0.1. Fault Detection and Diagnosis

Fault detection and diagnosis (FDD) arises from the need to facilitate system recoveries after agent faults, increase the availability of the system, and reduce potential losses of resources. In the domain of multirobot system, where agents interact with each other or even with human agents, faults have the potential to threaten the safety of the robot system itself and its surroundings [?]. The prompt detection and diagnosis of faults allows the controller to plan recovery procedures or to re-plan the task execution, in this situation we can imagine that the controller can be a human agent that retrieves the faulty agent and replaces it, or it can be a software controller that schedules a re-plan or interrupt the execution of the task.

FDD in swarm robotics spaces over different levels of analysis: it is possible to examine it from the poitn of view of every single agent and all its components or as the whole swarm in its entirety. The difference in what to consider detection and what diagnosis can differ from author to author. Some authors use diagnosis to refer to component level fault detection while others refer to single agent level fault detection. In this work, we consider diagnosis the identification of the kind of fault or the component not working in the single-agent while detection is the identification of the faulty agent.

The difficulty of detecting faults may change from depending on the kind of anomaly or task considered. For example, detecting a robot that is not communicating with its neighbors but behaves correctly may be impossible from an external agent. On the other hand, if a robot battery fails, this anomaly is quite easy to detect because it would imply a complete physical stop of the agent, a detachment from the swarm, and an interruption of communication with an external controller if there is one.

FDD approaches can be partitioned into three subcategories: knowledge-based, model-based, and data-driven [?]. Knowledge-based methods use previously known faults and behaviors to recognize predefined known anomalies and diagnoses. Model-based methods use knowledge of the internal system functioning to simulate the agent behavior and detect any anomaly separating from the nominal functioning [?]. The data-driven approach exploits the advantage of being model-free and not requiring any previous knowledge of the system's internal functioning; these techniques are founded on statistical and machine learning procedures to differentiate non-nominal behaviors from nominal observed ones.

0.1.1. Exogenous and Endogenous Fault Detection

The task of fault detection in multirobot systems can be partitioned into exogenous and endogenous anomaly detection. Endogenous fault detection refers to the task of identifying anomalies from the point of view of the agent itself. It can be achieved by performing internal analysis and can be done using model-based prediction techniques [?] or local neural networks [?]. This kind of technique presents some advantages on the diagnostic level but becomes challenging whenever a misleading diagnosis occurs. If an agent does not diagnose a fault in itself it will not communicate it to the other robots or, even worse, if the communication module of an agent breaks, any failure message will not be sent to its neighbors [?].

Exogenous fault detection refers to the ability of robots to detect faults in one another.

Harzard	Description
H_1	Motor failure
H_2	Communication failure
H_3	Avoidance sensor(s) failure
H_4	Beacon sensor failure
H_5	Control system failure
H_6	All system failure

Table 1: Internal hazards for a single bot.

This approach has been proposed by Christensen et al. in [?] where they tried to identify faults in a system of robots synchronizing in the task of led flashing. Christensen et al. proposed a fault detection approach that does not require communication from one agent to another but relies solely on the flashing led, assuming that a robot not flashing synchronously is not working properly. In a broader sense, our approach could be classified as exogenous fault detection since we rely on an external controller that observes the swarm and classifies each agent based on its behavior.

0.1.2. Faults

In a generic operational environment, we can find two kinds of errors: random errors and systematic (design) errors [?]. Random errors are defined as those errors caused by hardware or component faults, they usually can be mitigated by employing high reliability or redundant components. By the definition of swarm, these systems exhibit a high level of redundancy and tolerance failure by construction so random errors are not the most concerning ones. Instead, systematic errors are anomalies that can lead the system to manifest undesirable behaviors. These kinds of faults require two levels of analysis: individual agent level and whole swarm level.

Winfield and Nembrini [?] list a series of possible internal hazards that can appear in individual robots. Table 1 lists a group of possible individual robot hazards. Hazard H_1, motor failure, covers the condition of a mechanical failure in the wheels actuation motors making the robot unable to move at all or move only on the spot. This kind of fault can either be trivial or not depending on the task under execution. If we take into consideration the task of flocking, explained in Section ??, a single faulty bot can have two consequences: it can be avoided by all the other agents and not be harmful, or it can anchor other robots with itself compromising the completion of the task.

Hazard H_2, communications failure, consists in the malfunctioning of one or a small number of mobile robots communications network subsystems. This fault implies the disconnection of one agent or more from the swarm. Communication failure is not trivial to detect since the behavior of the robot deeply depends on the kind of task under execution. This fault does not pose any harm in a task like flocking which does not require the use of any communication procedure. Conversely, in a task like foraging, explained in Section ??, if an agent is unable to communicate with its neighbors, two consequences might arise: the agent might be unable to know whether to keep on exploring, or the agent might be unable to communicate to its neighbors to continue in the execution of the task.

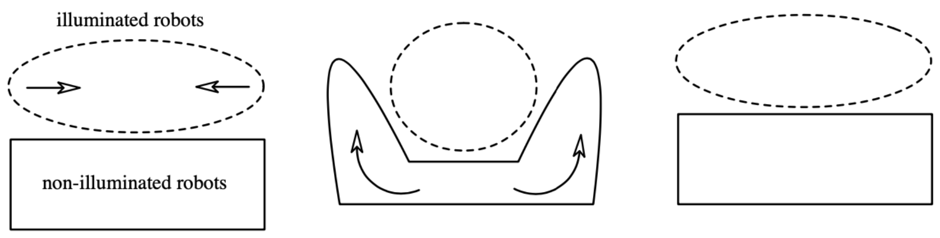


Figure 1: Emergent taxis behavior illustration.
The figure on the left depicts a set of robots with the role of beacon taxis, namely illuminated robots. In the center we can see the rest of the robots, namely non-illuminated robots, that moves towards the light emitted from the beacon of the robots in front of them. The continuous combination of these two movements constitutes the emergent taxis behavior.

Hazard H_3, avoidance sensor(s) failure, intuitively identifies the malfunctioning of one or more avoidance sensors. This fault is not harmful in most experimental environments. A single robot with failed avoidance sensors will be avoided by all the other robots. If two robots present avoidance sensor(s) malfunctioning they might collide resulting in physical damage but the swarm behavior should not be affected. This fault could manifest some undesirable effects if the environment includes some human agents.
Hazard H_4, beacon sensor failure, refers to the malfunctioning of the sensor in charge of identifying other agents' beacons. As the communication failure hazard, the beacon sensor failure deeply depends on the kind of task in execution and if it involves at all the use of the beacon. In [?] the use of the beacon represents a fundamental part of the task execution. In the flocking task, the beacon is necessary to identify other agents and keep the flock formation. In the encapsulation task, the behavior of the emergent taxi (illustrated on Figure 1 taken from[?]) has little or no effect at all from the failure of the beacon sensor.
Hazard H_5, control system failure, is difficult to characterize and to identify. With control system failure we identify a malfunctioning in the control software, this can induce the agent to abnormal movements like indefinitely going forward or turning on the spot. In the first scenario, the robot would drift away and detach from the swarm posing no harm. In the second scenario, if this fault manifests while being the beacon taxis, the faulty agent would deceive the perceptions of its neighbors inducing them to execute wrong actions. For example, imagine a robot with the role of beacon taxis but not aggregating with the other beacon taxis robots due to control system failure, some of the agents that are not beacon taxis will start moving towards the faulty agent thus corrupting the emergent taxis behaviour.
Hazard H_6, total system failure, will render the robot stationary and inactive. While this fault is the most serious, as counterintuitive it might be, it is instead the most benign. In this situation, the malfunctioning agent would simply be treated as a static obstacle.
To better understand the preceding faults we have illustrated the physical functioning of the robots and their components in Section **??**.

0.1.3. Features

In this work, by feature we intend the measurable properties or characteristic of the phenomenon under analysis. Features have the role of embedding the object under consideration in an d-dimensional space with values that can be numerical or categorical. In this paragraph, we will show the main sources and the main motivations behind the choice of each feature. The complete list of features we have used can be seen in Table 2.
From [?], we can see that the coordinates value features together with the robot speed has been used to detect physical and logic faults in two robots pushing a box.
Khaldi et al. [?] introduces the right and left wheel speed, the average mean distance error (AMDE), and the group speed to monitor a virtual viscoelastic model (VVC). They have inspired the overall speed of the agent, the neighbors' average distance, and the swarm speed, respectively.
Neighbors count has been used in [?] for a decentralized fault detection system.
Wei et al. [?] introduces the metrics for flocking behavior. Among them, we can find:

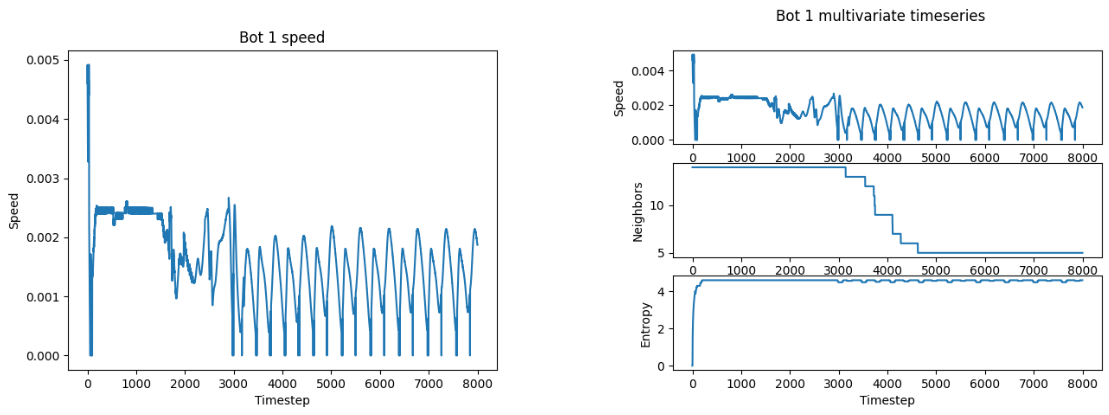
- " The agents of the swarm should always face approximately the same direction ", that suggested we analyze the direction of each agent.
- " The agent should remain close to each other ", meaning the average distance from all the other agents can enclose insightful information.

Khalastchi et al. [?] present an online data-driven anomaly detection approach with a sliding window technique to meet the challenge of detecting dynamically correlated attributes. This approach has led us to consider cumulative features like cumulative speed and cumulative centroid distance.
The objective of our work can be interpreted from two points of view: the view of a summary, a collection of ideas on how to interpret data collected from the agents and how to elaborate the data with complex features; the view of an experiment, an analysis of performances on the usage of the computed features. This approach aims at lightening the work on the search for the new features to compute, giving inspiration on new features, and informing the reader of which features might be avoided to reduce computational time. Nevertheless, we underline that features performances are deeply dependent on the scenario in which they are considered and, as with all the data-driven approaches, they might or might not influence the final results.

0.2. Machine Learning Techniques for Fault Detection

0.2.1. Data-Driven Fault Detection

Data-driven approaches can be divided into statistical approaches and machine learning approaches [?]. Kalman filters [?] or particle filters [?] are examples of statistical techniques used to filter statistical noise. These approaches compute the deviation to classify data as normal or as faulty depending on the discrepancies from nominal behavior. Machine learning approaches use the most various techniques to build models that are able to distinguish between fault and nominal behaviors. Among these techniques, we



(a) Univariate time series.

(b) Multivariate time series

Figure 2: Univariate and multivariate timseries examples.
On the left we can see a univariate time series of the speed value for a single agent in the swarm. On the right we can see a visualization of what would be a multivariate time series composed of the values of speed, number of neighbors and positional entropy for a single agent in the swarm.

can find neural networks integrated with adaptive quasi-continuous second-order sliding mode controller [**?**] or deep learning minimally supervised methods [**?**]. All these techniques share the usage of data and the exploitation of features aimed at describing the task under execution. Machine learning techniques are divided into supervised and unsupervised; supervised approaches need all the samples to be labeled, unsupervised approaches do not require a labeled dataset and aim at performing sample distinction based on common patterns.
Further subdivision over data-driven methods is between univariate and multivariate time series. Blázquez-García et al. [**?**] propose a definition for univariate and multivariate time series:

Definition 0.2.1. *A univariate time series* $X = \{x_t\}_{x \in T}$ *is an ordered set of real-valued observations, where each observation is recorded at a specific time* $t \in T \subseteq \mathbb{Z}^+$.

Definition 0.2.2. *A multivariate time series* $X = \{\mathbf{x_t}\}_{t \in T}$ *is defined as an ordered set of* k*-dimensional vectors, each of which is recorded at a specific time* $t \in T \subseteq \mathbb{Z}^+$ *and consists of* k *real-valued observations,* $\mathbf{x_t} = (x_{1,t}, x_{2,t}, \ldots, x_{k,t})$.

It is assumed that each observation x_t is a realized value of a certain random variable X_t. Multivariate time series are preferred to univariate approaches thanks to their ability to capture the context in which the data is collected. An example of both can be seen in Figure 2.
In our work, we adopt a multivariate supervised data-driven approach based on fault injection during simulation execution. We used a gradient boosting model with balanced data retrieved from simulations.

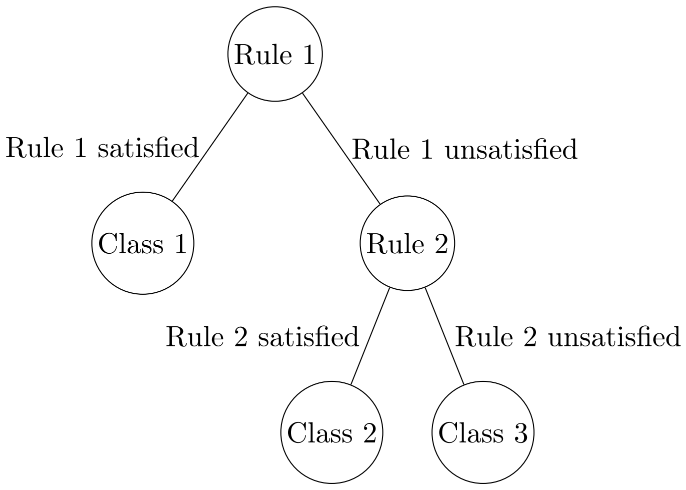


Figure 3: Decision tree example.

0.2.2. Decision Trees

Decision trees are a non-parametric supervised learning method used for classification and regression. Classification and Regression Trees (CART) [?] is considered to be the algorithm that regenerated interest in the subject of decision trees [?]. By using the conception of tree construction from CART and combining it with the Boosting technique we can achieve gradient boosting.
The decision trees method is based on the construction of data structures organized as undirected connected acyclic graphs with n nodes and $n-1$ edges. Each internal node states a condition that rules which branch must be taken by the data sample. Each leaf node contains a class or a probability distribution over possible classes. When a data sample reaches a leaf, it is classified or has a probability distribution over a set of possible classes. The goal of a decision tree is to partition the source set into subsets based on each internal node splitting rule. We can see a simplified version of a decision tree in Figure 3. Decision trees are built following decision tree inducers algorithms [?]. It has been shown that finding a minimal decision tree consistent with the training set [?] or finding the minimal equivalent decision tree for a given decision tree is NP-Hard [?]. Heuristics methods are required to bypass the challenges presented by decision trees. Heuristics methods can be divided into top-down and bottom-up, with the first being the most developed in the literature. Top-down algorithms are greedy by definition and construct the tree in a " divide and conquer " manner. The splitting rules are built based on univariate or multivariate splitting criteria. In our approach we use multivariate splitting criteria which are based on linear combinations of the input attributes. Multivariate splitting criteria can be found using greedy search, linear programming, and others techniques.

0.2.3. Ensemble Methods

Gradient boosting exploits the simplicity of multiple short decision trees to iteratively learn from its previous errors. This approach is part of the ensemble methods and represents the strongest characteristic of gradient boosting. Ensemble methods exploit several base estimators to increase generalizability and robustness with respect to a single estimator. Ensemble methods are composed of two families of techniques:

- Averaging methods: multiple estimators are trained independently to average their predictions. These techniques allow decreasing the variance of the prediction. Hashem and Schmeiser [?] propose an application of this technique with neural networks.
- Boosting methods: multiple simple estimators are trained sequentially on different sections of the dataset. These techniques aim at lowering the bias of the combined estimator and building a powerful ensemble of estimators [?].

Gradient boosting trivially belongs to the boosting techniques family. The training procedure of gradient boosting takes inspiration from the AdaBoost algorithm.

AdaBoost algorithm

For the following part we will assume to perform binary pattern recognition. The concept of pattern recognition will be taken from [?]: " automatic discovery of regularities in data through the use of computer algorithms and with the use of these regularities to take actions such as classifying the data into different categories ". The main idea behind AdaBoost is to have a " dream team " of classifiers taken from a large pool of classifiers [?]. For a given pattern p_i each expert classifier k_j can emit an opinion $k_j(p_i) \in \{-1, 1\}$ and the final decision of the committee K of experts is $sign(C(x_i))$, the sign of the weighted sum of experts opinions, where:

- $C(p_i) = \alpha_1 k_1(p_i) + \alpha_2 k_2(p_i) + \cdots + \alpha_n k_n(p_i)$.
- $k_1, k_2, \ldots, k_n$ denote the n experts selected from the pool of classifiers.
- $\alpha_1, \alpha_2, \ldots, \alpha_n$ are constant and represent the weights of each expert classifiers in the committee.

The classifiers choice is made by testing them in the pool of classifiers using a training set T of N multidimensional data points p_i. For each point p_i we have a label $y_i = 1$ or $y_i = -1$. Classifiers are ranked according to an exponential loss function cost:

- $e^{-\beta}$ for each success.
- e^{β} for each fail.

It is required to have $\beta > 0$ so that whenever a classifier fails it gets more penalized than a success.
The main idea of AdaBoost is to systematically proceed on extracting one classifier from the pool at each iteration, the goal is to draft the best pool of classifiers by the end of the procedure. In the beginning, all data points have the same weight. As the algorithm proceeds, the data points where the classifiers perform badly are assigned larger and larger weights. The drafting aims at selecting new classifiers that can help on the samples that are misclassified from the members of the committee.
To explain the ranking of classifiers we have to imagine a pool with already $m-1$ classifiers in it, at the m-th iteration we have to select the next classifier. The linear combination

of classifiers is currently:

$$C_{(m-1)}(p_i) = \alpha_1 k_1(p_i) + \alpha_2 k_2(p_i) + \cdots + \alpha_{m-1} k_{(m-1)}(p_i)$$

It needs to be extended to:

$$C_m(p_i) = C_{(m-1)}(p_i) + \alpha_m k_m(p_i).$$

At he beginning of the algorithm ($m = 1$), $C_{(m-1)}$ is the zero function. The total cost of the extended classifier as exponential loss is:

$$E = \sum_{i=1}^{N} e^{-y_i C_{(m-1)}(p_i) + \alpha_m k_m(p_i)} \tag{1}$$

where α_m and k_m are yet to be optimally determined.
Equation 1 can be rewritten as:

$$E = \sum_{i=1}^{N} w_i^{(m)} e^{-y_i \alpha_m k_m(p_i)} \tag{2}$$

where

$$w_i^{(m)} = e^{-y_i C_{(m-1)}(p_i)} \; \forall i \in \{1, \ldots, N\}$$

The algorithm begins with $w_i^{(m)} = 1 \; \forall i \in \{1, \ldots, N\}$.During the algorithm execution, the vector $w^{(m)}$ represents the weight assigned to each data point at iteration m. Equation 2 can be split into:

$$E = \sum_{y_i = k_m(p_i)} w_i^{(m)} e^{-\alpha_m} + \sum_{y_i \neq k_m(p_i)} w_i^{(m)} e^{\alpha_m} \tag{3}$$

meaning that the total cost is the weighted cost of all hits plus the weighted cost of all misses. Writing the first summands in Equation 3 as $W_c e^{-\alpha_m}$ and the second as $W_e e^{\alpha_m}$, it can be simplified as:

$$E = W_c e^{-\alpha_m} + W_e e^{\alpha_m} \tag{4}$$

the first term is weighted total cost of all hits and the second term is the weighted total cost of all misses. Equation 4 can be rewritten as:

$$e^{2\alpha_m} E = (W_c + W_e) + W_e(e^{2\alpha_m} - 1)$$

where $(W_c + W_e)$ is the total sum of weights W. We can observe that the right hand side is minimized when the classifier with the lowest cost W_e is chosen. Intuitively, the next draftee k_m is the one with the lowest penalty given the current set of weights. After picking the m-th member, the optimal weight α_m is:

$$\alpha_m = \frac{1}{2} \ln \left(\frac{1 - e_m}{e_m} \right)$$

where $e_m = \frac{W_e}{W}$.

0.2.4. Gradient Boosting

The gradient boosting model is rather old but it is founded on simple concepts and can achieve promising results thanks to its robustness to overfitting and its ability on handling unbalanced data.
Gradient boosting is based on the concept of decision trees and the AdaBoost algorithm with some modifications. AdaBoost works by iteratively drafting new classifiers and weighting their output based on their performance on misclassified data. Gradient boosting starts with a single trivial classifier and then iteratively builds classifiers able to predict the errors left from their predecessor. The classifiers built after the first one are weighted by the learning rate, this allows the model to slowly reach the lower values of the loss function. Algorithm 1 is the generic gradient tree-boosting algorithm as stated in [?].

Algorithm 1 Gradient Tree Boosting Algorithm.

```
1:  Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$
2:  for $m = 1$ to $M$ do
3:      for $i = 1$ to $N$ do
4:          Compute $r_{i,m} = -\left[\frac{\delta L(y_i, f(x_i))}{\delta f(x_i)}\right]_{f=f_{m-1}}$.
5:      end for
6:      Fit a regressor tree to the targets $r_{i,m}$ giving terminal regions $R_{j,m}$, $j = 1, 2, \dots, J_m$.
7:      for $j = 1, 2, \dots, J_m$ do
8:          Compute $\gamma_{j,m} = \arg\min_\gamma \sum_{x_i \in R_{j,m}} L(y_i, f_{m-1}(x_i) + \gamma)$.
9:      end for
10:     Update $f_m(x) = f_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{j,m} I(x \in R_{j,m})$.
11: end for
12: Output $\hat{f}(x) = f_M(x)$
```

The algorithms works as follows:

1. At the first step we select a function $f_0(x)$ able to classify x in order to minimize $\sum_{i=1}^{N} L(y_i, \gamma)$. Where:
    - x is a data sample.
    - N is the total number of samples.
    - y_i is the lable of x_i.
    - γ identifies the predicted value.
    - $L(y_i, \gamma)$ is a differentiable loss function on label y_i. For the classification it is convenient to use the negative log likelihood also known as cross-entropy error function:

$$L(y_i, \gamma) = -\sum_{i=1}^{N} y_i \ln(\gamma) + (1 - y_i) \ln(1 - \gamma)$$

considering γ as the predicted probability. To minimize the error function we compute the first derivative of $L(y_i, \gamma)$. It results that:

$$\frac{\delta}{\delta \ln \gamma} [y_i \ln(\gamma) + (1 - y_i) \ln(1 - \gamma)] = -y_i + \gamma$$

2. We then proceed to iterate over the total number M of estimators we want to build.
3. We iterate over all the samples to compute the pseudo-residuals with

$$r_{i,m} = - \left[\frac{\delta L(y_i, f(x_i))}{\delta f(x_i)} \right]_{f=f_{m-1}}$$

where:

- The indexes i and m stands for number of sample and number of classifier respectively.
- The right term identifies the derivative of the loss function with respect to the predictive value calculated with the previous classifier $(m - 1)$.

Computing the derivative of the loss function with respect to the predictive function we obtain that:

$$r_{i,m} = y_i - \gamma.$$

4. Given the pseudo-residuals computed at the previous step we proceed on fitting a new decision tree able to predict $r_{i,m}$ and returning terminal regions $R_{j,m}$ as leaf nodes.
5. We iterate over the terminal regions $R_{j,m}$ and compute a prediction for each of them. The prediction are computed according to

$$\gamma_{j,m} = \arg\min_{\gamma} \sum_{x_i \in R_{j,m}} L(y_i, f_{m-1}(x_i) + \gamma)$$

where:

- $\gamma_{j,m}$ is the new prediction.
- $R_{j,m}$ are the terminal regions.
- $f_{m-1}(x_i)$ is the prediction of the previous classifier.

This formula computes the sum of the function loss over all the terminal regions given the previous classifier prediction and the new prediction. In the end, it selects the prediction γ that minimizes the sum. Approximating the loss function with a second order Taylor polynomial we obtain that the new prediction γ is computed as:

$$\gamma_{j,m} = \frac{r_{i,m}}{\gamma(1 - \gamma)}$$

6. Then we proceed on updating the prediction for each sample according to

$$f_m(x) = f_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{j,m} I(x \in R_{j,m})$$

   where:

   - $f_m(x)$ is the new prediction for classifier m.
   - $f_{m-1}(x)$ is the previous classifier prediction.
   - ν is the learning rate.
   - $\gamma_{j,m}$ is the prediction of terminal region $R_{j,m}$.
   - $I(x \in R_{j,m})$ is the identity function: $\begin{cases} 1 & \text{if } x \in R_{j,m} \\ 0 & \text{otherwise} \end{cases}$.

7. In the end the algorithm returns $\hat{f}(x)$ which is the classifier function computed at the last step.

0.2.5. Feature Importance

Despite its simplicity, it is quite difficult to understand the real mechanism behind gradient boosting. To interpret the execution of the algorithms it is useful to know the choices made during the training phase. Since it would be bothersome to control each estimator, we use the permutation importance of features introduced in [?]. Breiman proposes to use internal out-of-bag estimates and rerun the algorithm using only selected variables. Consider a dataset with M input variables. At the end of the training phase, the values of the m-th feature in the left out data are randomly permuted and the trained model is evaluated on the data with shuffled values. This is repeated for each feature $m = 1, 2, \ldots, M$. At the end of the run, each model evaluates out-of-bag data $\mathbf{x}_n$. The plurality of votes from each classifier for $\mathbf{x}_n$ with the m-th value shuffled is compared with the true class label of $\mathbf{x}_n$ to give a misclassification rate. In the end, we obtain the percent increase in misclassification rate compared to the out-of-bag with intact variables. In our approach we used the `permutation_importance` implemented in the `scikit-learn` library [?]. From their documentation: " permutation feature importance is defined to be the decrease in a model score when a single feature value is randomly shuffled ". Permutation feature importance is able to show more realistic results since it does not show bias towards numerical features and it detaches from the training set.