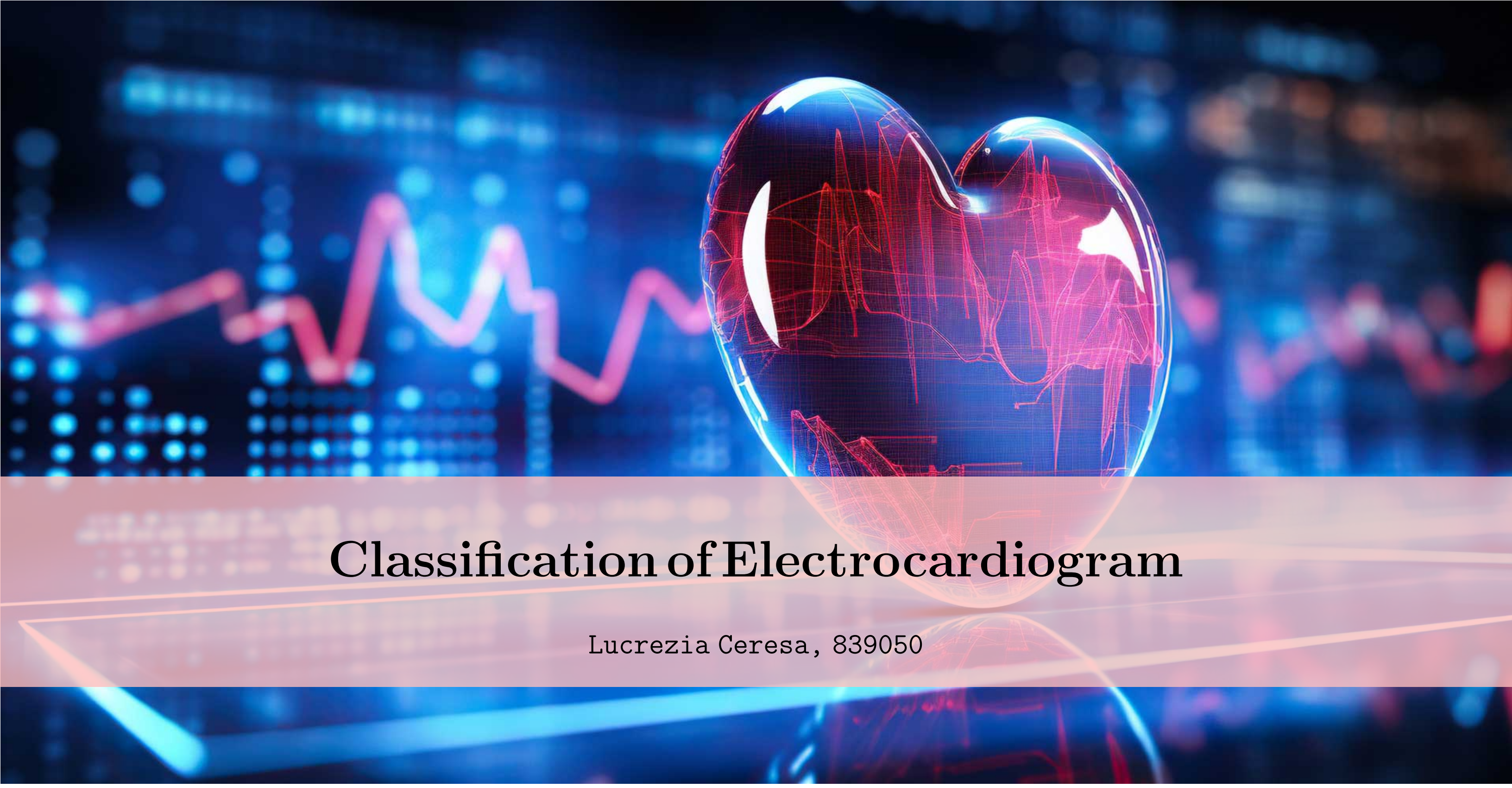

# Classification of Electrocardiogram

Lucrezia Ceresa, 839050

## Contents

**1 Introduction** **1**
1.1 Electrocardiogram (ECG) . . . . . . . . . . 1
1.2 Classification . . . . . . . . . . 3

**2 Modeling** **4**
2.1 $k$-Nearest Neighbors ($k$NN) . . . . . . . . . . 4
2.2 Random Forest . . . . . . . . . . 8
2.2.1 Decision Tree Classifier . . . . . . . . . . 9
2.2.2 Design Issues of Decision Tree Induction . . . . . . . . . . 11
2.2.3 Decision Tree and Random Forest . . . . . . . . . . 12
2.3 Artificial Neural Network . . . . . . . . . . 14
2.3.1 Perceptron . . . . . . . . . . 14
2.3.2 Multi-layer Neural Network . . . . . . . . . . 15

**3 Algorithm Presentation** **21**
3.1 $k$NN Algorithm . . . . . . . . . . 21
3.2 Decision Tree Algorithm . . . . . . . . . . 21
3.3 Random Forest Algorithm . . . . . . . . . . 22
3.4 ANN Algorithm . . . . . . . . . . 22

**4 Algorithm Analysis** **24**
4.1 $k$-Nearest Neighbors Algorithm Analysis . . . . . . . . . . 24
4.2 Random Forest Algorithm Analysis . . . . . . . . . . 24
4.3 Artificial Neural Network Algorithm analysis . . . . . . . . . . 25

**5 Conclusions** **27**

## 1 Introduction

As of the early 21st century, 56 million people die per year. The most common reason is cardiovascular disease, which is a disease that affects the heart or the blood vessels. The deaths at world level decreased from 12,4 million in 1990 to 19,8 million in 2022. This is a good trend if you consider the population aging and growth, but however the percentages are very high (48% in Europe, 35% in the world every year).

For these reasons it is important monitoring the state of health of the heart. A quick non-invasive test that everyone can do is the electrocardiogram (ECG or EKG). It shows a graph of th electrical activity of the heart using electrodes placed on the skin. Test results can help diagnose heart attacks and irregular heartbeats, called arrhythmia.

Symptoms of heart diseases come and go, so it could be crucial repeating very often the test to avoid critical consequences. But, at the same time, it can be unfeasible to be frequently visited by a doctor. This is the reason why in the last few years, a great effort was made for the creation of portable instruments able to record heart impulses and the implementation of automatic algorithms that allows a timely ECG interpretation.

Several algorithms are used for the classification of ECG. Traditional machine learning examples are Support Vector Machine (SVM), Random Forest, $k$-Nearest Neighbors ($k$NN), Naive Bayes. Also Deep Learning algorithms are used increasingly frequent. All the algorithms interpret the measures of the peaks and the intervals between them and categorize the waves as normal or irregular. In some case (depending on the dataset) they can also give an interpretation on what kind of irregularity is happening.

The aim of this paper is to analyze three algorithms: $k$-nearest neighbors, Random Forest and Artificial Neural Network. In particular, the computational complexity and the accuracy will be shown and compared.

In the following subsections the Electrocardiogram test and the problem of Classification will be explored separately for a better comprehension of the task.

### 1.1 Electrocardiogram (ECG)

To explore every cardiac irregularity, is necessary to record a **12-lead ECG**. The name is due to the positions of the leads from which signals arrive. The Figure 1 shows the angles that must be set between the leads.

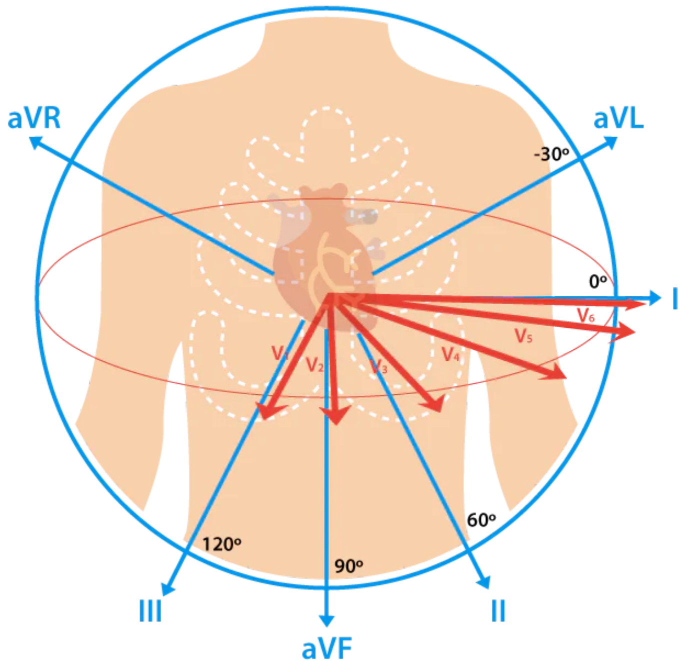

Figure 1: 12-leads ECG angles

Twelve electrodes are placed on the patient's limbs and on the surface of the chest. The overall magnitude of the heart's electrical potential is then measured from twelve different angles and is recorded over a period of time (usually ten seconds). In this way, the overall magnitude and direction of the heart's electrical depolarization is captured at each moment throughout the cardiac cycle (see Figure 2).

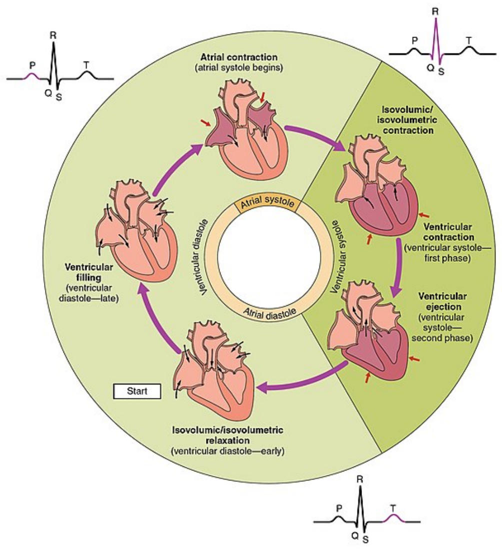

Figure 2: Cardiac Cycle

At each heart beat a peak, called **R peak** is recorded. It is preceded by two waves containing the **P peak** and the **Q peak** and it is followed by other two waves containing the **S peak** and the **T peak**. The Figure 3 shows a typical shape of the so-called **QRS complex**.

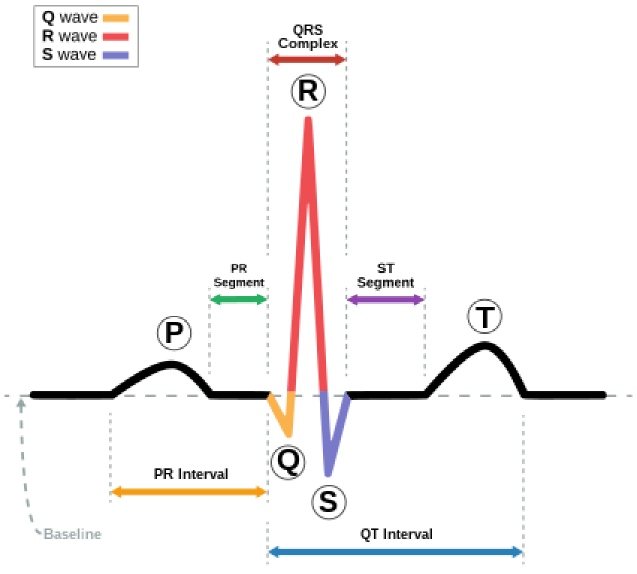

Figure 3: QRS complex

This paper aims to classify a **serial electrocardiography**. Serial analysis of ECGs is an important diagnostic tool that relies on comparing two or more successive recordings of the same patient. Even small, differences between two recordings may have a pathological origin: the role played by experienced interpreters is therefore crucial. These changes are either used to detect new pathology or to verify the efficacy of a specific therapy or intervention.

Helping human resources is really important for two reasons. As we said in the introduction, the first is being able to take the ECG test without needing to travel to specialized medical environments. The second is trying to avoid the human mistakes. A great deal of information is needed to analyze this type of test, and multiple conditions must be considered. For example, the resting heart rate must be between 60 and 100 beats/min, although some rhythms may be basic normal, but intermittent variation in the RR interval (i.e., the time between one R peak and the next) makes them irregular. In contrast, if the heart rhythm is irregular, one must decide whether it is completely irregular with no discernible RR pattern, whether the rhythm is regular with intermittent irregularity, or whether there is cyclic variation in the RR intervals. Other information to be taken into account are measurements regarding ectopic (or premature) beats, duration of the QRS complex, atrial activity and relationship to ventricular activity, etc. Once these characteristics have been assessed, a diagnosis must be made. Fibrillation, tachycardia, or asystole or at worst pulseless electrical activity may present.

For each of the diagnoses listed above, the timely intervention of physicians or nurses is necessary. It's for this reason that nowadays, devices (such as watches, chest straps...) are very popular. If worn correctly, they are able to make an analysis of the heartbeat using classification algorithms and, if needed, they are able to contact the appropriate phone numbers (such as 112 in Italy).

## 1.2 Classification

The classification is the task of assigning labels to unlabeled data instances and a **classifier** is used to perform such a task. A classifier is typically described in terms of a model that is created using a given set of instances, known as the **training set**, which contains attribute values as well as class labels for each instance. Finally it is evaluated on another set of new data, called **test set**.

The classification can be **supervised** or **unsupervised**. In the first case, the sets of data contain the class label and the aim is to predict them on the test set with the lowest possible error rate (i.e. the lowest number of mismatched predictions). The second one consists in a procedure that generates groups of data based on certain kind of similarities between instances, without having any class labels.

In this document the objective is to explore procedures of supervised classification of serial electrocardiograms, where the classes to predict are labeled as 1 or 0, where 1 indicates a normal heartbeat and 0 represents some kind of abnormal measurement.

The systematic approach for learning a classification model given a training set is known as a **learning algorithm**. The process of using a learning algorithm to build a classification model from the training data is known as **induction**. This process is also often described as "learning a model" or "building a model". This process of applying a classification model on unseen test instances to predict their class labels is known as **deduction**. Thus, the process of classification involves two steps: applying a learning algorithm to training data to learn a model, and then applying the model to assign labels to test instances.

## 2 Modeling

The ECG Classification is a process that aims to categorize a heartbeat, that can be normal or irregular, on the base of the intervals and peaks measures. The graph obtained by ECG test shows the change of voltage during a specific period of time. Typically, peaks are measured in millivolts (mV) and intervals are measured in milliseconds (ms). It is important to analyze records of each one of the 12 leads (so 12 graphs must be considered), otherwise the risk that some critical issues will not emerge is high.

Measures of peaks and intervals will be processed for building a supervised model. Then classes of a new set of data will be predicted.

The three models we aim to compare work in very different ways. In the following subsections we will explore them in details.

### 2.1 $k$-Nearest Neighbors ($k$NN)

The $k$-Nearest Neighbors algorithm, or $k$NN, can be used to determine the class label of a test instance analyzing the $k$ training examples that are relatively similar to its attributes. It makes use of the **nearest neighbor classifier**, that represents each example as a point in a $d$-dimensional space, where $d$ is the number of attributes. Given a test instance, we compute its distance to the training data points according to a **proximity measure**. The $k$-nearest neighbors of a given test instance $z$ refer to the $k$ training examples that are closest to $z$. This algorithm is a so-called **lazy learner**, that means it delays the process of modeling the training data until it is needed to classify the test instances.

The instance is classified based on the class labels of its neighbors. In the case where the neighbors have more than one label, the test instance is assigned to the **majority class** of its nearest neighbors.

Let's see an example where we have two classes. In Figure 4, the 1-nearest neighbor of the instance is a negative example. Therefore the instance is assigned to the negative class. If the number of nearest neighbors is three, as shown in Figure 6, then the neighborhood contains two positive examples and one negative example. Using the majority voting scheme, the instance is assigned to the positive class. In the case where there is a tie between the classes (see Figure 5), we may randomly choose one of them to classify the data point.

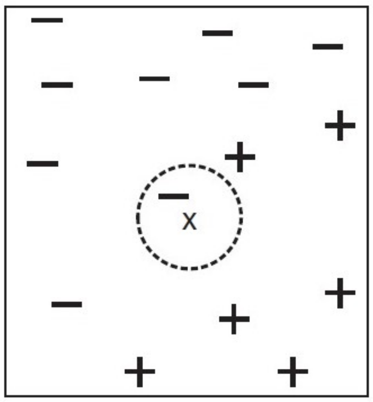
Figure 4: 1-NN

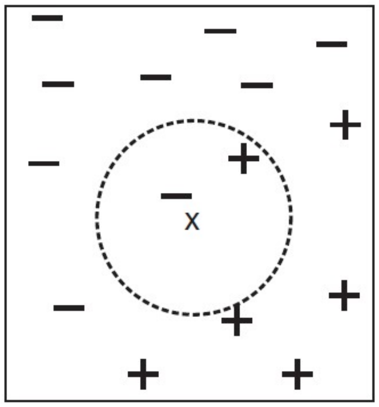
Figure 5: 2-NN

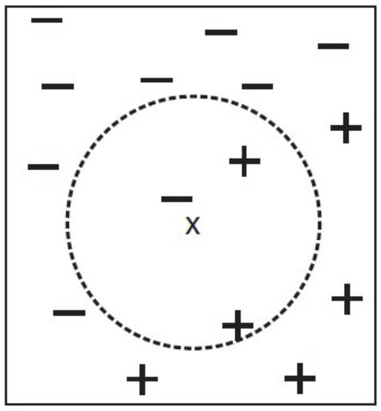
Figure 6: 3-NN

The preceding discussion underscores the importance of choosing the right value for $k$. If $k$ is too small, then the nearest neighbor classifier may be susceptible to overfitting due to noise (i.e., mislabeled example in the training data). On the other hand if $k$ is too large the nearest neighbor classifier may misclassify the test instances as it is shown in Figure 7.

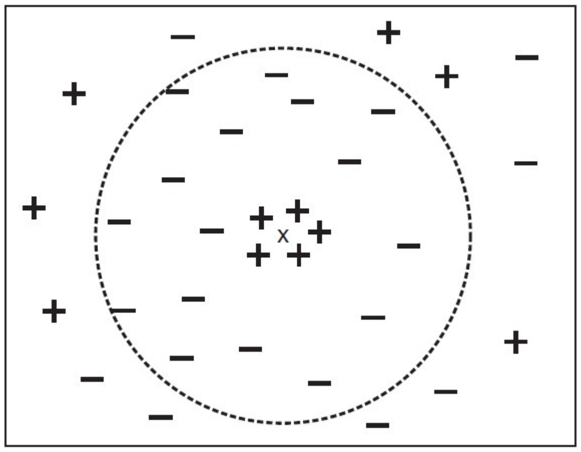

Figure 7: $k$NN with large $k$

Let's consider the Euclidean distance as proximity measure. Each element of the training set $D$ has $d$ attribute and one target, i.e.: $\mathbf{x}_i = (x_i^1, ..., x_i^d, y_i) \in D$ for $i = 1, ..., n$. Suppose that $\mathbf{x}_0$ is a new data that must be classified. First of all, we compute the distance between $\mathbf{x}_0$ and all the instances present in the training dataset $D$. That is, for each $\mathbf{x}_1, ..., \mathbf{x}_n \in D$, compute:

$$dist_i(\mathbf{x}_0, \mathbf{x}_i) = \sqrt{(x_0^1 - x_i^1)^2 + ... + (x_0^d - x_i^d)^2}$$

If $d_{i_1}, ..., d_{i_k}$ are the $k$ smallest distances, then the data points $\mathbf{x}_{i_1}, ..., \mathbf{x}_{i_k}$, must be considered for computing the majority vote. Then compute $\hat{y}$ as follows:

$$\hat{y} = \arg\max_{c \in C} \sum_{j=1}^{k} I(y_{i_j} = c)$$

where $C$ is the set of the class labels and $I(\cdot)$ is the characteristic function. If $\hat{y} = y_0$ the the $k$ nearest neighbor classifier has found the correct class for $\mathbf{x}_0$, otherwise there is a misclassification.

To show the algorithm's behavior, we consider a simple example: for a better visualization only twenty measures of two features are taken in account. The two features are RR intervals and R peaks. The values shown in the Table 1 are taken from the *St Petersburg INCART 12-lead Arrhythmia Database*.

Now, let's divide the twenty records into train and test set. In particular, the 20% will represent the test set and the remaining 80% will represent the training set. This subdivision is shown in the Table 2. Our task is to predict the CLASS column of the test set.

| CLASS | RR INTERVAL | R PEAK |
|---|---|---|
| 1 | 163 | 0.614133 |
| 1 | 165 | -0.078704 |
| 1 | 166 | -0.010649 |
| 1 | 231 | -0.083499 |
| 1 | 165 | 0.613374 |
| 1 | 163 | -0.145137 |
| 1 | 166 | 1.120258 |
| 1 | 161 | 0.601190 |
| 1 | 160 | -0.018564 |
| 1 | 158 | -0.053904 |
| 1 | 162 | -0.062096 |
| 1 | 161 | 0.719685 |
| 1 | 156 | -0.164207 |
| 1 | 164 | 0.756064 |
| 1 | 229 | 0.845887 |
| 1 | 157 | -0.069553 |
| 0 | 102 | -0.101098 |
| 0 | 99 | -0.337828 |
| 0 | 107 | -0.002046 |
| 0 | 98 | -0.214091 |

Table 1: Subset of *INCART* dataset

| CLASS | RR INTERVAL | R PEAK |
|---|---|---|
| 1 | 163 | 0.614133 |
| 1 | 165 | -0.078704 |
| 1 | 166 | -0.010649 |
| 1 | 231 | -0.083499 |
| 1 | 165 | 0.613374 |
| 1 | 163 | -0.145137 |
| 1 | 166 | 1.120258 |
| 1 | 161 | 0.601190 |
| 1 | 160 | -0.018564 |
| 1 | 158 | -0.053904 |
| 1 | 162 | -0.062096 |
| 1 | 161 | 0.719685 |
| 0 | 102 | -0.101098 |
| 0 | 99 | -0.337828 |

| CLASS | RR INTERVAL | R PEAK |
|---|---|---|
| N | 156 | -0.164207 |
| N | 164 | 0.756064 |
| N | 229 | 0.845887 |
| N | 157 | -0.069553 |
| VEB | 98 | -0.214091 |
| VEB | 107 | -0.002046 |

Table 2: Train and test datsets

The chosen points are well separated. This situation is really unfeasible in the real world problems, but it is used only for helping in the explanation of the algorithm.

Now, let's see in details what is the behaviour of the $k$NN algorithm, for $k = 3$. In the Figures from 8 to 13 are shown the steps of the prediction. The blue points represent the training instances with CLASS=0, while the red points represent those once with CLASS=1.

In the first step we randomly choose the point with index equal to 14. Then the three smallest distances are printed in green to show what are the training points considered for predicting the new class. They are all red, then the the majority vote will be 1.

In the second step, the chosen point has index equal to 31. the three nearest neighbors are two blue and one red. Then the prediction will be 0.

In the third step, the three nearest neighbors to the point 51 are all blue. Then also in this case the predicted class will be 0.

The remaining steps are similar: in all the cases the three nearest neighbors are red. This means that the classes of the test examples are all 1.

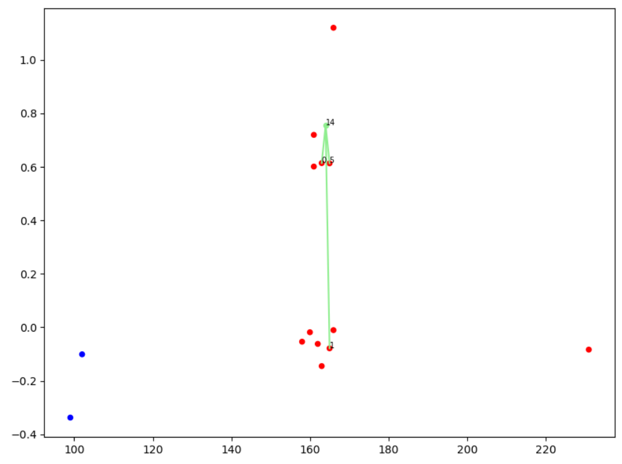
Figure 8: First step

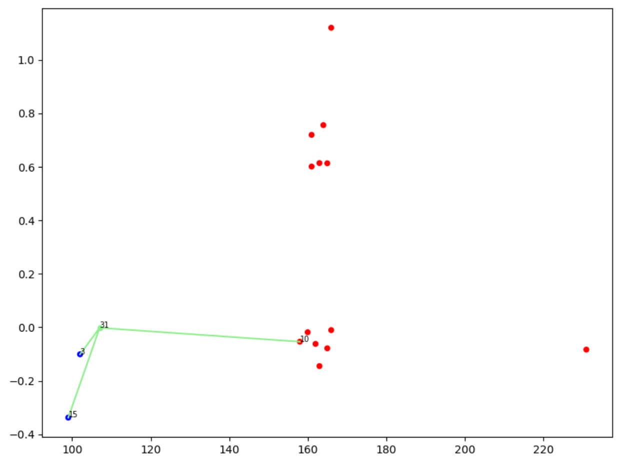
Figure 9: Second step

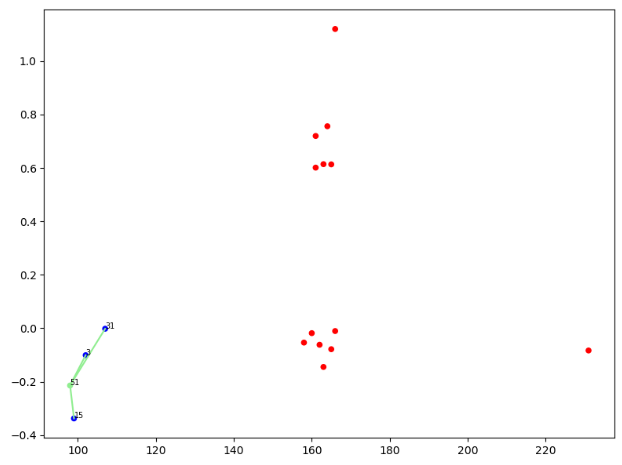
Figure 10: Third step

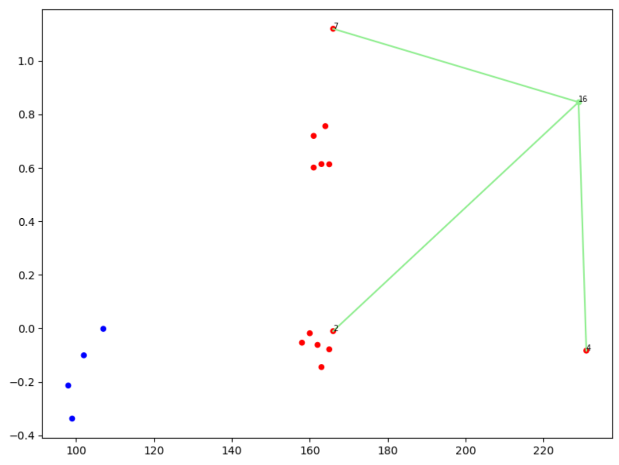
Figure 11: Fourth step

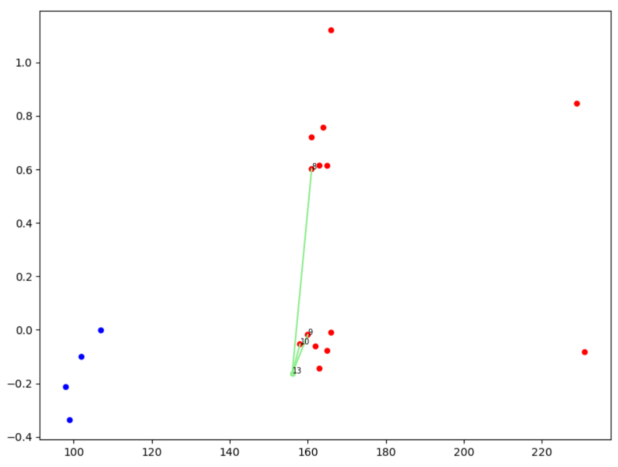

Figure 12: Fifth step

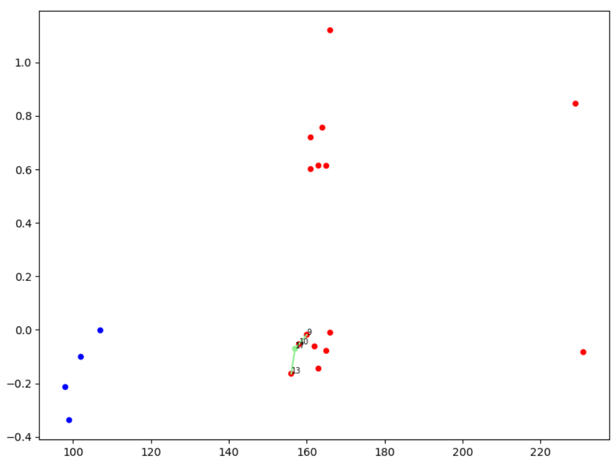

Figure 13: Sixth step

The Figure 14 shows the final step: all the six test instances are predicted.

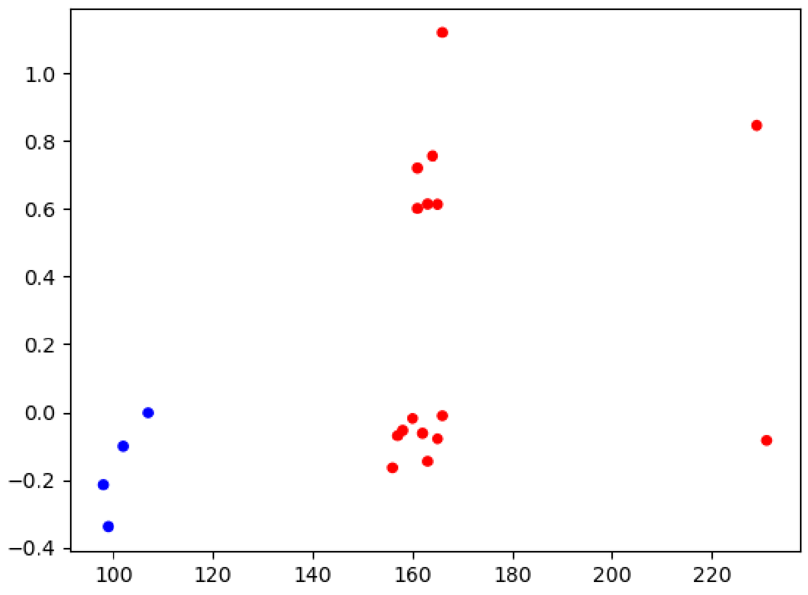

Figure 14: Final step

## 2.2 Random Forest

Random Forest is an **ensemble method**, because it puts together the results of other algorithms to make a classification. In particular it uses *decorrelated* decision trees as its base classifiers. For classification task, the output of the random forest is the class selected by the most of the trees.

For this reason in this section we will explore Decision Tree classifier and the technique of bootstrap, essential for the model.

### 2.2.1 Decision Tree Classifier

The decision tree classifier is a simple classification technique. It is based on a series of questions and their possible answers that can be organized into a hierarchical structure called a **Decision Tree** (DT).

The tree has three types of nodes:

- a **root node**, with no incoming links and zero or more outgoing links;
- **internal nodes**, each of which has exactly one incoming link and two or more outgoing links;
- **leaf** or **terminal nodes**, each of which has exactly one incoming link and no outgoing links.

Every leaf node in the decision tree is associated with a class label. The **non-terminal nodes**, which include the root and internal nodes, contain **attribute test conditions** that are typically defined using a single attribute. Each possible outcome of the attribute test condition is associated with exactly one child of this node.

The attribute test condition maximizes the separation of data according to a certain metric (like the Gini index or the Entropy, shown in Section 2.2.2).

The process continues until data are correctly classified or a certain stopping criterion is reached (for instance, the max depth of the tree).

**Hunt's Algorithm** Many possible decision trees can be constructed from a particular dataset. One of the earliest method is **Hunt's algorithm**, which is the basis for many current implementations of decision tree classifiers. In this paper it will be explored and implemented.

In Hunt's algorithm, a decision tree is grown in a recursive fashion. The tree initially contains a single root that is associated with all the training instances. If a node is associated with instances from more than one class, it is expanded using an attribute test condition that is determined using a **splitting criterion**. A child leaf node is created for each outcome of the attribute test condition and the instances associated with the parent node are distributed to the children based on the test outcomes. This node expansion step can then recursively applied to each child node, as long as it has labels of more than one class. If all the instances associated to a leaf node have identical class labels (or if a stopping condition is set), then the node is not expanded any further. Each leaf node is assigned to a class label that occurs most frequently in the training instances associated to the node.

To illustrate how the algorithm works, consider the training set shown in Table 3 for the classification of ECG.

| CLASS | RR INTERVAL | R PEAK |
|---|---|---|
| 1 | 163 | 0.614133 |
| 1 | 165 | -0.078704 |
| 1 | 166 | -0.010649 |
| 1 | 231 | -0.083499 |
| 1 | 165 | 0.613374 |
| 1 | 163 | -0.145137 |
| 1 | 166 | 1.120258 |
| 1 | 161 | 0.601190 |
| 1 | 160 | -0.018564 |
| 1 | 158 | -0.053904 |
| 1 | 162 | -0.062096 |
| 1 | 161 | 0.719685 |
| 0 | 102 | -0.101098 |
| 0 | 99 | -0.337828 |

Table 3: Subset of INCART dataset

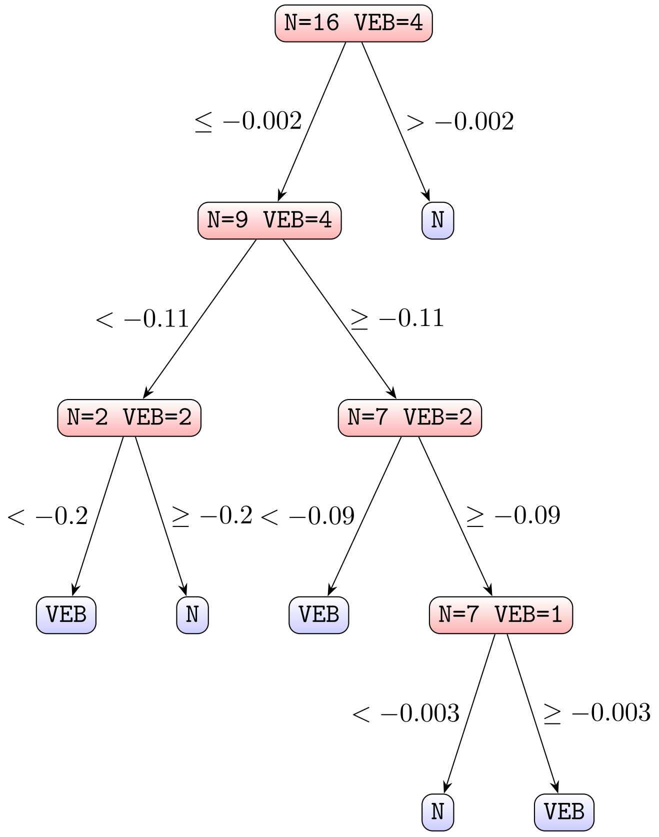

Figure 15: A decision tree for the ECG classification problem using conditions about R PEAK.

We want to to split the table until the new subsets contain only one label (1 or 0) into the CLASS column. For example, the test conditions of the root and the internal nodes of the tree shown in Figure 15 concern the attribute R PEAK.

A shorter tree (and so a better idea) is given by taking in account conditions about the RR INTERVAL column. Indeed, if we use the conditions:

$$\text{RR INTERVAL} \leq 107 \qquad \text{and} \qquad \text{RR INTERVAL} > 107$$

then the Table 3 will be divided in two subsets containing only instances from one class for each. The picture of the resulting tree is shown in the Figure 16.

The tree is really much smaller then the previous one. A smaller tree is preferable then a bigger one because it is more interpretable and a lower amount of resources, in terms of time and memory, is used.

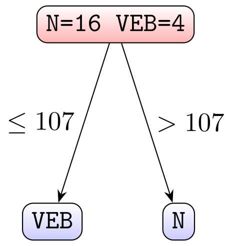

Figure 16: A decision tree for the ECG classification problem using conditions about RR INTERVAL.

### 2.2.2 Design Issues of Decision Tree Induction

Hunt's algorithm is a generic procedure for growing decision trees in a greedy fashion. To implement the algorithm, there are two key design issues that must be addressed.

- **What is the splitting criterion?** At each recursive step, an attribute must be selected to partition the training instances associated with a node into smaller subsets associated with its child nodes. The splitting criterion determines which attribute is chosen as the test condition and how the training instances should be distributed to the child nodes.
- **What is the stopping criterion?** The basic algorithm stops expanding a node only when all the training instances associated with the node have the same class labels or have identical attribute values. Although these conditions are sufficient, there are reasons to stop expanding a node much earlier even if the leaf node contains training instances from more than one class. This process is called **early termination** and the condition used to determine when a node should be stopped from expanding is called a **stopping criterion**.

**Methods for Expressing Attribute Test Conditions** Decision tree induction algorithms must provide a method for expressing an attribute test condition and its corresponding outcomes for different attribute types.

- **Binary Attributes**: The test condition for a binary attribute generates two potential outcomes
- **Nominal Attributes**: Since a nominal attribute can have many values, its attribute test condition can be expressed in two ways, as a multi-way split or a binary split.
- **Ordinal Attributes**: Ordinal attributes can also produce binary or multi-way splits. Ordinal attribute values can be grouped as long as the grouping does not violate the order property of the attribute values.
- **Continuous Attributes**: For continuous attributes, the attribute test condition can be expressed as a comparison test (e.g., $Attribute < value$) producing a binary split, or as a range query of the form $v_i \leq A < v_{i+1}$, for $i = 1, ..., k$, producing a multi-way split.

**Measures for Selecting an Attribute Test Condition** There are many measures that can be used to determine the goodness of an attribute test condition. These measures try to give preference to attribute test conditions that partition the training instances into *purer* subsets in the child nodes, which mostly have the same class labels. Having purer nodes is useful since a node that has all of its training instances from the same class does not need to be expanded further. In contrast, an impure node containing training instances from multiple classes is likely to require several levels of node expansions, thereby increasing the depth of the tree considerably. Larger trees are less desirable as they are more susceptible to model overfitting, a condition that may degrade the classification performance on unseen instances. They are also difficult to interpret and incur more training and test time as compared to smaller trees.

**Impurity Measure for a Single Node** The impurity of a node measures how dissimilar the class labels are for the data instances belonging to a common node. Following are examples of measures that can be used to evaluate the impurity of a node $t$:

$$\text{Entropy} = -\sum_{i=1}^{c} p_i(t) \log_2 p_i(t)$$

$$\text{Gini index} = 1 - \sum_{i=1}^{c} p_i(t)^2$$

$$\text{Classification error} = 1 - \max_{i \in \{1,\ldots,c\}} [p_i(t)]$$

where $p_i(t)$ is the relative frequency of training instances that belong to class $i$ at node $t$, $c$ is the total number of classes, and $0 \log_2 0 = 0$ in entropy calculations. All three measures give a zero impurity value if a node contains instances from a single class and maximum impurity if the node has equal proportion of instances from multiple classes.

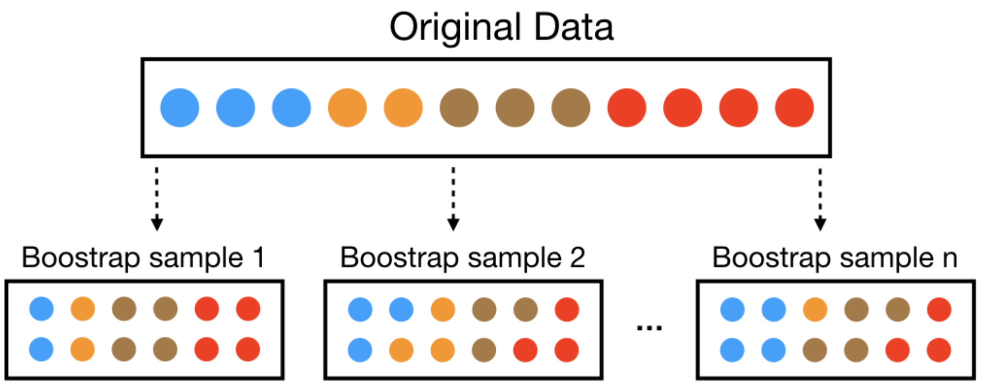

Figure 17: A scheme of bootstrap

**Collective Impurity of Child Nodes** Consider an attribute test condition that splits a node containing $N$ training instances into $k$ children, $\{v_1, v_2, .., v_k\}$, where every child node represents a partition of the data resulting from one of the $k$ outcomes of the attribute test condition. Let $N(v_j)$ be the number of training instances associated with a child node $v_j$ , whose impurity value is $I(v_j)$. Since a training instance in the parent node reaches node $v_j$ for a fraction of $N(v_j)/N$ times, the collective impurity of the child nodes can be computed by taking a **weighted sum of the impurities** of the child nodes, as follows:

$$I(\text{children}) = \sum_{j=1}^{k} \frac{N(v_j)}{N} I(v_j).$$

**Identifying the best attribute test condition** To determine the goodness of an attribute test condition, we need to compare the degree of impurity of the parent node (before splitting) with the weighted degree of impurity of the child nodes (after splitting). The larger their difference, the better the test condition. This difference, $\Delta$, also termed as the **gain impurity** of an attribute test condition, can be defined as follows:

$$\Delta = I(\text{parent}) - I(\text{children})$$

where $I(\text{parent})$ is the impurity of a node before splitting and $I(\text{children})$ is the weighted impurity measure after splitting. It can be shown that the gain is non-negative since $I(\text{parent}) \geq I(\text{children})$ for any reasonable measure such as those presented above. The higher the gain, the purer are the classes in the child nodes relative to the parent node. The splitting criterion in the decision tree learning algorithm selects the attribute test condition that shows the maximum gain. Note that maximizing the gain at a given node is equivalent to minimizing the weighted impurity measure of its children since $I(\text{parent})$ is the same for all candidate attribute test conditions. Finally, when entropy is used as the impurity measure, the difference in entropy is commonly known as **information gain**, $\Delta_{info}$.

### 2.2.3 Decision Tree and Random Forest

The Random Forest algorithm is an **ensamble method**, i.e., a technique for improving classification accuracy by aggregating the predictions of multiple classifiers, called **base classifiers**. In the case of Random Forest, the base classifiers are *decorrelated* decision trees.

Random Forest uses a different **bootstrap** sample of the training data for learning decision trees. The bootstrap method, schematized in the Figure 17, implies that training instances are sampled *with replacement* from the labeled set, i.e., an instance previously selected to be part of the training set is equally likely to be drawn again. If the original data has $N$ instances, it can be shown that, on average, a bootstrap sample of size $N$ contains about 63.2% of the instances in the original data.

At every internal node of a tree, the best splitting criterion is chosen among a small set of randomly selected attributes. In this way, random forests construct ensembles of decision trees by not only manipulating

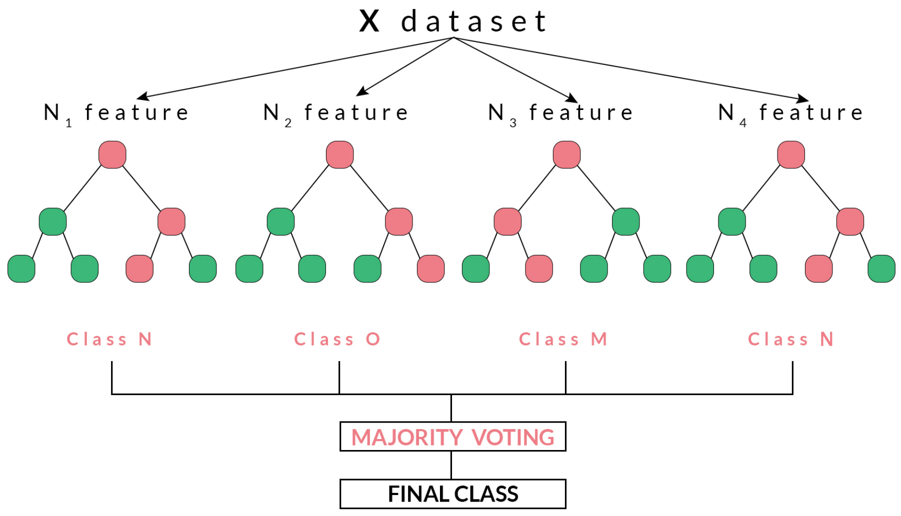

Figure 18: A scheme of random forests algorithm

training instances (by using bootstrap samples), but also the input attributes (by using different subsets of attributes at every internal node).

Given a training set $D$ consisting of $n$ instances and $d$ attributes, the basic procedure of training a random forest classifier can be summarized using the following steps:

1. Construct a bootstrap sample $D_i$ of the training set by randomly sampling $n$ instances (with replacement) from $D$.
2. Use $D_i$ to learn a decision tree $T_i$ as follows. At every internal node of $T_i$, randomly sample a set of $p$ attributes and choose an attribute from this subset that shows the maximum reduction in an impurity measure for splitting. Repeat this procedure till every leaf is pure, i.e., containing instances from the same class.

Once an ensemble of decision trees have been constructed, their average prediction (majority vote) on a test instance is used as the final prediction of the random forest. A scheme representing the algorithm is shown in Figure 18.

The number of attributes to be selected at every node, $p$, is a hyperparameter of the random forest classifier. A small value of $p$ can reduce the correlation among the classifiers but may also reduce their strength. A large value can improve their strength but may result in correlated trees. Although common suggestions for $p$ in the literature include $\sqrt{p}$ and $\log_2(d+1)$.

There is a way for selecting hyper-parameters in random forests, which does not require using a separate validation set. It involves computing a reliable estimate of the generalization error rate directly during training, known as the **out-of-bag (oob)** error estimate. The approach for computing oob estimate can be described as follows.

Consider an independent base classifier $T_i$ built on a bootstrap sample of the training set $D_i$. Since every training instance $\mathbf{x}$ will be used for training approximately 63% of base classifiers, we can call $\mathbf{x}$ as an **out-of-bag sample** for the remaining 27% of base classifiers that did not use it for training. If we use these remaining 27% classifiers to make predictions on $\mathbf{x}$, we can obtain the oob error on $\mathbf{x}$ by taking their majority vote and comparing it with its class label. Note that the oob error estimates the error of 27% classifiers on an instance that was not used for training those classifiers. Hence, the oob error can be considered as a reliable estimate of generalization error. By taking the average of oob errors of all training instances, we can compute the overall oob error estimate. This can be used as an alternative to the validation error rate for selecting hyperparameters. Hence, random forests do not need to use a separate partition of the training set for validation, as it can simultaneously train the base classifiers and compute generalization error estimates on the same data set.

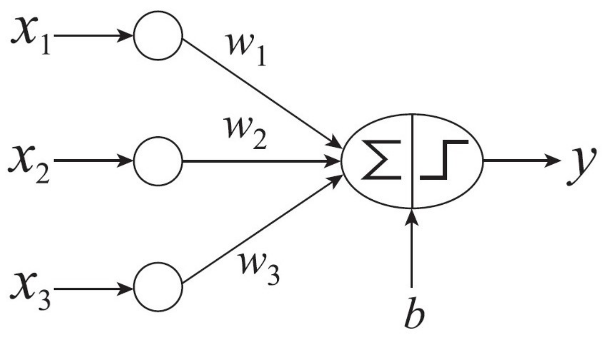

Figure 19: Basic architecture of a perceptron.

## 2.3 Artificial Neural Network

Today, the Artificial Neural Networks (ANN) are fundamental for the automatic classification of serial ECG. Historically, the study of ANN was inspired by attempts to emulate biological neural systems. The human brain consists primarily of nerve cells called **neurons**, linked together with other neurons via strands of fiber called **axons**. Whenever a neuron is stimulated (e.g., in response to a stimuli), it transmits nerve activations via axons to other neurons. The receptor neurons collect these nerve activations using structures called **dendrites**, which are extensions from the cell body of the neuron. The strength of the contact point between a dendrite and an axon, known as a synapse, determines the connectivity between neurons. Neuroscientists have discovered that the human brain learns by **changing the strength** of the synaptic connection between neurons upon repeated stimulation by the same impulse.

The composition of these simple functions is able to express complex functions. This idea is at the basis of constructing ANN. Indeed, it is composed of a number of processing units, called nodes, that are connected with each other via directed links. The nodes correspond to neurons that perform the basic units of computation, while the directed links correspond to connections between neurons, consisting of axons and dendrites. Further, the weight of a directed link between two neurons represents the strength of the synaptic connection between neurons. As in biological neural systems, the primary objective of ANN is to adapt the weights of the links until they fit the input-output relationships of the underlying data.

### 2.3.1 Perceptron

A perceptron is a basic type of ANN model that involves two types of nodes: input nodes, which are used to represent the input attributes, and an output node, which is used to represent the model output. Figure 19 illustrates the basic architecture of a perceptron that takes three input attributes, $x_1$, $x_2$, and $_3$, and produces a binary output $y$. The input node corresponding to an attribute $x_i$ is connected via a weighted link $w_i$ to the output node. The weighted link is used to emulate the strength of a synaptic connection between neurons.

The output node is a mathematical device that computes a weighted sum of its inputs, adds a bias factor $b$ to the sum, and then examines the sign of the result to produce the output $\hat{y}$ as follows:

$$\hat{y} = \begin{cases} 1, & \text{if } \mathbf{w}^T\mathbf{x} + b > 0 \\ -1, & \text{otherwise} \end{cases}$$

To simplify notations, $\mathbf{w}$ and $b$ can be concatenated to form $\tilde{\mathbf{w}} = (\mathbf{w}^T b)^T$, while $\mathbf{x}$ can be appended with 1 at the end to form $\tilde{\mathbf{x}} = (\mathbf{x}^T 1)^T$. The output of the perceptron $\hat{y}$ can then be written as:

$$\hat{y} = \text{sign}(\tilde{\mathbf{w}}^T\tilde{\mathbf{x}})$$

where the sign function acts as an activation function by providing an output value of +1 if the argument is positive and -1 if its argument is negative.

**Learning the Perceptron** Given a training set, we are interested in learning parameters $\tilde{\mathbf{w}}$ such that $\hat{y}$ closely resembles the true $y$ of training instances. This is achieved by using the perceptron learning algorithm. The key computation for this algorithm is the iterative weight update formula of the algorithm:

$$w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i - \hat{y}_i^{(k)})x_{ij},$$

where $w_j^{(k)}$ is the weight parameter associated with the $j^{th}$ input link after the $k^{th}$ iteration, $\lambda$ is a parameter known as the **learning rate**, and $x_{ij}$ is the value of the $j^{th}$ attribute of the training example $\mathbf{x}_i$.

Note that $(y_i - \hat{y}_i)$ captures the discrepancy between $y_i$ and $\hat{y}_i$, such that its value is 0 only when the true label and the predicted output match. Assume $x_{ij}$ is positive. If $\hat{y} = 0$ and $y = 1$, then $w_j$ is increased at the next iteration so that $\tilde{\mathbf{w}}^T\tilde{\mathbf{x}}_i$ can become positive. On the other hand, if $\hat{y} = 1$ and $y = 0$, then $w_j$ is decreased so that $\tilde{\mathbf{w}}^T\tilde{\mathbf{x}}_i$ can become negative. Hence, the weights are modified at every iteration to reduce the discrepancies between $\hat{y}$ and $y$ across all training instances. The learning rate $\lambda$, a parameter whose value is between 0 and 1, can be used to control the amount of adjustments made in each iteration. The algorithm halts when the average number of discrepancies are smaller than a threshold $\gamma$.

### 2.3.2 Multi-layer Neural Network

A multi-layer neural network generalizes the basic concept of a perceptron to more complex architectures of nodes that are capable of learning nonlinear decision boundaries. A generic architecture of a multi-layer neural network is shown in Figure 20 where the nodes are arranged in groups called layers. These layers are commonly organized in the form of a chain such that every layer operates on the outputs of its preceding layer. In this way, the layers represent different levels of *abstraction* that are applied on the input features in a sequential manner. The composition of these abstractions generates the final output at the last layer, which is used for making predictions. In the following, we briefly describe the three types of layers used in multi-layer neural networks.

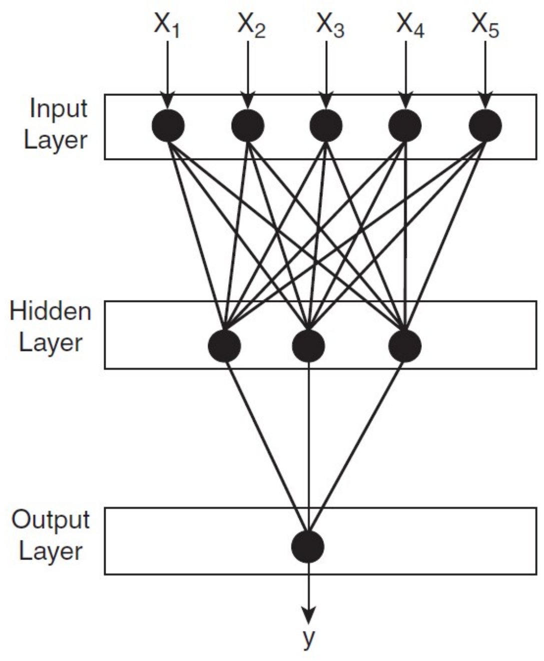

Figure 20: Example of a multi-layer artificial neural network (ANN)

The first layer of the network, called the **input layer**, is used for representing inputs from attributes. Every numerical or binary attribute is typically represented using a single node on this layer. These inputs are fed into intermediary layers known as **hidden layers**, which are made up of processing units known as hidden nodes. Every hidden node operates on signals received from the input nodes or hidden nodes at the preceding layer, and produces an activation value that is transmitted to the next layer. The final layer is called the **output layer** and processes the activation values from its preceding layer to produce predictions of output variables. For binary classification, the output layer contains a single node representing the binary class label. In this architecture, since the signals are propagated only in the forward direction from the input layer to the output layer, they are also called **feedforward neural networks**.

The major difference between multi-layer neural networks and perceptrons is the inclusion of hidden layers, which dramatically improves their ability to represent arbitrarily complex decision boundaries.

The hidden nodes can be viewed as learning latent representations or features that are useful for distinguishing between the classes. While the first hidden layer directly operates on the input attributes and thus captures simpler features, the subsequent hidden layers are able to combine them and construct more complex features. From this perspective, multi-layer neural networks learn a hierarchy of features at different levels of abstraction that are finally combined at the output nodes to make predictions. Further, there are combinatorially many ways we can combine the features learned at the hidden layers of ANN, making them highly expressive. This property chiefly distinguishes ANN from other classification models such as decision trees, which can learn partitions in the attribute space but are unable to combine them in exponential ways.

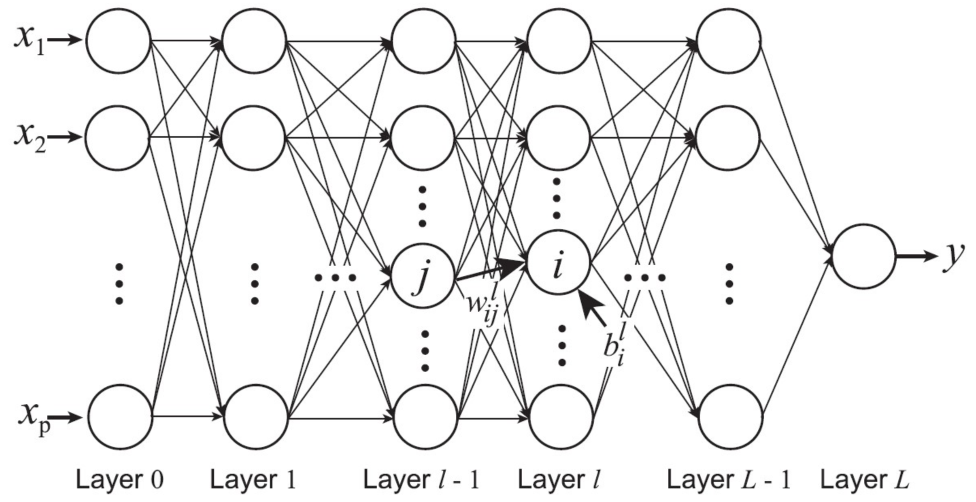

Figure 21: Schematic illustration of the parameters of an ANN model with $(L-1)$ hidden layers

To understand the nature of computations happening at the hidden and output nodes of ANN, consider the $i^{th}$ node at the $l^{th}$ layer of the network ($l > 0$), where the layers are numbered from 0 (input layer) to $L$ (output layer), as shown in Figure 21. The activation value generated at this node, $a_i^l$, can be represented as a function of the inputs received from nodes at the preceding layer. Let $w_{ij}^l$ represent the weight of the connection from the $j^{th}$ node at layer $(l-1)$ to the $i^{th}$ node at layer $l$. Similarly, let us denote the bias term at this node as $b_i^l$. The activation value $a_i^l$ can then be expressed as

$$a_i^l = f(z_i^l) = f\left(\sum_j w_{ij}^l a_j^{l-1} + b_i^l\right)$$

where $z$ is called the **linear predictor** and $f(\cdot)$ is the **activation function** that converts $z$ to $a$. Further, note that, by definition, $a_j^0 = x_j$ at the input layer and $a^L = \hat{y}$ at the output node.

There are a number of alternate activation functions apart from the sign function that can be used in multi-layer neural networks. Some examples include ReLU, sigmoid (logistic), and hyperbolic tangent functions. These functions are able to produce real-valued and nonlinear activation values. Among these activation functions, the sigmoid $\sigma(\cdot)$ has been widely used in many ANN models. We can thus represent $a_i^l$ as

$$a_i^l = \sigma(z_i^l) = \frac{1}{1+e^{-z_i^l}}$$

**Learning Model Parameters** The weights and bias terms $(\mathbf{w}, \mathbf{b})$ of the ANN model are learned during training so that the predictions on training instances match the true labels. This is achieved by using a loss function

$$E(\mathbf{w}, \mathbf{b}) = \sum_{k=1}^{n} \text{Loss}(y_k, \hat{y}_k)$$

where $y_k$ is the true label of the $k^{th}$ training instance and $\hat{y}_k$ is equal to $a^L$, produced by using $\mathbf{x}_k$. A typical choice of the loss function is the **squared loss function**:

$$\text{Loss}(y_k, \hat{y}_k) = (y_k - \hat{y}_k)^2$$

Note that $E(\mathbf{w}, \mathbf{b})$ is a function of the model parameters $(\mathbf{w}, \mathbf{b})$ because the output activation value $a^L$ depends on the weights and bias terms. We are interested in choosing $(\mathbf{w}, \mathbf{b})$ that minimizes the training loss $E(\mathbf{w}, \mathbf{b})$. Unfortunately, because of the use of hidden nodes with nonlinear activation functions, $E(\mathbf{w}, \mathbf{b})$ is not a convex function of $\mathbf{w}$ and $\mathbf{b}$, which means that $E(\mathbf{w}, \mathbf{b})$ can have **local minima** that are not globally optimal. However, we can still apply standard optimization techniques such as the **gradient descent** method to arrive at a locally optimal solution. In particular, the weight parameter $w_{ij}^l$ and the bias term $b_i^l$ can be iteratively updated using the following equations:

$$w_{ij}^l \leftarrow w_{ij}^l - \lambda \frac{\partial E}{\partial w_{ij}^l} \tag{1}$$

$$b_i^l \leftarrow b_i^l - \lambda \frac{\partial E}{\partial b_i^l} \tag{2}$$

where $\lambda$ is a hyper-parameter known as the learning rate. The intuition behind this equation is to move the weights in a direction that reduces the training loss. If we arrive at a minima using this procedure, the gradient of the training loss will be close to 0, eliminating the second term and resulting in the convergence of weights. The weights are commonly initialized with values drawn randomly from a Gaussian or a uniform distribution.

A necessary tool for updating weights is to compute the partial derivative of $E$ with respect to $w_{ij}^l$. This computation is nontrivial especially at hidden layers $(l < L)$, since $w_{ij}^l$ does not directly affect $\hat{y} = a^L$ (and hence the training loss), but has complex chains of influences via activation values at subsequent layers. To address this problem, a technique known as **backpropagation** was developed, which propagates the derivatives backward from the output layer to the hidden layers. This technique can be described as follows.

Recall that the training loss $E$ is simply the sum of individual losses at training instances. Hence the partial derivative of $E$ can be decomposed as a sum of partial derivatives of individual losses

$$\frac{\partial E}{\partial w_{ij}^l} = \sum_{k=1}^{n} \frac{\partial \text{Loss}(y_k, \hat{y}_k)}{\partial w_{ij}^l}$$

To simplify discussions, we will consider only the derivatives of the loss at the $k^{th}$ training instance, which will be generically represented as $\text{Loss}(y, a^L)$. By using the chain rule of differentiation, we can represent the partial derivatives of the loss with respect to $w_{ij}^l$ as

$$\frac{\partial \text{Loss}}{\partial w_{ij}^l} = \frac{\partial \text{Loss}}{\partial a_i^l} \cdot \frac{\partial a_i^l}{\partial z_i^l} \cdot \frac{\partial z_i^l}{\partial w_{ij}^l} \tag{3}$$

The last term of the Equation 3 can be written as

$$\frac{\partial z_i^l}{\partial w_{ij}^l} = \frac{\partial(\sum_j w_{ij}^l a_j^{l-1} + b_i^l)}{\partial w_{ij}^l} = a_j^{l-1}.$$

Also, if we use the sigmoid activation function, then

$$\frac{\partial a_i^l}{\partial z_i^l} = \frac{\partial \sigma(z_i^l)}{\partial z_i^l} = a_i^l(1 - a_i^l).$$

So we can simplify Equation 3 as

$$\frac{\partial \text{Loss}}{\partial w_{ij}^l} = \delta_i^l \cdot a_i^l(1-a_i^l) \cdot a_j^{l-1} \tag{4}$$

$$\text{where } \delta_i^l = \frac{\partial \text{Loss}}{\partial a_i^l}.$$

A similar formula for the partial derivatives with respect to the bias terms $b_i^l$ is given by

$$\frac{\partial \text{Loss}}{\partial b_i^l} = \frac{\partial \text{Loss}}{\partial a_i^l} \cdot \frac{\partial a_i^l}{\partial z_i^l} \cdot \frac{\partial z_i^l}{\partial b_i^l} = \delta_i^l \cdot a_i^l(1-a_i^l). \tag{5}$$

Hence, to compute the partial derivatives, we only need to determine $\delta_i^l$. Using a squared loss function, we can easily write $\delta^L$ at the output node as

$$\delta^L = \frac{\partial \text{Loss}}{\partial a^L} = \frac{\partial (y-a^L)^2}{\partial a^L} = 2(a^L - y)$$

However, the approach for computing $\delta_j^l$ at hidden nodes ($l < L$) is more tricky. Notice that $a_j^l$ affects the activation values $a_i^{l+1}$ of all nodes at the next layer, which in turn influences the loss. Hence, again using the chain rule of differentiation, $\delta_j^l$ can be represented as

$$\begin{aligned}
\delta_j^l = \frac{\partial \text{Loss}}{\partial a_j^l} &= \sum_i \left( \frac{\partial \text{Loss}}{\partial a_i^{l+1}} \cdot \frac{\partial a_i^{l+1}}{\partial a_j^l} \right) \\
&= \sum_i \left( \frac{\partial \text{Loss}}{\partial a_i^{l+1}} \cdot \frac{\partial a_i^{l+1}}{\partial z_i^{l+1}} \cdot \frac{\partial z_i^{l+1}}{\partial a_j^l} \right) \\
&= \sum_i \left( \delta_i^{l+1} \cdot \frac{\partial a_i^{l+1}}{\partial z_i^{l+1}} \cdot w_{ij}^{l+1} \right)
\end{aligned} \tag{6}$$

The previous equation provides a concise representation of the $\delta_j^l$ values at layer $l$ in terms of the $\delta_i^{l+1}$ values computed at layer $l+1$. Hence, proceeding backward from the output layer $L$ to the hidden layers, we can recursively apply Equation 6 to compute $\delta_i^l$ at every hidden node. $\delta_i^l$ can then be used in Equations 4 and 5 to compute the partial derivatives of the loss with respect to $w_{ij}^l$ and $b_i^l$, respectively.

For our purpose, that is the classification of ECG, now we will see an example of application of the method just explained. It will be used the first row of the INCART dataset to observe a simple application of ANN's algorithm. For simplicity we consider only the first value of the RR INTERVAL attribute (shown in the Table 4) for representing *D.train*, the set of training instances.

| CLASS | RR INTERVAL |
|---|---|
| 1 | 163 |

Table 4: Subset of INCART dataset

In the following computation the loss function will be the **cross entropy**. The cross entropy loss function of a real-valued prediction $\hat{y} \in (0,1)$ on a data instance with binary label $y \in \{0,1\}$ can be defined as

$$\text{Loss}(y, \hat{y}) = -y \log \hat{y} - (1-y)\log(1-\hat{y})$$

where log represents the natural logarithm (to base $e$) and $0\log(0) = 0$ for convenience. The partial derivative of this loss function with respect to $\hat{y} = a^L$ can be given as

$$\delta^L = \frac{\partial \text{Loss}}{\partial a^L} = -\frac{y}{a^L} + \frac{1-y}{1-a^L} = \frac{a^L - y}{a^L(1-a^L)}.$$

Using it in Equation 4, we can obtain the partial derivative of the loss at the output node as

$$\frac{\partial \text{Loss}}{\partial w_j^L} = \frac{a^L - y}{a^L(1-a^L)} \cdot a^L(1-a^L) \cdot a_j^{L-1} = (a^L - y) \cdot a_j^{L-1}$$

$$\frac{\partial \text{Loss}}{\partial b^L} = \frac{a^L - y}{a^L(1-a^L)} \cdot a^L(1-a^L) \cdot = (a^L - y)$$

Notice the simplicity of the previous formulas using the cross entropy loss function. The partial derivatives of the loss with respect to the weights at the output node depend only on the difference between the prediction $a^L$ and the true label $y$. Hence, the gradients are high whenever $(a^L - y)$ is large, promoting effective learning of the model parameters at the output node. This has been a major breakthrough in the learning of modern ANN models and it is now a common practice to use the cross entropy loss function with sigmoid activations at the output node.

First of all we need to set the architecture of the ANN. We will use a very simple structure: the input layer contains only one node in which is set the value 163, the hidden layer contains two nodes and the output layer contains one binary prediction. Then we must initialize the weights and the biases. Usually weights are initialized randomly using a uniform distribution over a small range, say -1 to 1 (there are also other techniques), while biases are usually set to zero.

$$\mathbf{w} = (w_1^1, w_2^1, w_1^2, w_2^2) = (-0.34, -0.14, -0.38, 0.8)$$

$$\mathbf{b} = (b_1^1, b_2^1, b^2) = (0, 0, 0)$$

The activation functions will be ReLU for internal node and Sigmoid for the output one:

$$\text{ReLU}(x) = \max\{0, x\} = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

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

For each node, the schema shown in Figure 22 must be followed: weights must be multiplied, while biases must be added.

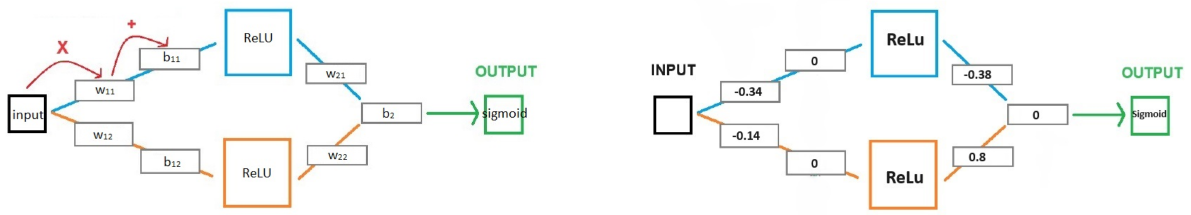

Figure 22: A schema starting from one input

Let's start considering the RR INTERVAL column and the input value $x = 163$. The two hidden nodes are denoted as $a_1^1$ and $a_2^1$. The output of the network, $a^2 = \hat{y}$, is computed by the Sigmoid function, whose codomain is in $(0, 1)$. It can be therefore interpreted as an estimate of the probability that a given sample belongs to the class 'N'.

$$y = \mathbb{P}(\text{CLASS} = \text{N} | \text{RR INTERVAL} = 163) = 1$$

The feedforward pass gives the following results:

$$a_1^1 = \text{ReLU}(xw_1^1 + b_1^1) = \text{ReLU}(163 \cdot -0.34 + 0) = 0$$

$$a_2^1 = \text{ReLU}(xw_2^1 + b_2^1) = \text{ReLU}(163 \cdot -0.14 + 0) = 0$$

$$a^2 = \hat{y} = \sigma\left(\sum_i w_i^2 a_i^1 + b^2\right) = \sigma(-0.38 \cdot 0 + 0.8 \cdot 0 + 0) = \sigma(0) = 0.5$$

The true output is 1, since the CLASS = 1, so the result 0.5 is not satisfying. Indeed, the cross entropy of 0.5 is 0.69, then we must use the backpropagation to minimize this value, i.e. bring it closer to 0. For this reason the cross entropy (C) partial derivatives must be computed. The input, hidden and output layers are labeled with 0,1,2 respectively.

$$\frac{\partial \mathrm{C}}{\partial b^2} = a^2 - y = \hat{y} - y = 0.5 - 1 = -0.5$$

$$\frac{\partial \mathrm{C}}{\partial b_1^1} = \delta_1^1 \cdot a_1^1(1 - a_1^1) = \delta^2 a^2(1 - a^2) w_1^2 = \frac{\partial C}{\partial b^2} w_1^2 = -0.5 \cdot (-0.38) = 0.19$$

$$\frac{\partial \mathrm{C}}{\partial b_2^1} = \frac{\partial C}{\partial b^2} w_2^2 = -0.5 \cdot 0.8 = -0.4$$

$$\frac{\partial \mathrm{C}}{\partial w_1^2} = (a^2 - y) a_1^1 = (\hat{y} - y) a_1^1 = -0.5 \cdot 0 = 0$$

$$\frac{\partial \mathrm{C}}{\partial w_2^2} = (\hat{y} - y) a_2^1 = -0.5 \cdot 0 = 0$$

$$\frac{\partial \mathrm{C}}{\partial w_1^1} = \delta_1^1 \cdot a_1^1(1 - a_1^1) \cdot a^0 = \frac{\partial C}{\partial b_1^1} \cdot x = 0.19 \cdot 163 = 30.97$$

$$\frac{\partial \mathrm{C}}{\partial w_2^1} = \frac{\partial C}{\partial b_2^1} \cdot x = -0.4 \cdot 163 = -65.2$$

Substituting this values in the Equations 4 and 5, where $\lambda = 0.01$ we obtain:

$$\mathbf{w}^{(1)} = (-0, 65, 0, 51, -0.38, 0.8)$$

$$\mathbf{b}^{(1)} = (-0.0019, 0.004, -0.0019)$$

The same computations must be repeated until weights and biases converge.

## 3 Algorithm Presentation

In this Section all the algorithms explained in Section 2 are written explicitly in the form of pseudocodes.

### 3.1 $k$NN Algorithm

A summary of the nearest neighbors classification is given in Algorithm 1. It computes the distance (or similarity) between each test instance $z = (\mathbf{x}', y')$ and all the training examples $(\mathbf{x}, y) \in D$ to determine its nearest neighbors list, $D_z$.

**Algorithm 1:** The $k$-nearest neighbor classifier

```
Data: $k$ = number of nearest neighbors, $D$ = set of training axamples
for each test instance $z = (\boldsymbol{x}', y')$ do
    Compute $d(\mathbf{x}', \mathbf{x})$, the distance between $z$ and every example, $(\mathbf{x}, y) \in D$.
    Select $D_z \subseteq D$, the set fo $k$ closest neighbors of $z$.
    $y' = \arg\max_c \sum_{(\mathbf{x}_i, y_i) \in D_z} I(c = y_i)$.
end
```

Once the nearest neighbor list is obtained, the test instance is classified based on the majority class of its neighbors:

$$\text{Majority Voting: } y' = \arg\max_c \sum_{(\mathbf{x}_i, y_i) \in D_z} I(c = y_i)$$

where $c$ is a class label, $y_i$ is the class label of one of the nearest neighbors and $I(\cdot)$ is an indicator function that returns the value 1 if its argument is true and 0 otherwise.

In the majority voting approach, every neighbor has the same impact on the classification. This makes the algorithm sensitive to the choice of $k$. One way to reduce the impact of $k$ is to weight the influence of each nearest neighbor $x_i$ according to its distance: $w_i = 1/d(\mathbf{x}', \mathbf{x}_i)^2$. As a result, training examples that are located far away from $z$ have a weaker impact on the classification compared to those that are located close to $z$. Using the distance-weighted voting scheme, the class label can be determined as follows:

$$\text{Distance-Weighted Voting: } y' = \arg\max_c \sum_{(\mathbf{x}_i, y_i) \in D_z} w_i \cdot I(c = y_i)$$

### 3.2 Decision Tree Algorithm

Algorithm 2 presents a pseudocode for decision tree induction algorithm. The input to this algorithm is a set of training instances $S$ along with the attribute set $F$. The algorithm works by recursively selecting the best attribute to split the data and expanding the nodes of the tree until the stopping criterion is met. The details of this algorithm are explained below.

1. The CREATENODE() function extends the decision tree by creating a new node. A node in the decision tree either has a test condition, denoted as *node.test_cond*, or a class label, denoted as *node.label*.

2. The FINDBESTSSPLIT() function determines the attribute test condition for partitioning the training instances associated with a node. The splitting attribute chosen depends on the impurity measure used. The popular measures include entropy and the Gini index.

3. The CLASSIFY() function determines the class label to be assigned to a leaf node. For each leaf node $t$, let $p(i|t)$ denote the fraction of training instances from class $i$ associated with the node $t$. The label assigned to the leaf node is typically the one that occurs most frequently in the instances that are associated with this node:
$$leaf.label = \arg\max_i p(i|t),$$
where the argmax operator returns the class $i$ that maximizes $p(i|t)$. Besides providing the information needed to determine the class label of a leaf node, $p(i|t)$ can also be used as a rough estimate of the probability that an instance assigned to the leaf node $t$ belongs to class $i$.

4. The STOPPINGCOND() function is used to terminate the tree-growing process by checking whether all the instances have identical class label or attribute values. Since decision tree classifiers employ a top-down, recursive partitioning approach for building a model, the number of training instances associated with a node decreases as the depth of the tree increases. As a result, a leaf node may contain too few training instances to make a statistically significant decision about its class label. This is known as the data fragmentation problem. One way to avoid this problem is to disallow splitting of a node when the number of instances associated with the node fall below a certain threshold.

**Algorithm 2:** A decision tree induction algorithm

```
TREEGROWTH($S, F$)
if STOPPINGCOND($S, F$) = True then
    $leaf$ = CREATENODE()
    $leaf.label$ = CLASSIFY($S$)
    return $leaf$
    else
        $root$ = CRATENODE()
        $root.test\_cond$ = FINDBESTSPLIT($S, F$)
        $V = \{v | v$ is a possible outcome of $root.test\_cond\}$
        for each $v \in V$ do
            $S_v = \{s \in S | root.test\_cond(s) = v\}$
            $child$ = TREEGROWTH($S_v, F$)
            add child as descendent of $root$ and label the edge ($root \rightarrow child$) as $v$
        end
    end
end
return $root$
```

## 3.3 Random Forest Algorithm

Algorithm 3 presents a pseudocode for random forests. A training set $S = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$, a set of attribute $F$ and a number $B$ are given in input. $B$ represents the number of trees in a forest.

**Algorithm 3:** Random Forest algorithm

```
RANDOMFOREST($S$,$F$):
$H = 0$
for $i \in 1, ...B$ do
    $S^{(i)}$ = BOOTSTRAP($S$)
    $h_i$ = RANDOMIZEDTREELEARN($S^{(i)}$,$F$)
    $H = H \cup \{h_i\}$
end
return $H$
-
RANDOMIZEDTREELEARN($S, F$):
for each node do
    $f$ = very small subset of $F$, randomly chosen
    TREEGROWTH($S, f$)
end
```

## 3.4 ANN Algorithm

Algorithm 4 summarizes the complete approach for learning the model parameters of ANN using backpropagation and gradient descent method.

**Algorithm 4:** Learning ANN using backpropagation and gradient descent

```
$D.train = \{(\mathbf{x}_k, y_k) | i = 1, ..., n\}$ set of training instances
$c = 0$
Initialize the weight and bias terms $(\mathbf{w}^{(0)}, \mathbf{b}^{(0)})$ with random values
**repeat**
    **for** $(\boldsymbol{x}_k, y_k) \in D.train$ **do**
        Compute the set of activations $(a_i^l)_k$ by making the feedforward step using $\mathbf{x}_k$
        Compute the set $(\delta_i^l)_k$ by propagation using Equations 4 and 5
    **end**
    Compute $\partial E/\partial w_{ij}^l \leftarrow \sum_{k=1}^{n} (\partial \text{Loss}/\partial w_{ij}^l)_k$
    Compute $\partial E/\partial b_i^l \leftarrow \sum_{k=1}^{n} (\partial \text{Loss}/\partial b_i^l)_k$
    Update $(\mathbf{w}^{(c+1)}, \mathbf{b}^{(c+1)})$ by gradient descent using Equations:

        $$w_{ij}^l \leftarrow w_{ij}^l - \lambda \frac{\partial E}{\partial w_{ij}^l}$$

        $$b_i^l \leftarrow b_i^l - \lambda \frac{\partial E}{\partial b_i^l}$$

    $c \leftarrow c + 1$
**until** $(\boldsymbol{w}^{(c+1)}, \boldsymbol{b}^{(c+1)})$ *and* $(\boldsymbol{w}^{(c)}, \boldsymbol{b}^{(c)})$ *converge to the same value*;
```

# 4 Algorithm Analysis

In this section complexity analysis of the three algorithms will be discussed.

## 4.1 $k$-Nearest Neighbors Algorithm Analysis

As we shown in Subsection 2.1 basic $k$NN algorithm is very simple, but if the training set has a large number of samples, it can represent a problem. Using $O$-notation, we can describe the running time of the $k$NN algorithm merely by inspecting the algorithm's overall structure.

Suppose there are $n$ training examples and $m$ test example, each of dimension $d$. For computing the Euclidean distance between one one test sample and all the training ones the complexity is $O(n \cdot d)$, because it is needed an elementwise subtraction (with complexity $O(d)$) and the computation of the norm (with complexity $O(d)$) for every $n$ instances:

$$n \cdot (O(d) + O(d)) = O(n \cdot d).$$

All the computations must be done for each of the $m$ test element. So the time complexity is $O(n \cdot d \cdot m)$

The index must be ordered with respect to the distance with a complexity of $O(n \cdot \log n)$. Then the $k$ nearest neighbors are found in $O(k)$ time, exactly like the extraction of corresponding labels.

The computation of the most common label takes $O(k \cdot \log k)$ in the worst case due to the necessity of an ordering to determine the unique occurrences.

Finally, append a predicted class to a list has a constant complexity $O(1)$.

Summarizing we get:

$$O(n \cdot d \cdot m \cdot \log n \cdot k \cdot \log k)$$

As the number of features or the size of the dataset increases, the computational complexity of $k$NN also increases significantly, making it computationally expensive and impractical for large datasets. Additionally, finding the optimal value of $k$ by brute force can also increase the time complexity of the algorithm. Hence, it is important to improve the algorithm with techniques that reduce in some ways the computations.

## 4.2 Random Forest Algorithm Analysis

Before analyzing the time complexity of the Random Forest algorithm, it is necessary to take in account the Decision Tree and analyze its running time. Consider again that the training set contains $n$ samples and $d$ attributes.

What happens in the training stage is that for each of the attributes in the dataset we'll sort the data which takes $O(n \log n)$ time. This is due to the fact that sorting algorithm splits in two equal parts the dataset in a recursive way: redo the split until the dataset contains only one instance. It means that the input size is continuously divided by 2 (at the step $k$ the size will be $n/2^k$). If $n/2^k = 1$, then the number of steps is $k = \log n$ (considering the base 2 algorithm). Since at every split the problem to solve is duplicated, then we can consider the input size equal to $n$ at each recursion (e.g.: at first step $2 \cdot O(n/2) = O(n)$, at second step $4 \cdot O(n/4) = O(n)$). Hence, the time complexity is $O(n) \cdot O(\log n) = O(n \cdot \log n)$.

Then we traverse the data points to find the right threshold. For each value the Gini index is computed. the list of labels is converted into an array and this operation is $O(n)$, then a function counts the occurrences of each integer value in the array. It traverses the array so it is $O(n)$. If $k$ labels have been found, then finding the percentages is $O(k)$. Also computing the Gini index has tha same time complexity since it involves a loop on the $k$ percentages. The overall complexity is:

$$O(n) + O(n) + O(k) + O(k) = O(n + k)$$

Then we traverse the data points to find the right threshold which takes $O(n)$ time, for $d$ dimensions. The total time complexity would be

$$O(d \cdot n \cdot \log n) + O(d \cdot n) = O(d \cdot n \cdot \log n).$$

In typical scenarios where the range of values $k$ is much smaller than the size of the input $n$, the function runs in $O(n)$ time. However, if $k$ is very large (e.g., if the input array has a wide range of integer values), $O(k)$ might become the dominant factor.

After the Gini index we compute the gain. For computing the weights it is needed $O(1)$ for each weights. Let's denote $n_p$ the length of parent node, $n_l$ the length of left child, $n_r$ the length of right child, $k_p$ the the number of class labels of parent node, $k_l$ the number of class labels of left child, $k_r$ the number of class labels of right child. The Gini index for the parent and the child nodes are respectively $O(n_p + k_p)$, $O(n_l + k_l)$, $O(n_r + k_r)$. Because parent is the superset, $n_p$ is the largest, and typically, the ranges of values $k$'s are not more significant than the lengths of the lists $n$'s. So, we can generalize:

$$O(n_p + k_p) + O(n_l + k_l) + O(n_r + k_r) = O(n_p + k_p)$$

For finding the best split two loops must be done: one on the number of columns and one on the unique values of the current column. The first is $O(n\_col)$ and the second is $O(n\_rows \cdot \log n\_rows)$ (in the worst case). For constructing the left and right dataset first we must concatenate the features and the target. It is a $O(n\_rows \cdot n\_col)$ operation. The same time complexity is needed for splitting the dataset with respect to the threshold. The last step is computing the gain, that for each calculation is $O(n\_rows + k)$. The we get:

$$O(n\_cols \cdot n\_rows \cdot \log n\_rows \cdot (n\_rows \cdot n\_col + n\_rows + k))$$

Supposing that $k$ is irrelevant with respect to $n\_cols$ we can simplify:

$$O(n\_cols^2 \cdot n\_rows^2 \cdot \log n\_rows)$$

The building step has the same complexity.

The prediction function is recursive and the base case is $O(1)$. If the tree is balanced, the recursion has a time complexity of $O(\log n)$, because of the binary split. While if it is unbalanced it is $O(max\_depth)$, where $max\_depth$ is the depth of the tree.

Now, let's consider the Random Forest algorithm. If we choose to fit $m$ decision tree classifiers, then the time complexity will be the previous one, multiplied by $m$. But we must take in account that $n\_cols$ is reduced to $f$, the number of features randomly chosen at each step. It must be clarify that a randomic choice has a linear time complexity with respect to the input size. For example, bootstrap sample is $O(n)$ since it must sample a dataset of $n$ elements.

Finally, the prediction step is $O(n\_estimators) \cdot \log n$, if we consider $n$ the length of the test set. For computing the majority vote it is needed an $O(n \cdot n\_estimators)$ time, since counting and finding the most common prediction involves iterating over the array of predictions.

## 4.3 Artificial Neural Network Algorithm analysis

In a fully connected Neural Network each node of every layer is connected to all the nodes present in the next layer.

Suppose the input layer $l_0$ has $d_0$ nodes (as many as the features in the dataset). Denote the hidden layers as $l_i$ for $i = 1, ..., L-1$ and the output layer as $l_L$. Each hidden layer $l_i$ have size $d_i$, while the output layer has size 1, since it contains a number in $(1, 0)$ that is the probability of the input to be classified as normal beat.

The time complexity for the initialization step depends on the number of parameters.

- Initializing weights between the input layer and hidden layer: $O(d_0 \cdot d_1)$
- Initializing weights between two hidden layers $l_i, l_{i+1}$: $O(d_i \cdot d_{i+1})$
- Initializing weights between the hidden layer and output layer: $O(d_{L-1})$
- Initializing biases for the hidden layer $l_i$: $O(d_i)$
- Initializing biases for the output layer: $O(1)$

So, the overall time complexity for initialization is:

$$O\left(\sum_{i=0}^{L-2} d_i \cdot d_{i+1} + d_{L-1}\right)$$

The forward pass calculates the output array of the neural network. The hidden layer activations used in the algorithm is the ReLU function, while the output activation is the Sigmoid. Both are $O(1)$ function if the input is a scalar or $O(s)$ if the size input is $s$. While for computing $z_i^l$, the output of the node $i$ of layer $L$ it is compute an inner product summed with a scalar.

- Output of a node belonging to the layer $l_i$ : $O(d_{i-1} + 1)$
- Layer $l_i$ activations: $O(d_i)$

The overall time complexity of the forward step is:

$$O\left(\sum_{i=1}^{L} (d_i \cdot d_{i-1} + d_i)\right)$$

The backward pass makes use of the function delta, that calculates the derivatives of the loss function with respect to the activations (in Subsection 2.3.2 it is denoted as $\partial \text{Loss}/\partial a_i^l$). Thus this function does operations for each nodes belonging to hidden and output layers. After using delta function, the backward step computes the partial derivatives of the loss function with respect weights and biases and updates the parameters.

- Delta function for the node in the output layer: $O(1)$
- Delta function for a node of hidden layer $l_i$: $O(d_{i+1})$
- Derivative of the loss function with respect to the weights: $O\left(\sum_{i=0}^{L-1} d_i \cdot d_{i+1}\right)$
- Derivative of the loss function with respect to the biases: $O\left(\sum_{i=1}^{L} d_i\right)$

The overall time complexity of the backward step is:

$$O\left(\left[\sum_{i=0}^{L-1} d_i \cdot d_{i+1}\right] \cdot (d_{i+1} + 1) + \left[\sum_{i=1}^{L} d_i\right] (d_{i+1} + 1)\right) = O\left(d_{i+1} \cdot \left[\sum_{i=0}^{L-1} d_i \cdot d_{i+1} + \sum_{i=1}^{L} d_i\right]\right)$$

The training method performs the forward and the backward passes for each epoch, say $e$, and for each instance of the training dataset, say *n_train*. Then the time complexities of the two functions must be multiplied by $n \cdot e$.

The prediction involves only the forward step and it is repeated for each row of the test dataset, that has size *n_test*.

## 5 Conclusions

The classification of ECG is an hard, but at the same time really importat, task. In this paper I've tried to explain how it can be solved, using three kind of algorithms. I've chosen two Machine Learning algorithms (one is an ensamble method and the other is not) and one Deep Learning algorithm, for showing differences in time complexity and accuracy.

The $k$-Nearest Neighbors algorithm would seem too much simple, and it is, but it has the major explainability. Indeed, if two vectors are "near" (with respect to a certain distance definition) they have very similar characteristics. In medical field this algorithm is used a lot when the dataset involves measurements that must be compared. For this reason it's not surprising that it works very well. The simulation with the subset of INCART dataset shows an accuracy of 97% with only 4 errors out of 150 test instances, using $k = 3$. Furthermore, it takes a small amount of time.

The Random Forest algorithm shows a similar accuracy. I've run it three times to see how much the random choice of the 5 attributes during each split influence the result. The accuracies are 97%, 98%, 98%. It took about 30 seconds, while $k$-nearest neighbors a fraction of a second.

The Artificial Neural Network, despite it is the most complex of the three algorithms, has the lowest accuracy: 85%, if we use 3 hidden layer with 2, 3, 4 nodes respectively and a learning rate of 0.001. The problem is that the best hyperparameters must be found. It means that we don't know a-priori values for the number of layers and the number of nodes inside them or for the learning rate, but many algorithms can be used for solving this task.

Finally, the question is: what is the best algorithm? The answer unfortunately is always: it depends. The last models is the most efficient (if it is improved with algorithms for choosing hyperparameters) but it requires much more memory than the other. Indeed, at every epoch it must store all the weights, the biases, the output layers and the activation values. In the literature there can be find neural networks with 1000 epochs or more and, if our dataset is big, it is necessary to have a machine with a computing power very high. Also Random forest requires a big amount of memory (less then ANN) to find the best split. Indeed, it must be stored every gain result for each threshold value and for each column. The less memory is used by the $k$NN algorithm, that must store $n$ distances at every step. At the same time, also this algorithm needs methods to find the best parameter $k$. For all this reasons, it is a good practice to try many models and choose the best.