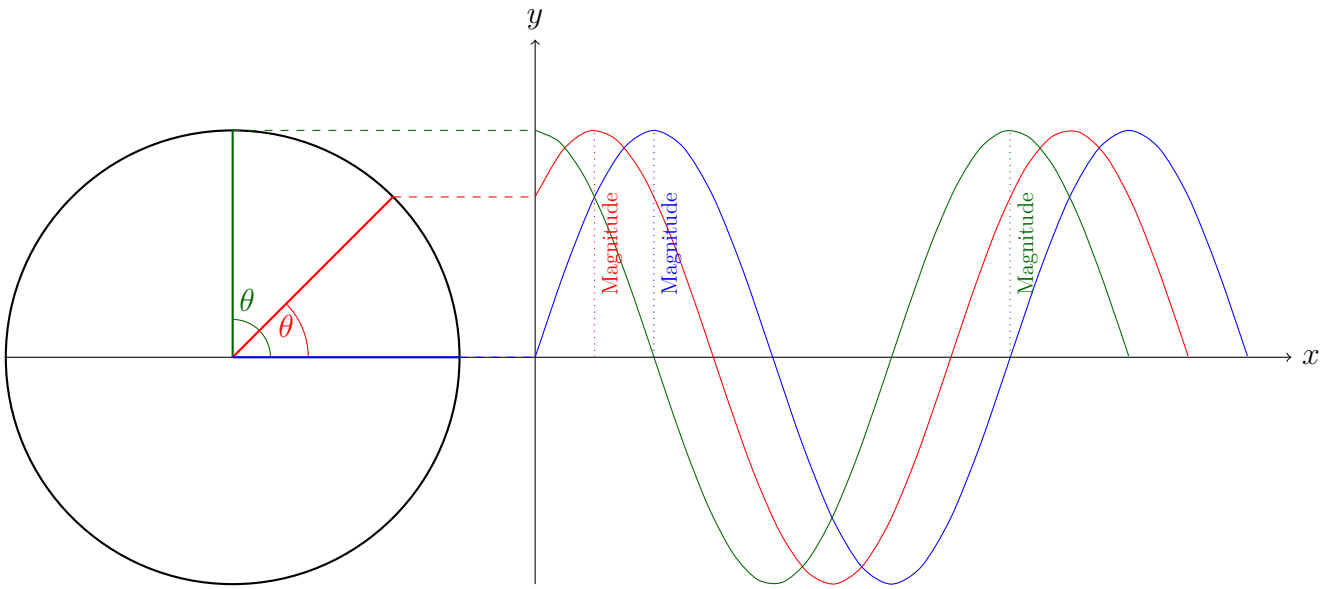


Figure 2-4: Magnitude and angle (θ) of sinusoidal functions

Moreover, when talking about electrical components, the term of impedance can come across. The impedance can be seen as a generalization of resistance of an electrical component. It is defined as the ratio of the voltage over current that passes through this component and takes a complex value of form $z = x + yi$. Every complex number can be also written in form of $z = e^{i\phi}$, where ϕ is the angle of the complex number z, and in this particular case the angle of the impedance.

Every previously mentioned event is described by **128 features**: 116 provided by four IEDs (each one provides 29 types of measurements) and 12 other features are reserved for control panel logs, snort [1] alerts, relay logs of 4 IEDs. The mentioned 116 features, each has a label formed by **concatenation** of the **source IED reference** (it can be R1, R2, R3, R4) and the **measurement name**, as provided in table 2.1. For example R4-PM5:I stands for phase B current phase magnitude measured by R4.

[1]Snort - Network Intrusion Detection and Prevention System

Table 2.1: IED measurements [2]

Feature	Description
PA1:VH – PA3:VH	Phase A-C Voltage Phase Angle
PM1:V – PM3:V	Phase A-C Voltage Phase Magnitude
PA4:IH – PA6:IH	Phase A-C Current Phase Angle
PM4:I – PM6:I	Phase A-C Current Phase Magnitude
PA7:VH – PA9:VH	Pos.–Neg.– Zero Voltage Sequence Angle
PM7:V – PM9:V	Pos.–Neg.–Zero Voltage Sequence Magnitude
PA10:VH - PA12:VH	Pos.–Neg.–Zero Current Sequence Angle
PM10:V - PM12:V	Pos.–Neg.–Zero Current Sequence Magnitude
F	Frequency for relays
DF	Frequency Delta (dF/dt) for relays
PA:Z	Appearance Impedance for relays
PA:ZH	Appearance Impedance Angle for relays
S	Status Flag for relays

Those datasets have been used in several works related to CPS cyber-attack classification, one of which is [5], where the author try to find the most accurate algorithm to predict the status of the power system. The following chapter shows an attempt to partially reproduce the results obtained by them.

Chapter 3

Machine learning algorithms comparison

Before going further and analysing neural networks in order to create a hybrid machine learning algorithm for anomaly detection in the power distribution system presented in chapter 1, a deeper look at classical machine learning algorithms will be taken, in particular Random Forest, Support Vector Machine (SVM), NaïveBayes and multilayer perceptron. However this was done before in [5] using the black-box model algorithms only. In their approach they used Weka [6] in order to find the most performant algorithm among 7 they have chosen (OneR, NNge, Random Forest, Naïve Bayes, SVM, JRipper, Adaboost).

This chapter shows an attempt to reproduce the results provided in [5], using two different machine learning toolkits (Weka and scikit-learn [7]) in order to confirm the obtained results. That is why, first, these two toolkits will be presented, then the obtained results will be discussed.

3.1 Machine learning toolkits

3.1.1 Weka

Weka, or more exactly **W**aikato **E**nvironment for **K**nowledge **A**nalysis, is an open source machine learning software developed at The University of Waikato in Hamilton, New Zealand and based on Java programming language. It is well known especially in academic environments and a lot of machine learning researches were conducted using it, one of them the mentioned before [5]. It incorporates various machine learning tools: classifiers, regressors, visualizers, data pre-processor etc...

Weka is characterised by 3 main operating schemes. First, it can be run using a **graphical user interface** (GUI), enabling the user, even without deep knowledge in programming, to make machine learning experiments and analyse available classifiers. Second, more advanced users have the option

to run all the available tools using a **command line**. Finally, Weka's tools can be **integrated directly into code in several programming languages (Java, Python, R, Spark)**, which enables even larger versatility.

3.1.2 scikit-learn

scikit-learn is an open-source machine learning **Python library** developed originally by David Cournapeau as a Google Summer of Code project and now it is maintained by a team of volunteers. It is well known in both academic and commercial environments, since it is used by many enterprises such as Spotify, Evernote, Booking.com, and research facilities like Inria or Télécom ParisTech [8]. It includes, just like Weka, various machine learning tools such as classifiers, regressors, data preprocessor etc... Its visualisation capabilities are limited, however there exist many additional Python packages for data visualisation such as YellowBrick, Eli5 and others... They will be further discussed in next chapter.

scikit-learn can be used only as an extension of Python language, what makes a bit harder for non experts to start working with it. However, since Python is an user friendly programming language and scikit-learn has a well made documentation, this toolkit is easy to use. Along with other Python packages, and especially pandas, scikit-learn is a very powerful, ergonomic and versatile solution for machine learning problems.

It might be also interesting to mention that scikit-learn can be used within Weka after installing the appropriate add-on. This enables Weka users to use scikit-learn classifiers and regressors and run Python code just inside Weka's GUI.

3.2 Used machine learning algorithms

Before talking about the comparison of machine learning algorithms, the used classifiers are briefly described in this section in order to better understand how do they work. In parallel, values of classifiers' parameters used during all the tests are presented in tables 3.1, 3.2 and 3.3, and, in majority, they represent the default values found in Weka, in order to reproduce the results from [5].

3.2.1 Random Forest

Random Forest, as the name indicates, is an **ensemble** of a given number of trees, and more exactly **decision trees**. This given number of trees is set using the parameter *numIterations* in Weka and *n_estimators* in scikit-learn. Each of decision trees is created based on a randomly chosen set of

features of the dataset. The number of features is given by the parameter *numFeatures* in Weka and *max_features* in scikit-learn, which indicate the way it is calculated - in this case as the logarithm of base 2 of the number of features.

On the other hand, decision trees represent a structure capable of determining the class of the predicted sample. It is composed of nodes connected to each other in a form of tree. Each node corresponds to a condition related to the features, for example if a particular feature is higher than a certain number. Each node, except the decision ones, has two child nodes corresponding to the answer as yes or no to the condition from from parent node. For each of two answers, particular classes are assigned, for example, given the case study in this work, yes answer can correspond to classes Attack and Natural and the answer no to class NoEvents. When running a tree on a particular sample, the nodes are followed given the answers on conditions, until arriving to the decision node. The class corresponding to this node is taken as the final output of the decision tree. The number of layers of nodes is called the depth of the tree and it is set using the parameter *maxDepth* in both Weka and scikit-learn.

The choice of condition in a node is based on calculating **gini impurity** or **information gain** metric for each feature and determining, based on one of them, which feature has the biggest effect on the choice of the class. Gini impurity determines the probability of a feature being classified incorrectly when selected randomly, it is given by the equation:

$$\text{Gini impurity} = 1 - \sum_{i=1}^{n} (p_i)^2, \tag{3.1}$$

where n is the number of classes and p_i the probability of the feature being classified for the class i. The information gain on the other hand, determines the quantity of information that the features gives about a particular class, it is calculated as:

$$E(S) = \sum_{i=1}^{n} -p_i \log_2 p_i. \tag{3.2}$$

Weka uses the information gain metric, while in scikit-learn it is possible to choose the metric using the parameter *criterion*, which by default is set to gini impurity, which is less computationally complex.

The random forest, composed of a number of those decision trees, determines the class of a sample taking the class, which was chosen by the biggest amount of decision trees. In both Weka and scikit-learn, this classifier comes with more, less crucial parameters, which are all listed in table 3.1.

Table 3.1: Random Forest classifier parameters

(a) Weka [9]

Parameter	Value
bagSizePercent	100
batchSize	100
breakTiesRandomly	False
calcOutOfBag	False
computeAttributeImportance	False
debug	False
doNotCHeckCapabilities	False
maxDepth	0
numDecimalPlaces	2
numExecutionSlots	1
numFeatures	0
numIterations	100
outputOutOfBagComplexityStatistics	False
printClassifiers	False
seed	1
storeOutOfBagPredictions	False

(b) scikit-learn [10]

Parameter	Value
n_estimators	100
criterion	"gini"
max_depth	None
min_samples_split	2
min_samples_leaf	1
min_weight_fraction_leaf	0.0
max_features	"log2"
max_leaf_nodes	None
min_impurity_decrease	0.0
min_impurity_split	None
bootstrap	True
oob_score	False
n_jobs	None
random_state	None
verbose	0
warm_start	False
class_weight	None
ccp_alpha	0.0
max_samples	None

3.2.2 SVM

SVM, or more exactly **Support Vector Machine**, is a classifier that creates an N-dimensional space, in which all the samples from the training dataset are put, and then divided, geometrically, into regions, where each region corresponds to one class. The N dimensions of this space corresponds to all the features of the datasets and their transformations (for example a square of one of the features). Those transformations are done using a kernel function which can be set in Weka and scikit-learn using the *kernelType/kernel* parameter. Three kernel types are more commonly used: linear, polynomial and radial basis function (exponent based). The default kernel in both Weka and scikit-learn is radial basis function. Each type of kernel function comes with a set a parameters itself, like the degree in the case of polynomial and all of them can be set through the parameters of both used machine learning toolkits.

In fact, the transformations do not calculate additional dimensions for every sample, but calculate the distance between points in the N-dimensional space. This distance is mathematically defined

as the **dot product** of the two points. The dot product on other hand is defined of the sum of products of coordinates for every dimension.

The division is made by determining so called **hyperplane**, which for example in a 2D features spaces is just a line. The points from each class that are the closest to the hyperplane are called **support vector points**, from where comes the name of this classifier [11] [12].

The algorithm is running within a set number of iterations (can be set within Weka and scikit-learn) dividing the training dataset to smaller tranches using the cross-validation technique. The goal is to find the hyperplane that way so the number of misclassified samples is the smallest in that particular iteration.

SVM in Weka and scikit-learn comes additionally with other parameters than a:re listed on table 3.2.

Table 3.2: SVM classifier parameters

(a) Weka [13]

Parameter	Value
SVMType	C-SVC (classification)
batchSize	100
cacheSize	40.0
coef0	0.0
cost	1.0
debug	False
degree	3
doNotCHeckCapabilities	False
doNotReplaceMissingValues	False
eps	0.001
gamma	0.0
kernelType	radial basis function
loss	0.1
modelFile	Weka-3-8-4
normalize	False
nu	0.5
numDecimalPlaces	2
probabilityEstimates	False
seed	1
shrinking	True
weights	

(b) scikit-learn [14]

Parameter	Value
C	1.0
kernel	"rbf"
degree	3
gamma	"scale"
coef0	0.0
shrinking	True
probability	True
tol	1e-3
cache_size	7000
class_weight	None
verbose	False
max_iter	1000
decision_function_shape	"ovr"
break_ties	False
random_state	None

3.2.3 NaïveBayes

NaïveBayes is a classifier based on the **Bayes theorem**, which states that the probability of event A, given that the event B happened is equal to the product of the probability of B given that A occurred and the probability of A, divided by the probability of B:

$$P(A|B) = \frac{P(B|A)\dot{P}(A)}{P(B)}. \tag{3.3}$$

While the naivety of the algorithm is due to it does not taking into consideration the eventual relations between the features.

The NaïveBayes classifier can be both **multinomial** and **gaussian**. In the multinomial case the probabilities are calculated the classical way (number of occurrences over all samples), whereas in the gaussian case, the probabilities are taken from the gaussian distribution of a given event. For both Weka and scikit-learn the gaussian case were used.

The list of all the parameters available is listed on table 3.3, however they impact on how the algorithm works is marginal.

Table 3.3: NaïveBayes classifier parameters

(a) weka [15]

Parameter	Value
batchSize	100
debug	False
displayModelInOldFormat	False
doNotCHeckCapabilities	False
numDecimalPlaces	2
useKernelEstimator	False
useSupervisedDiscretization	False

(b) scikit-learn [16]

Parameter	Value
priors	None
var_smoothing	1e-9

3.2.4 Multilayer perceptron

The multilayer perceptron is a classifier inspired by the structure of the human brain. It is constituted from a set of interconnected neurons, which are divided into at least 2 layers: input layer, n hidden layers and output layer. Each layer consist of a number of neurons. The input layer has always a number of neurons equal to the number of features, and the output layer has always this number equal to the number of outputs of the problem. That is why, in the analysed case the input layer has 128 neurons and the output layer only one neuron corresponding to the status of the power plant (normal behaviour, natural fault, attack). Each of the mentioned neurons has a weight. The

neurons are interconnected. The output of a neuron is considered as the input for the neurons of the following layer. Those neurons of the following layer calculates the weighted sum of the inputs and passes the results through a function, called activation function. The result obtained this way is considered as the output of the given neuron and goes to the following layer. The only exception is the input layer, which input is the values of features and its neurons just copy the input to the output [17] [18] [19].

The training process consists in updating the weights of the neurons during a predefined number of iterations called epochs. Before starting the training process the initial weights w_i values for hidden and output layers should be chosen. This choice can be done empirically or randomly. Moreover, a real positive number η called training step must be fixed. In each epoch and for each training pattern, the weights are updated following the equation:

$$w_i = w_i + \eta \cdot (z^{(\mu)} - y) \cdot x_i^{(\mu)}, \tag{3.4}$$

where $z^{(\mu)}$ is the desired output for the μ-th training sample, y is the neuron output (weighted sum passed through the activation function), $x_i^{(\mu)}$ is the neuron's input value for the same μ-th training sample and i indicates the neuron index in the trained layer. After each epoch, the training samples are shuffled [19].

The scikit-learn implementation of multilayer perceptron classifier is limited because it does not give a full control of each hidden layer and its activation function. It integrates however the possibility to determine the number of epochs called iterations and the training step called learning rate. All available parameters available are shown in table 3.4. The implementation of MLP exists also in Weka, but it is as well limited, for this reason the analysis of its results will not be taken into consideration.

There exists multiple python frameworks offering much more flexibility when creating neural networks like PyTorch or Keras framework, which were created in order to offer the possibility to create complex neural networks able to be used in large-scale applications. Even the developers of scikit-learn state on their website [18]:

> *This implementation is not intended for large-scale applications. In particular, scikit-learn offers no GPU support.*

The mentioned frameworks will not be used in this work because this flexibility is not required in this case. However, it does not mean that it is not possible to make the results even better using those frameworks.

Table 3.4: MLP classifier parameters in scikit-learn [20]

Parameter	Value
hidden_layer_sizes	(20,)
activation	'relu'
solver	'adam'
alpha	0.0001
batch_size	'auto'
learning_rate	'constant'
learning_rate_init	0.001
power_t	0.5
max_iter	1000
shuffle	True
random_state	None
tol	$1e-4$
verbose	False
warm_start	False
momentum	0.9
nesterovs_momentum	True
early_stopping	True
validation_fraction	0.1
beta_1	0.9
beta_2	0.999
epsilon	$1e-8$
n_iter_no_change	10
max_fun	15000

3.3 Metrics for classifiers comparison

All those classifiers give predictions that can be divided into 4 main groups. Assuming for a moment, for illustration purposes, that normal fault class is positive and attack class is negative, it could be differentiated between:

- true positive predictions (**tp**): correct prediction of the normal fault class,
- true negative predictions (**tn**): correct prediction of the attack class,
- false positive predictions (**fp**): incorrect prediction of normal fault class,
- false negative predictions (**fn**): incorrect prediction of attack class.

For that two classes, several metrics can be defined, that will be useful to compare between the used classifiers:

- **accuracy**: ratio of correct classifications over the total number of samples, or in other words:

$$\text{accuracy} = \frac{tp + tn}{tp + tn + fp + fn} \tag{3.5}$$

- **precision**: ratio of correct classifications for a particular class over all classifications that indicated that class, or:

$$\text{precision} = \frac{tp}{tp + fp} \tag{3.6}$$

- **recall**: ratio of correct classifications for a particular class, over all samples corresponding for this class, or:

$$\text{recall} = \frac{tp}{tp + fn} \tag{3.7}$$

- **f-measure**: weighted average of precision and recall given by the equation:

$$\text{f-measure} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}.$$

Given that those metrics work only with binary problems with a positive and a negative class, for multiclass problems the **accuracy is always calculated as the ratio of true classifications over all classifications**, however for precision, recall and f-measure, the average is calculated, and that in 3 different ways [21]:

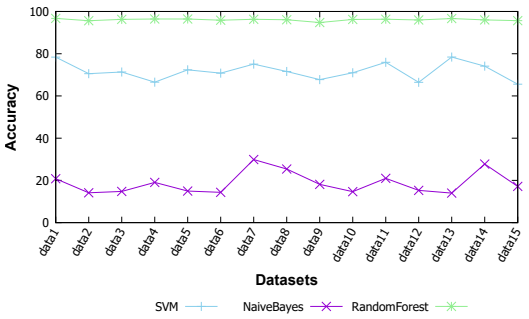
- **micro** average: the metrics are determined globally by calculating true positives as all correct predictions, false negatives and false positives as all incorrect predictions (assuming two classes A and B, when A is misclassified as B is a false positive for A, and in the same time a false negative for B). In this case precision, recall, f-measure and accuracy have exactly the same value,
- **macro** average: the metrics are calculated for each class, the concerned class is considered as positive, while the sum of others as negative. Then their arithmetic mean value is calculated,
- **weighted** average: the metrics are calculated for each class, then it calculates their weighted average value by the number of true instances for each class.

scikit-learn supports the calculation of precision, recall and f-measure in the 3 different ways explicitly, however in Weka only for f-measure all 3 are available, while for precision and recall

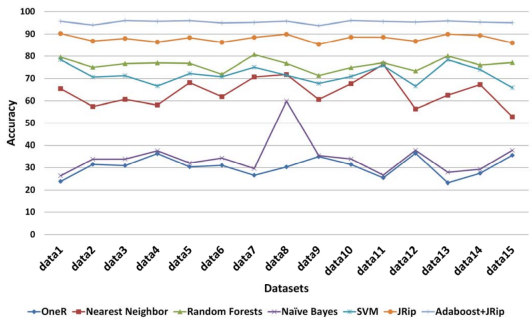
there is the choice between the generic ones and the weighted averages. The generic ones the most probably correspond to the macro average, since, after some tests, the obtained values for those metrics are not the same. The exact information about the nature of those averages could not be found in Weka's documentation.

3.4 Comparison results

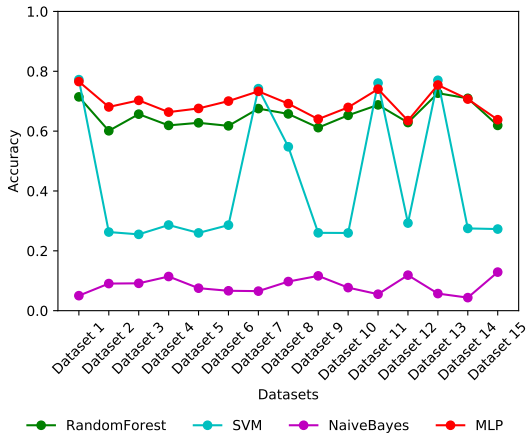
In order to reproduce the results presented in [5], Weka version 3.8.4 and scikit-learn version 0.23.1, running on Anaconda 3.18.11 with Python 3.7.6.final.0, were used. SVM in Weka comes from a package untitled *libsvm* available for this toolkit. The results are displayed in the same order as in [5], thus, the accuracy was presented on figures 3-1, 3-2 and 3-3 for the three class, binary and all 37 class cases respectively. Then, the precision, recall and f-measure can be found on figures 3-4, 3-5 and 3-6. For f-measure the scikit-learn plot is not displayed because Weka differentiates between micro, macro and weighted averages and a generic value for f-measure. The generic value does not correspond by value to the other averages.



(a) Our attempt - Weka

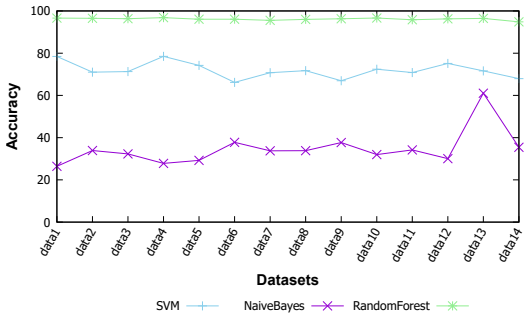


(b) Original results [5]

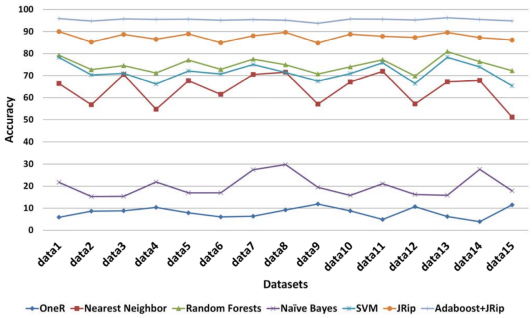


(c) Our attempt - scikit-learn

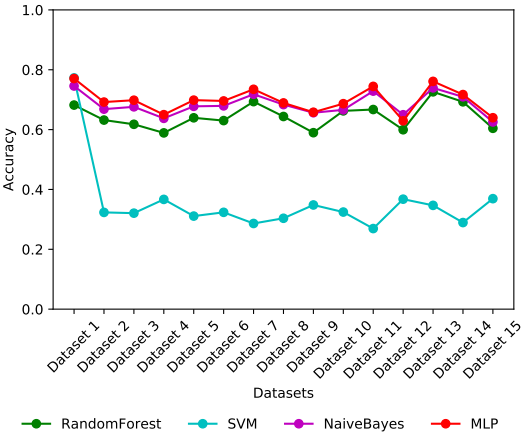
Figure 3-1: Accuracy for three-class datasets



(a) Our attempt - Weka

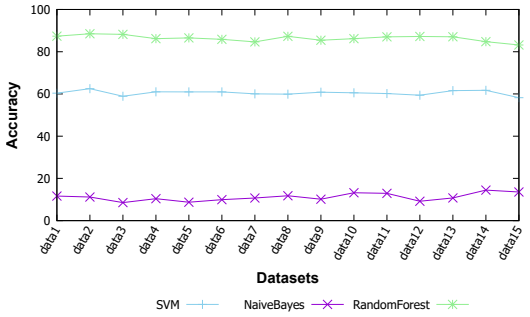


(b) Original results [5]

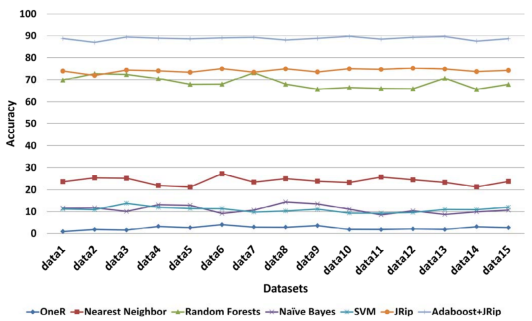


(c) Our attempt - scikit-learn

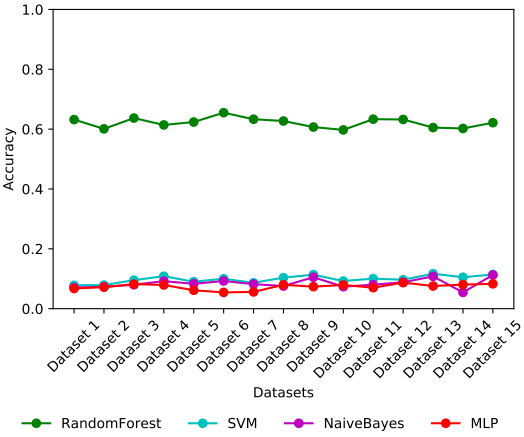
Figure 3-2: Accuracy for binary datasets



(a) Our attempt - Weka

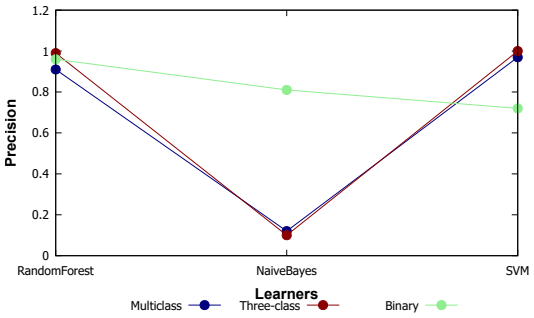


(b) Original results [5]

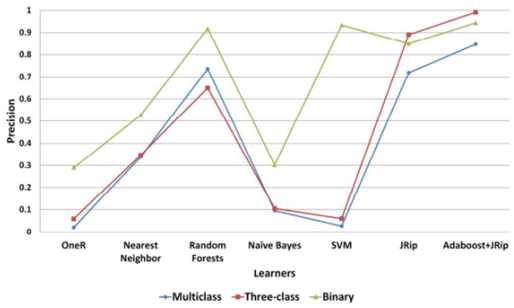


(c) Our attempt - scikit-learn

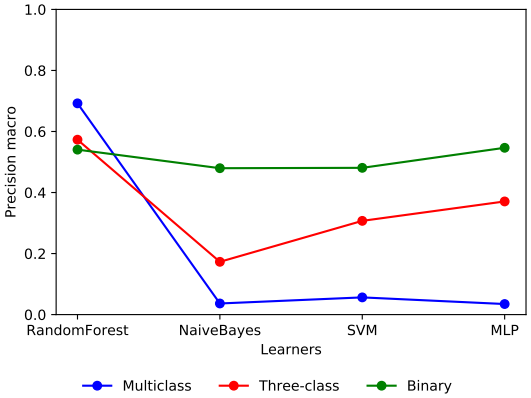
Figure 3-3: Accuracy for multiclass datasets



(a) Our attempt - Weka

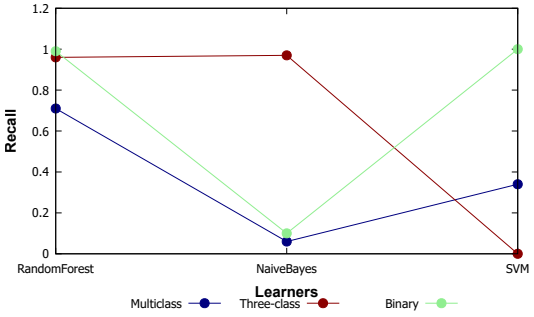


(b) Original results [5]

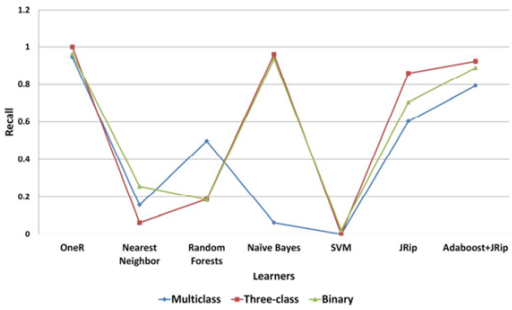


(c) Our attempt - scikit-learn

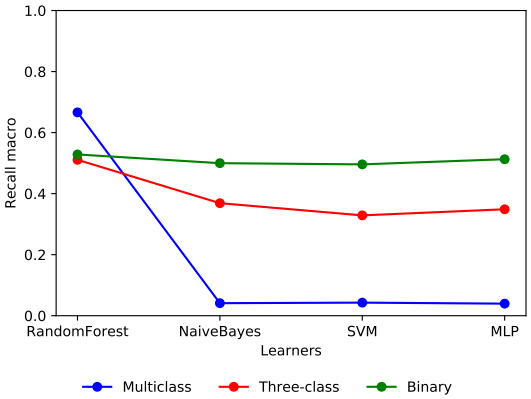
Figure 3-4: Precision



(a) Our attempt - Weka

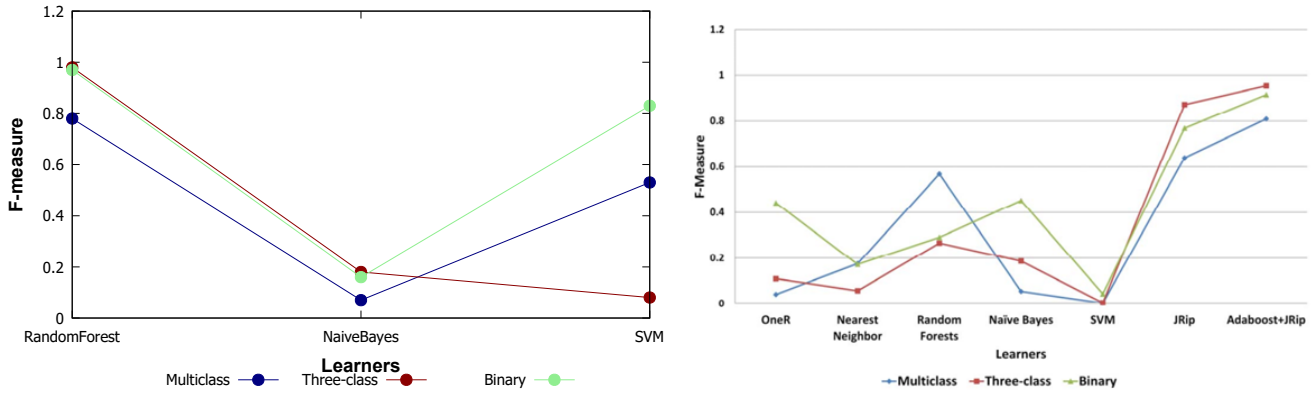


(b) Original results [5]



(c) Our attempt - scikit-learn

Figure 3-5: Recall



(a) Our attempt - Weka

(b) Original results [5]

Figure 3-6: F-measure

The accuracies in the three-class case are quite similar on the three plots, except SVM in scikit-learn, which show a huge discrepancy of accuracy between datasets. Moreover, a remarkably higher values of accuracy for Random Forest classifier are observed in the attempt using Weka, compared to other plots.

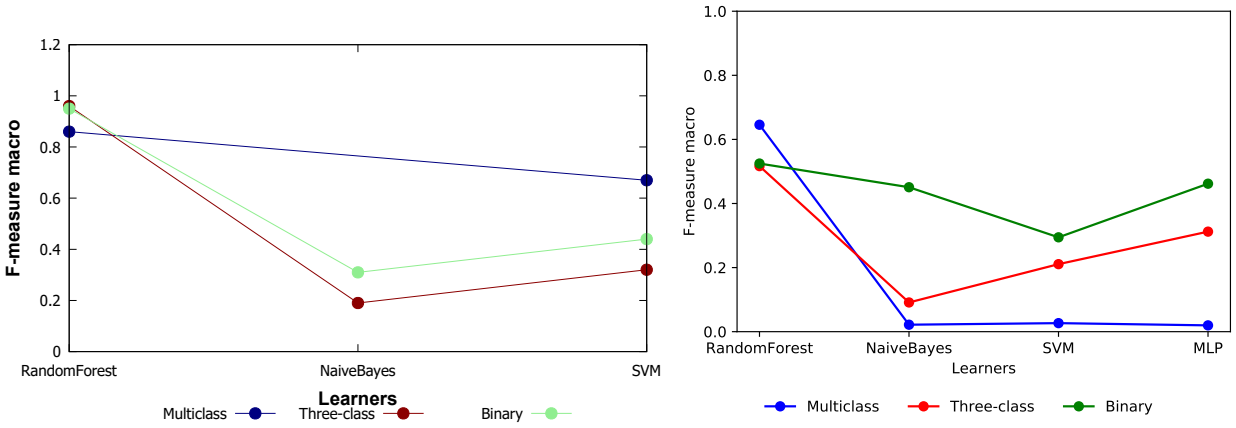
In the binary and multiclass cases, the Random Forest classifier in the attempts using Weka always shows higher values compared to other plots. Scikit-learn, on other hand, for Random Forest classifier, gives results similar to those made by [5]. For SVM classifier, the variance this time, in the plots made by scikit-learn, is remarkably less important and the values oscillate around a constant value. For the binary case, this value is about 0.3, while in Weka the obtained value is 0.75 (comparable to the original results in [5]). On the other hand, in the multiclass case, it oscillates around 0.1, like it is the case in the results from [5]. In the attempt made in Wake the accuracy is much higher at about 0.6. For the NaïveBayes classifier the results are comparable, except the results obtained in scikit-learn in the binary case.

Concerning the other metrics, Random Forest classifier shows values close to 1 in both Weka and scikit-learn, while the original results for recall and f-measure show small values. For the rest of classifiers the results are generally totally different between attempts, with some exceptions (like SVM recall with Weka in three-class case compared to [5], or also SVM recall, but with scikit-learn and in multiclass case compared to [5]).

It can be concluded from here, that the results from [5] were successfully reproduced in Weka in the three-class and binary cases, however for multiclass the reproduction of the results failed. Scikit-learn, on other hand, gives different results than Weka despite using the same algorithms with the same set of parameters. The more efficient classifier is certainly Random Classifier, especially when using Weka. However, MLP classifier gives also quite good results.

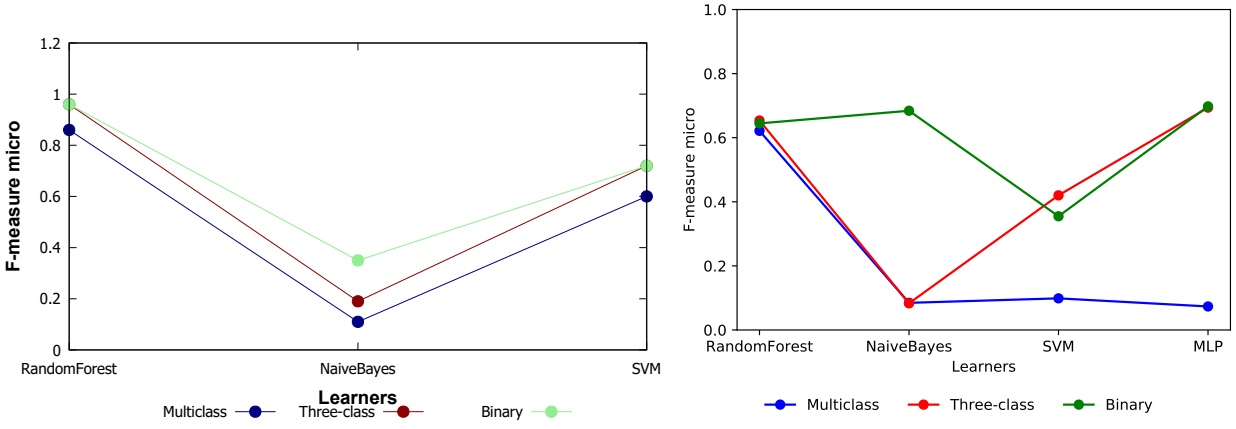
Moreover to get more information about the behaviour of the classifiers the f-measure macro,

micro and weighted averages were calculated, along with precision and recall weighted averages. The results are shown on figures 3-7, 3-8, 3-9, 3-10 and 3-11 respectively. The values of f-measure micro, weighted and precision weighted for NaïveBayes classifier for multiclass case could not be calculated for unknown reason. These metrics were not presented in [5].



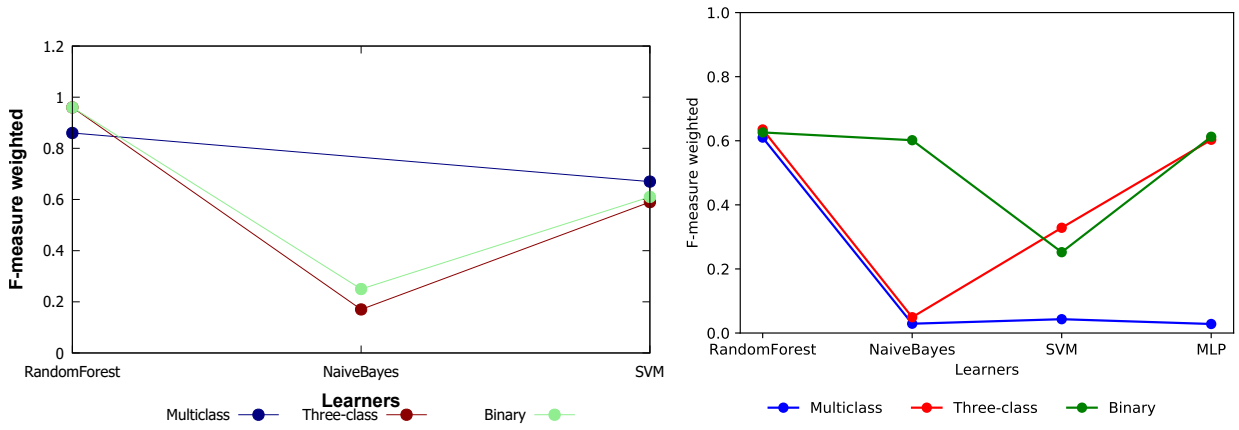
(a) Weka (b) scikit-learn

Figure 3-7: F-measure macro



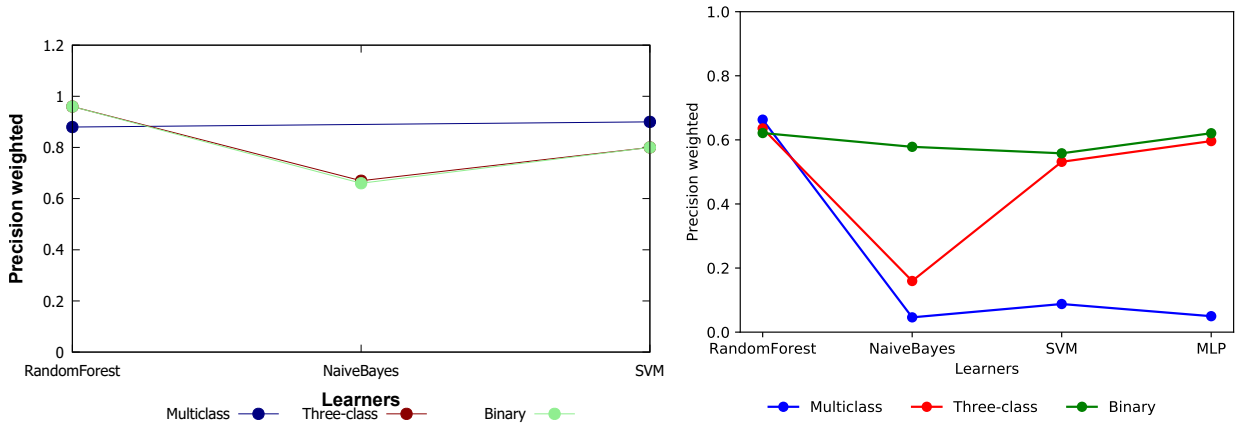
(a) Weka (b) scikit-learn

Figure 3-8: F-measure micro



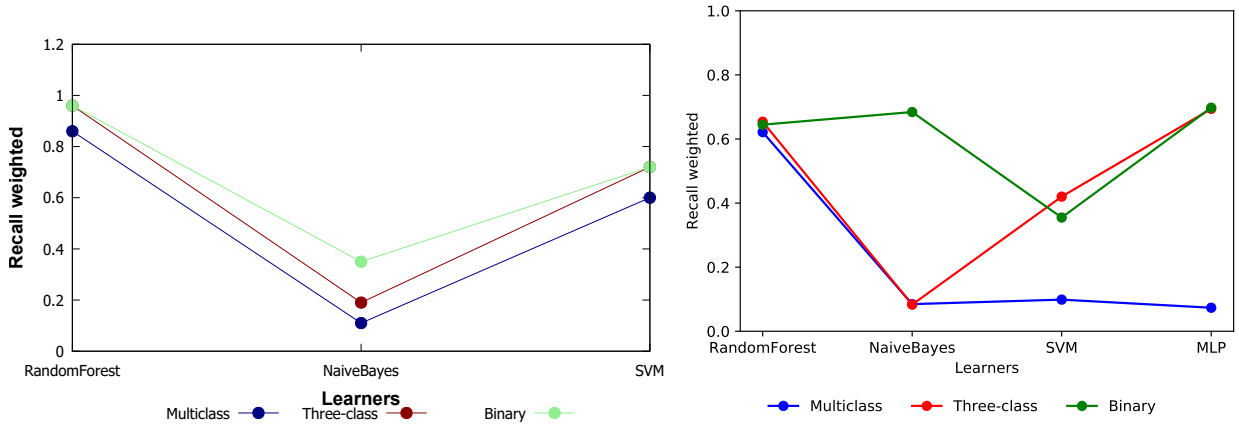
(a) Weka (b) scikit-learn

Figure 3-9: F-measure weighted



(a) Weka (b) scikit-learn

Figure 3-10: Precision weighted



(a) Weka (b) scikit-learn

Figure 3-11: Recall weighted

It would be hard to say that the obtained results in Weka and scikit-learn are similar, except the three-class case, where the results are quite similar. These results confirm once more time that

Weka and scikit-learn tend to produce different results and that the best classifier is Random Forest, followed by MLP, especially in binary and three-class cases.

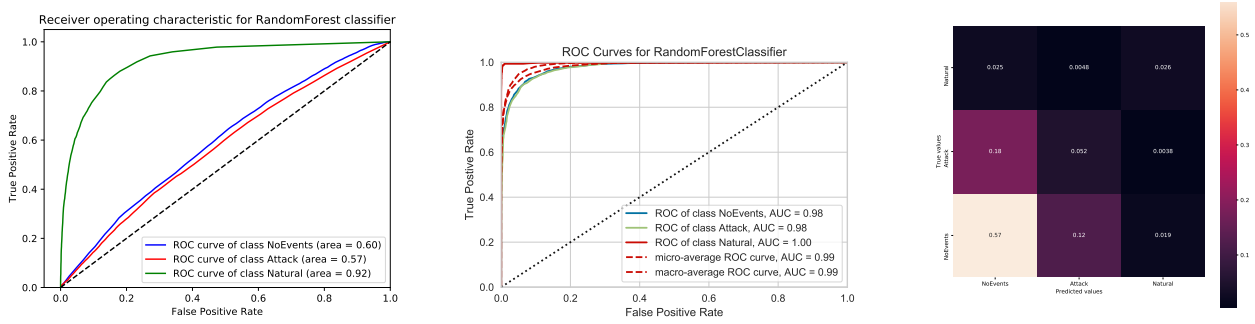
Since the results for three-class and binary cases were comparable and the three-class case gives more information about the status of the system, in what follows, the focus will be on the three-class case.

3.5 scikit-learn further methods' analysis

In addition to all that, scikit-learn enables the user to plot the receiver operating characteristic (ROC) curves for each class and the confusion matrix. The ROC curve represents the plot od true positive rate when the false positive rate changes. It is based on the prediction probabilities for each class given for each tested sample. Those predictions can be provided in two different ways:

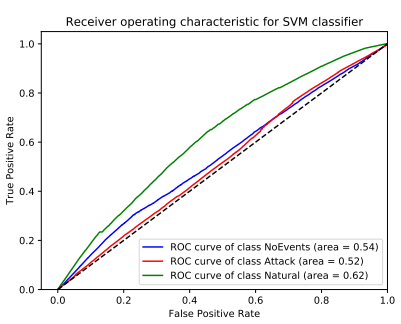
- concatenation of these probabilities for each cross-validation iteration,
- determination of probabilities for preselected test dataset values.

The confusion matrix on the other hand shows the normalized number (over the total number of samples) of predicted values of each class for each class. The results are illustrated on figures 3-12, 3-13, 3-14 and 3-15.

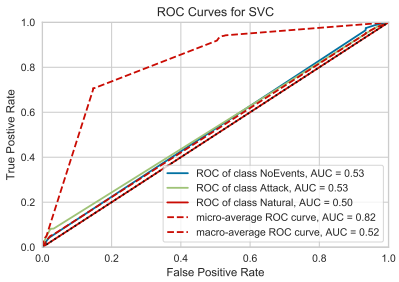


(a) ROC Curve - cross-validation (b) ROC Curve (c) Confusion Matrix

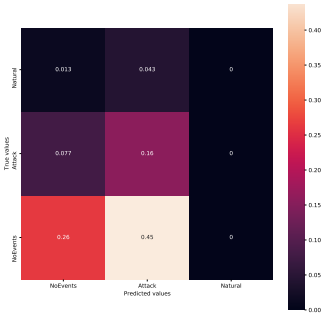
Figure 3-12: Random Forest ROC curve and confusion matrix



(a) ROC Curve - cross-validation

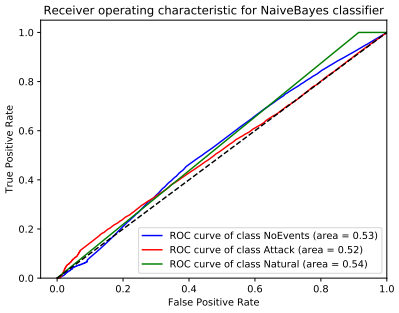


(b) ROC Curve

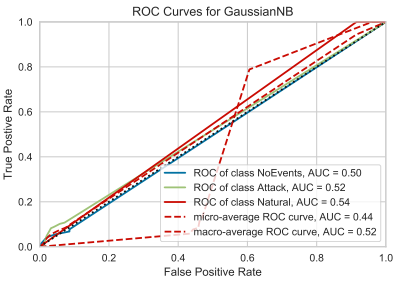


(c) Confusion Matrix

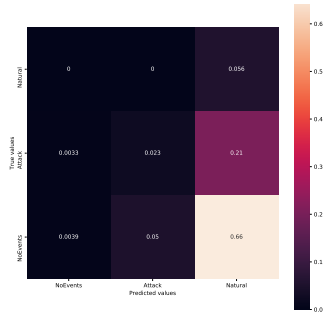
Figure 3-13: SVM ROC curve and confusion matrix



(a) ROC Curve - cross-validation

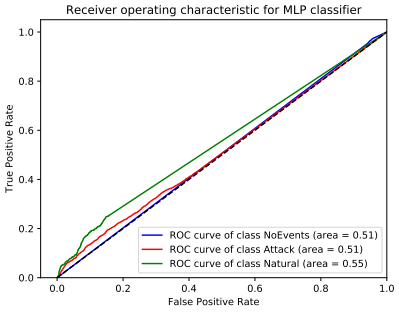


(b) ROC Curve

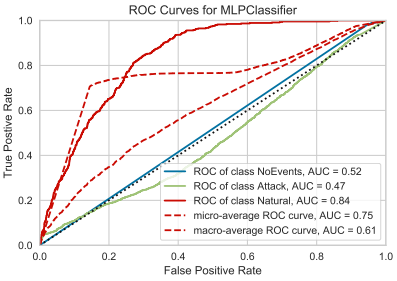


(c) Confusion Matrix

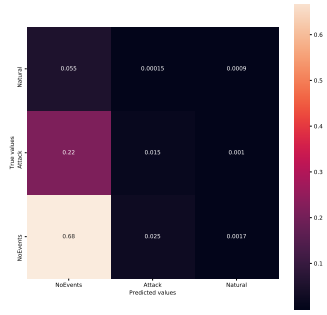
Figure 3-14: Naïve Bayes ROC curve and confusion matrix



(a) ROC Curve - cross-validation



(b) ROC Curve



(c) Confusion Matrix

Figure 3-15: MLP ROC curve and confusion matrix

The previous figures show that Random Forest classifier has the higher capacities to distinguish between the occurrence of each class, or it's absence. It's visible on both ROC curves and the confusion matrix, where the highest number of predictions is shown for the true positives for each class. SVM tends to predict only NoEvents and Attacks but does not really succeed in distinguishing between them. Naïve Bayes fails to make true predictions, it considers everything of class natural. Finally MLP, it succeeds in determining the class NoEvents, but does not distinguish over classes almost at all, despite the high accuracy (it is due because of the huge number of samples of class NoEvents).

The two methods of plotting the ROC curve does not give exactly the same results, however the conclusions remain the same in both cases. This fact may be due to a slightly different set of test data used for its creation.

Given this analysis, it can be deducted that Random Forest algorithm acts the best, and that is why it will be adapted in next chapters, in which, at first, an analysis of features and their importance will be made. However a deeper look at the amelioration of MLP, in order to succeed in differentiation between the classes, will be also made later.

Chapter 4

Features' importance

After having determined the most performant algorithm, which is Random Forest, it is time to go further and analyse which features impact the results of classification the most. The focus will be especially on false predictions. In order to do that, six tools will be compared: LIME [22], ELI5 [23], YellowBrick [24], Treeinterpreter [25], dtreeviz [26] and export_graphviz tool from scikit-learn, where the last three ones are designed for Decision Tree and Random Forest classifiers.

4.1 Result's interpreters' comparison

4.1.1 LIME

LIME (Local Interpretable Model-agnostic Explanations) is a tool that is used to explain the behaviour of machine learning classifiers. It supports, as for this day, only the explanation of individual predictions for any scikit-learn classifier or regressor. This explanation consist in a list of features (and more precisely decision nodes composed of feature name and a numeric condition) ordered by their relative importance for a particular prediction. This list can be shown is a raw mode (as a python list) or in a visual form (pyplot figure, jupyter notebook or html file).

In order to class the features according to their importance, LIME approximates the model by an interpretable one, created based on perturbing the features of the examined instance. More the perturbed instances are similar to the examined instance, higher is the weight of the perturbed feature. The interpretable model can be This approximation process is called explanation and is described using the following equation:

$$\xi(x) = \underset{g \in G}{\operatorname{argmin}} \mathcal{L}(f, g, \pi_x) + \Omega(g) \quad [22], \tag{4.1}$$

where g is the explanation model from the set of interpretable models G, f is the probability that the

sample x corresponds to a certain class, π_x is the distance between an instance z and the sample x. $\mathcal{L}(f, g, \pi_x)$ describes to which extent g is unfaithful in the approximation of f in the neighbourhood defined by π_x, while $\Omega(g)$ describes the complexity of the explanation (for example the depth of the tree if f is a decision tree). The goal of the explanation is to find the most faithful approximation of f taking into consideration the neighbourhood π_x.

To better illustrate the algorithm, let the pigeon image on figure 4-1 be considered as a sample used in a classification problem based on a classifier noted *f*. The image can be divided into several regions, like it is shown on figure 4-2, each region corresponds to a feature.

Figure 4-1: Pigeon

Figure 4-2: Image grid

In the next step, a new dataset is created by modifying the features. In the case of the pigeon image some regions can be simply greyed out like it is shown on figure 4-3. For each of these the probability if it represents a pigeon or not is calculated using the classifier *f*.

(a) Perturbation 1

(b) Perturbation 2

(c) Perturbation 3

Figure 4-3: Greyed-out images forming a new dataset

Using this data it is possible to determine the importance of regions/features of this particular sample. This importance is local and has a meaning only in relation to the sample image from figure 4-1.

4.1.2 ELI5

ELI5 (Explain like I am a 5-year old) is a tool, in form of a Python package, used to debug machine learning classifiers and explain their predictions. It supports multiple machine learning frameworks,

including scikit-learn. It can be used to explain how the model works both locally for one prediction and globally for the whole model. The output can take several forms just like in LIME case.

For white-box models, ELI5 works as an extension of scikit-learn and it's capable to extract the weights of model's features for different classes. In addition to that it can show the weights that contributed in a particular prediction. It is done by removing particular features and checking how the accuracy of the classifier was altered. On the other hand, for black-box models, this tool integrates a modified version of LIME, supporting more machine learning frameworks, and a permutation importance method, which checks how the model's accuracy decreases when removing one of the features and on this basis determines the importance of the features.

4.1.3 YellowBrick

YellowBrick is another Python package, which is an extension of scikit-learn framework. It is meant to give global interpretation of the analysed model on different levels. It is possible not only to visualize features importances calculated directly by scikit-learn, but also to give a classification report (accuracy, recall, precision, f-measure), plot a confusion matrix, a ROC curve and much more. In addition to all that, YellowBrick comes with a tool for determination of correlation between features in the dataset.

4.1.4 Treeinterpreter

Treeinterpreter is a simple Python package that works with scikit-learn trees and random forest classifiers. It's only usage consists in decomposing the obtained prediction into bias and contributions of different features. The output is given in the form of a numpy array.

4.1.5 scikit-learn export_graphviz

export_graphviz is a scikit-learn embedded function that enables the user to visualise a decision tree with all the branches and save it into Graphviz[1] format, that can be converted into a vector graphic. It is possible also to visualise decision trees composing random forest model in scikit-learn, since the possibility to extract particular decision trees when using this framework. However the interpretability of results for random forest classifier can be hard.

[1] a set of tools for diagram creation using graphs.

4.1.6 dtreeviz

dtreeviz is a more advanced version of export_graphviz available in scikit-learn. For every leaf in the tree it can show a histogram indicating the influence of feature value on class selection. In addition to that, dtreeviz enables the user to show the path of a particular prediction. The result is saved in the form of a svg vector graphic.

4.1.7 Summary

It can be concluded that ELI5 is the most versatile package compared to others, especially because it enables both global and local interpretations and does not limit its support to scikit-learn, plus it has LIME integrated in it. YellowBrick, on other hand, adds the possibility to analyse from a statistical view the features available in the dataset. Finally comes Treeinterpreter and dtreeviz that are interesting tools when analysing especially Decision Trees. A summary of the most import features of all 5 packages is shown in table 4.1, where 6 comparison metrics where taken into consideration:

- global interpretation: capacity of the tool to interpret the whole model,
- local interpretation: capacity of the tool to interpret a particular sample from the dataset,
- black-box models support: the fact if the tool supports only black-box models (models that can not be simply interpreted),
- features' statistical analysis: the fact if the tool supports statistical analysis of features in the dataset, without taking into consideration the model,
- works only with scikit-learn,
- decision trees graphical visualisation.

Table 4.1: Comparison of tools for model analysis

	LIME	ELI5	YellowBrick	Treeinterpreter	dtreeviz
Global interpretation	✗	✓	✓	✗	✗
Local interpretation	✓	✓	✗	✓	✓
Black-box models support	✓	✓	✓	✗	✗
Features' statistical analysis	✗	✗	✓	✗	✗
Works only with scikit-learn	✓	✗	✓	✓	✓
Decision Trees graphical visualisation	✗	✗	✗	✗	✓

4.2 Features' importance determination

For the rest of the chapter LIME was chosen to determine the features' importance because of it working with all black-box models available in scikit-learn. The capabilities of LIME were sufficient and that is why ELI5 was not used in his place.

In what follows the Decision Tree classifier will be used because its ease of interpretation. The explication of the algorithm was provided in 3.2.1 as the part of Random Forest classifier algorithm, since the Random Forest is composed of a certain number of Decision Trees.

Because in this case the explanations of single samples are not really interesting, an attempt to generalize the results was made: Lime explainer was run on 100 false predictions of a chosen class made using Decision Tree classifier. The results are concatenated together. For all the duplicates of features, the mean importance is calculated and only one entry is kept along with this calculated mean value. In other words:

```
1  import pandas as pd
2  import numpy as np
3  from lime.lime_tabular import LimeTabularExplainer
4
5  # explainer initialization with training data
6  explainer = LimeTabularExplainer(X, training_labels = y,
       feature_names = features, class_names = labels)
7
8  # Xall = all samples, yall = all true classes corresponding to
       samples (not used in training)
9  boole = (yall != clf.predict(Xall)) # all false predicitions
10 faulty = boole & (yall == 0) # false predicitions of true class
       equal 0 (NoEvents)
11 X_test = Xall[faulty] # all false predicition samples
12 y_test = yall[faulty] # all false predicition true classes
13 lst = [] # initialization of a list
14 for idx in range(0, 100): # create explanation for 100 first samples
15     exp = explainer.explain_instance(X_test[idx], clf.predict_proba,
           num_features=128, labels=[0, 1, 2])
16     lst.append(exp.as_list(label=0))
17 # transform to dataframe
```

```
lst = np.array(lst)
clst = np.concatenate(lst, axis=0)
dtfr = pd.DataFrame(clst, columns=['feature', 'importance'])
dtfr["importance"] = pd.to_numeric(dtfr["importance"])
# group by feature then sort
dtfr = dtfr.groupby(['feature']).mean()
res = dtfr.sort_values(by="importance")
```

The results, reduced to 5 most important features for NoEvents, Attack, Natural and all samples, are shown respectively in tables 4.2, 4.3 and 4.4.

Table 4.2: 5 most important features for NoEvents samples

feature	importance
R4-PA5:IH > 115.38	0.008534
R3-PM2:V <= 128425.29	0.007998
128762.21 < R2-PM1:V <= 129859.49	0.007776
0.00 < R1-PA12:IH <= 32.04	0.007514
R3-PM5:I <= 330.70	0.005762
...	...

Table 4.3: 5 most important features for Attack samples

feature	importance
R3:S > 0.00	0.010717
R2-PA7:VH <= -101.20	0.014018
R2-PM1:V > 130872.03	0.009242
R3-PA7:VH <= -101.22	0.008676
R3-PA2:VH <= -93.75	0.008307
...	...

Table 4.4: 5 most important features for Natural samples

feature	importance
R2:F > 60.00	0.005168
R3:F > 60.00	0.004873
R2-PA5:IH > 63.30	0.004103
R2-PM7:V > 130857.40	0.003693
R1-PA1:VH > 71.28	0.003518
...	...

It can be observed that the five most important features among false predictions for each class are different. This implies creating a correction model for every important feature among all classes. The exact procedure is described in the next chapter.

In this chapter different result interpreters were presented in details. Among them, Lime have been chosen to create an algorithm for features importance determination. The most important features will be used in next chapter to correct the obtained predictions.

Chapter 5

Model enhancement

After having chosen the appropriate machine learning algorithms and found a way to determine the most important features in the classification problem, this chapter is an attempt to create a model able to enhance the results obtained by the basic classification. Three different approaches are presented. First of all, the most important features values were altered and the effect of this modification is examined. Second, a formula for calculating the distance between different samples is established then a way to use that information. Finally, the Hidden Markov Models were used in order to determine the likeliness of prediction of a particular class and this data was used to modify the machine learning algorithm.

5.1 Features values modification

The first proposed solution considers changing the values of most important features in the dataset after the training process. In other words the training process occurs normally, then the importances are determined and a correction function is created (figure 5-1). This correction function will act then on the samples introduced to the model in order to change the features values and obtain better predictions (figure 5-2).

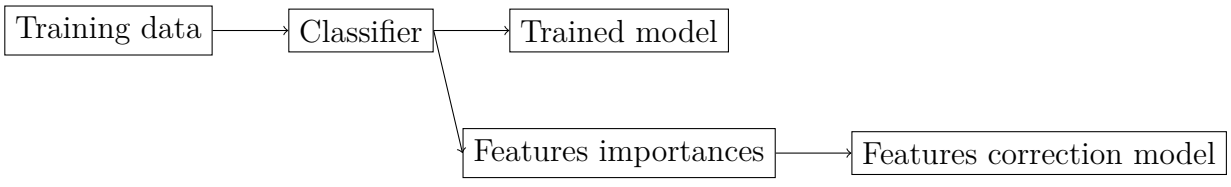


Figure 5-1: Training process illustration

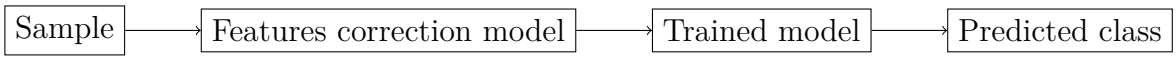


Figure 5-2: Prediction illustration

First of all the correction function was defined as the modification function of the feature values for the predicted samples. This modification consists in shifting the feature value so the feature value would not meet the condition to make a false prediction from tables 4.2-4.4. This function python code is as follows:

```
1 def modify(feat, val):
2     X[feat] = X[feat].apply(lambda x: x + val)
```

where X is a pandas DataFrame object containing all the samples to predict.

Second, the five most important features are taken from tables 4.2-4.4 and the values of features from the samples to predict are altered using the previous function. The following code shows this operation:

```
1  modify("R4-PA5:IH", -115.38)
2  modify("R3-PM2:V", 128525.29)
3  modify("R2-PM1:V", 2000)
4  modify("R1-PA12:IH", 32.04)
5  modify("R3-PM5:I", 330.7)
6  modify("R3:S", 0)
7  modify("R2-PA7:VH", 101.20)
8  modify("R2-PM1:V", -1300872.03)
9  modify("R3-PA7:VH", 101.22)
10 modify("R3-PA2:VH", 93.75)
11 modify("R2:F", -60)
12 modify("R3:F", -60)
13 modify("R2-PA5:IH",- 63.30)
14 modify("R2-PM7:V", -130857.40)
```

The samples modified this way are then used for the predictions. In order to check the success of this method, the classification_report method from scikit-learn was used. The results before and after modifying the samples are displayed in tables 5.1 and 5.2.

Table 5.1: Classification report before features modification

	precision	recall	f1-score	support
NoEvents	0.72	0.76	0.74	51797
Attack	0.27	0.26	0.26	17382
Natural	0.19	0.08	0.12	4232
accuracy			0.60	73411
macro avg	0.39	0.37	0.37	73411
weighted avg	0.58	0.60	0.59	73411

Table 5.2: Classification report after features modification

	precision	recall	f1-score	support
NoEvents	0.71	0.83	0.77	51797
Attack	0.26	0.17	0.21	17382
Natural	0.10	0.03	0.05	4232
accuracy			0.63	73411
macro avg	0.36	0.34	0.34	73411
weighted avg	0.57	0.63	0.59	73411

The tables 5.1 and 5.2 show a general increase of the accuracy of the model after changing the features. The more detailed results, show a remarkably better recall and f-measure for NoEvents class, with a small decrease of precision, but for all other classes a decrease can be observed for all the metrics.

For this particular set of samples, there is a slight increase of weighted recall, weighted f-measure and accuracy. That it is why, it may be concluded that this method succeeded with this particular dataset, especially given the unequal distribution of the samples between classes and model's tendency to predict NoEvents class. However, with any other dataset, where the NoEvents class samples are less common compared to other classes, this method causes worse predictions.

5.2 Distance between features

The second proposed solution considers using a transformation routine, which, reduces the number of features to the 15 most important from tables 4.2-4.4 and adds another feature that represents the closest class to the treated sample. The prediction steps were illustrated on figure 5-3.

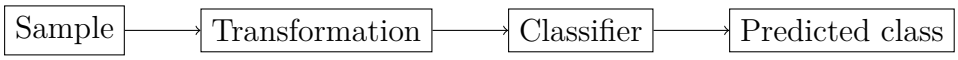


Figure 5-3: Distance algorithm illustration

The transformation routine on other hand takes the form of a python class and is composed of 4 methods:

1. **distance(X1, X2)**: returns the distance between two samples X1 and X2. It is calculated as the sum of differences between features,

2. **important(X)**: returns the samples with only 15 most important features. The input must be a pandas DataFrame or Series. The choice of features to keep is made by hand directly in the method, without the possibility to change them afterwards,

3. **fit(X,y)**: determines the reference class sample for each class by calculating the mean value of each feature among all samples corresponding to the treated class. It is called only during data fitting to the classifier,

4. **transform(X)**: determines the class with the smallest distance to the sample, based on reference samples determined by fit(X,y) method. The obtained values are then added as a new feature to the samples X and returned afterwards by the method.

This routine has been coupled with DecisionTree classifier using **pipeline** class in scikit-learn, which construct acts like a classifier (has fit and predict methods).

The success rate of this method was verified, once more, using classification_report method from scikit-learn. The result before the test are the same as in table 5.1, while the results after using the described method are shown in table 5.3.

Table 5.3: Classification report after using the distance routine

	precision	recall	f1-score	support
NoEvents	0.72	0.79	0.75	51797
Attack	0.28	0.23	0.25	17382
Natural	0.19	0.08	0.11	4232
accuracy			0.62	73411
macro avg	0.40	0.37	0.37	73411
weighted avg	0.58	0.62	0.60	73411

Tables 5.1 and 5.3 show an increase of model's accuracy by 0.02. Precision value for Attack class increased by 0.01, with no changes for other classes. Recall value increased for NoEvents by 0.03

and decreased for Attack by 0.03. Finally, f-measure increased for NoEvents class and decreased by 0.01 for other classes. The f-measure value difference for Natural class, despite the same precision and recall, is due to rounding numbers to two decimal places.

It may be concluded that this method does not really enhance the results, despite the better accuracy. For the other metrics, ones increased, but other decreased, what makes the obtained result less satisfying. In addition to that, the same problem with unequal class distribution mentioned in previous method discussion arises.

5.3 Hidden Markov Models

Chapter 6

Conclusions

Bibliography

[1] T. Morris, "Industrial Control System (ICS) Cyber Attack Datasets - Tommy Morris." https://sites.google.com/a/uah.edu/tommy-morris-uah/ics-data-sets.

[2] U. Adhikari, S. Pan, and T. Morris, "Power System Attack Datasets," Apr. 2014.

[3] "Three-phase electric power," *Wikipedia*, May 2020.

[4] J.L. Kirtley Jr., "Introduction To Symmetrical Components," *6.061 Introduction to Power Systems*, vol. Class Notes Chapter 4.

[5] R. C. Borges Hink, J. M. Beaver, M. A. Buckner, T. Morris, U. Adhikari, and S. Pan, "Machine learning for power system disturbance and cyber-attack discrimination," in *2014 7th International Symposium on Resilient Control Systems (ISRCS)*, (Denver, CO, USA), pp. 1–8, IEEE, Aug. 2014.

[6] "Appendix B - The WEKA workbench," in *Data Mining (Fourth Edition)* (I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, eds.), pp. 553–571, Morgan Kaufmann, fourth edition ed., 2017.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[8] "Who is using scikit-learn? — scikit-learn 0.23.1 documentation." https://scikit-learn.org/stable/testimonials/testimonials.html.

[9] "RandomForest." https://weka.sourceforge.io/doc.dev/weka/classifiers/trees/RandomForest.html.

[10] "Sklearn.ensemble.RandomForestClassifier — scikit-learn 0.23.1 documentation." https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[11] A. Yadav, "SUPPORT VECTOR MACHINES(SVM)." https://towardsdatascience.com/support-vector-machines-svm-c9ef22815589, Oct. 2018.

[12] "Support Vector Machines, Clearly Explained!!!," Jan. 2014.

[13] "LibSVM." https://javadoc.scijava.org/Weka/weka/classifiers/functions/LibSVM.html.

[14] "Sklearn.svm.SVC — scikit-learn 0.23.1 documentation." https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

[15] "NaiveBayes." https://weka.sourceforge.io/doc.dev/weka/classifiers/bayes/NaiveBayes.html.

[16] "Sklearn.naive_bayes.GaussianNB — scikit-learn 0.23.1 documentation." https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html.

[17] V. Patel, "Implementing a Multi Layer Perceptron Neural Network in Python." https://towardsdatascience.com/implementing-a-multi-layer-perceptron-neural-network-in-python-b22b5a3bdfa3, May 2020.

[18] "Neural network models (supervised)." https://scikit-learn.org/stable/modules/neural_networks_supervise formulation.

[19] K. Stokfiszewski, "Soft Computing Laboratory 1. The Delta rule."

[20] "Sklearn.neural_network.MLPClassifier — scikit-learn 0.23.1 documentation." https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html#sklearn.neural_network

[21] B. Shmueli, "Multi-Class Metrics Made Simple, Part II: The F1-score." https://towardsdatascience.com/multi-class-metrics-made-simple-part-ii-the-f1-score-ebe8b2c2ca1, June 2020.

[22] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why Should I Trust You?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pp. 1135–1144, 2016.

[23] Mikhail Korobov and Konstantin Lopuhin, "ELI5." https://github.com/TeamHG-Memex/eli5.

[24] B. Bengfort, R. Bilbro, N. Danielsen, L. Gray, K. McIntyre, P. Roman, Z. Poh, *et al.*, "Yellowbrick," 2018-11-14, 2018.

[25] Ando Saabas, "TreeInterpreter." https://github.com/andosa/treeinterpreter.

[26] Terence Parr and Prince Grover, “Dtreeviz.” https://github.com/parrt/dtreeviz.