POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Assessing Feature Importance for Centralized Fault Detection in Robot Swarms

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Luciano Franchin**

Student ID: 921093
Advisor: Prof. Francesco Amigoni
Co-advisors: ing. Davide Azzalini
Academic Year: 2021-22

Abstract

Here goes the Abstract in English of your thesis followed by a list of keywords. The Abstract is a concise summary of the content of the thesis (single page of text) and a guide to the most important contributions included in your thesis. The Abstract is the very last thing you write. It should be a self-contained text and should be clear to someone who hasn't (yet) read the whole manuscript. The Abstract should contain the answers to the main scientific questions that have been addressed in your thesis. It needs to summarize the adopted motivations and the adopted methodological approach as well as the findings of your work and their relevance and impact. The Abstract is the part appearing in the record of your thesis inside POLITesi, 0 the Digital Archive of PhD and Master Theses (Laurea Magistrale) of Politecnico di Milano. The Abstract will be followed by a list of four to six keywords. Keywords are a tool to help indexers and search engines to find relevant documents. To be relevant and effective, keywords must be chosen carefully. They should represent the content of your work and be specific to your field or sub-field. Keywords may be a single word or two to four words.

Keywords: here, the keywords, of your thesis

Abstract in lingua italiana

Qui va l'Abstract in lingua italiana della tesi seguito dalla lista di parole chiave.

Parole chiave: qui, vanno, le parole chiave, della tesi

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

2 State of the art **5**

2.1 Fault Detection and Diagnosis 5

2.1.1 Exogenous and Endogenous Fault Detection 6

2.1.2 Faults 6

2.1.3 Features 8

2.2 Machine Learning Techniques for Fault Detection 9

2.2.1 Data-Driven Fault Detection 9

2.2.2 Decision Trees 11

2.2.3 Ensemble Methods 11

2.2.4 Gradient Boosting 14

2.2.5 Feature Importance 17

3 Problem Formulation **19**

3.1 Theoretical Formalization 19

3.2 Problem Setting 19

4 Data collection **21**

4.1 Simulation Instruments 21

4.1.1 Simulators 21

4.1.2 Robot Description 21

4.1.3 Flocking 24

4.1.4 Foraging 26

4.1.5 Robot Mobile Fulfillment System 28
4.2 Fault Implementation 30
4.2.1 ARGoS Fault Injection 30

5 Fault Detection **33**
5.1 Data Retrieval 33
5.1.1 Running Examples 33
5.2 Simulation Data Translation 37
5.3 Feature Computation 39
5.3.1 Position, Direction, Speed, Cumulative Speed 39
5.3.2 Neighbors Number, Neighbors Average Distance 40
5.3.3 Position Entropy 41
5.3.4 Centroid Distance,Cumulative Centroid Distance 41
5.3.5 Area Coverage, Speed Area Coverage 42
5.3.6 Swarm Position, Swarm Speed, Swarm Area Coverage 42

6 Experiments **45**
6.1 Data Sets Analysis 45
6.1.1 Flocking Features Analysis 45
6.1.2 Foraging Features Analysis 53
6.1.3 RMFS Features Analysis 61
6.2 Metrics 67
6.2.1 Confusion Matrix 67
6.2.2 Precision Recall Curve 69
6.3 Gradient Boosting Performance Evaluation 69
6.3.1 Flocking Fault Detection 70
6.3.2 Foraging Fault Detection 72
6.3.3 RMFS Fault Detection 72

7 Conclusions and Future Works **75**
7.1 Results 75
7.2 Future Works 75

Bibliography **77**

List of Figures **83**

List of Tables **85**

Acknowledgements **87**

1 | Introduction

Robots are a concrete companion in our everyday life and constitute a fundamental gear in the nowadays production system. Since the introduction of the first industrial robots in the middle of the 20th century [38, pg.3-5], robots are a worldwide widespread production basic instrument. The rapid growth and diffusion of robot technologies have been driven by the need for automation and acceleration of human manufacturing capabilities. The propagation of robotic systems and the need of integrating multiple robot-operated machineries guided the birth of multirobot systems. This new approach leads to a new interpretation of the robot as an agent of a more complex system. With the development of the multirobot systems interaction capabilities, swarm robotics has become a concrete reality representing a fundamental component in nowadays process industries. It is possible to observe swarm technologies in industrial fields as managing warehouses [47], in environment preserving tasks like ocean-skimming and oil removal [36] or for farming activities [1]. Swarm robotics is defined as "the study of how a large number of relatively simple physically embodied agents can be designed such that a desired collective behavior emerges from the local interactions among the agents and between the agents and the environment" [43]. A swarm of robots can be considered a robust system thanks to its inherited properties: redundancy, naturally distributed system, and individual simplicity [52]. However, it has been demonstrated that, together with its advantages, swarm robotics exhibits high susceptibility to system failures due to individual failures [6]. To reach the goal of long-term autonomy [30], it has become a compelling need to build and deploy safe and reliable products. Modern days approaches have studied a collection of techniques to detect and diagnose faults to decrease downtimes and to be able to ensure safety and reliability for the entire system lifetime. Our approach focuses on a centralized machine learning approach for fault detection and, more precisely, we try to understand which are the features to analyze to perform fast and reliable fault detection.

Autonomous agents can be either software or physical agents, both representations can be analyzed as an entity that perceives and interacts with the environment. In the world of swarm robotics, the individuals that make up the swarm robotic system have to be autonomous with a physical embodiment in the world and they have to physically in-

teract with the environment [43]. The perception of the environment happens through the acquisition of data from several sensors which varies from different robots and different tasks, while the environment interaction happens through the activation of different actuators. Due to the physical limitations of sensors, the swarm agent is unable to perceive the entire environment and all of the other agents; the partial observability of the environment is one of the characteristics of swarm robotics that poses itself both as an advantage, to limit the size of analysis space in the single agent, and as a disadvantage, to limit the agent sensory capabilities.

The use of robots becomes very useful when it is necessary to execute a task that is unfeasible or too demanding to humans. These kinds of tasks are defined with the four D's: too Dangerous, too Dull, too Dirty, and too Difficult [27]. In production environments such as warehouses or industrial processing plans, robots reliability has become a key aspect to ensure security and decrease losses in terms of time and resources. Even if the deployed robots guarantee a declared level of reliability, it has become compulsory to perform fault detection and diagnosis to identify failures as quickly as possible. Being able to promptly detect malfunctions enables faster recoveries and lower downtimes. Fault detection and diagnosis in swarm robotics is already a deeply investigated subject with plenty of research possibilities for new findings and improvements. The main challenges we face are posed by the swarm paradigm, this approach offers a lot of analysis surface but presents technical complications like the size of the analysis space and the computational time required to identify an anomaly.

Fault detection is a field of study that resembles the field of anomaly detection, they both share common technicalities and approaches. It is possible to find studies that refer to model-based fault detection that dates back to 1997 [24] and more recent data-driven approaches [29]. Modern days approaches focus on machine learning techniques [21] and aim to perform fault detection with unsupervised methods exploiting deep learning [3]. All the machine learning and data-driven approaches offer a vast field of techniques to test, however, in front of all these possibilities, it comes in handy to analyze which kind of data and which features are useful to analyze to obtain significant performances.

Our project is based on the analysis of task executions from two different simulators, namely ARGoS and RAWSim-O, and the prediction performance of a Gradient Boosting model. The ARGoS simulator is used to simulate the tasks of flocking and foraging while RAWSim-O simulates the execution of a warehouse management situation. We injected the faults in the simulator's core code and then we have collected different task executions. The centralized approach of this work refers to the point of view from which we observe the swarm behaviors. In our work we assume there is an external observer which has to control each agent in the swarm. The activity of the external controller consists in iden-

tifying each agent and computing its position with respect to a local positioning system. This assumption allows us to detach from the concept of retrieving data directly from the agent itself, allowing us to detect any kind of fault. We use the data collected from the simulations to compute complex features and build a multivariate time series. The model in consideration is trained on datasets composed of single timesteps from chained multivariate time series with an equal number of nominal and fault samples. The goal of our research is to analyze the informativeness of raw data like position, speed, orientation, and complex data like position entropy, number of neighbors, and area coverage to name a few. Our experiments show that... [TO BE CONTINUED AFTER CONCLUSIONS]

The thesis is organized as follows.
In Chapter 2, we describe the current state of the art in the field of fault detection for swarm robotics and present the theoretical background of the instruments we have used to develop our work. Chapter 3 presents the theoretical formalization of our approach and describes the overall environment in which this thesis is constructed. Chapter 4 introduces the technical instruments we have used to simulate a swarm robotic environment, it proceeds on presenting a general overview on how the swarm robotic tasks work, and, in the end, it explains the modification we have implemented to collect the data from the simulators. In Chapter 5 we present the precise settings we have used to execute the swarm simulations and we explain in detail the procedure to compute the features used to perform the fault detection. In Chapter 6 we show the graphs of the the elaborated data and comment on how they can influence the model performances. We proceed then on describing the results obtained from the model and how they can be interpreted. Chapter 7 concludes the thesis by showing the results obtained from our experiments and what are the final consideration we can derive from them. We finish the chapter by analyzing future development opportunities for our work, listing some aspects that can be further explored, and proposing more advanced technologies that can improve the performance and the results of this work.

2 | State of the art

2.1. 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 systems, 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 [44]. 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 point 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 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 [26]. 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 [4]. 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.

2.1.1. 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 [35] or local neural networks [45]. 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 [20].
Exogenous fault detection refers to the ability of robots to detect faults in one another. This approach has been proposed by Christensen et al. in [12] 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.

2.1.2. Faults

In a generic operational environment, we can find two kinds of errors: random errors and systematic (design) errors [52]. 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 [52] list a series of possible internal hazards that can appear in individual robots. Table 2.1 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

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 2.1: Internal hazards for a single bot.

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 4.1.3, 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 4.1.4, 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.
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 [12] 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 in Figure 2.1 taken from[37]) 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

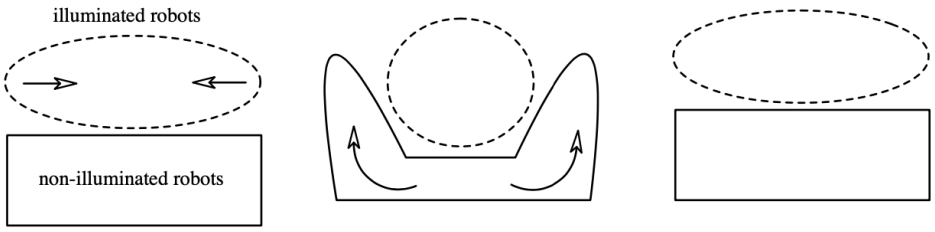


Figure 2.1: Emergent taxis behavior illustration.
The figure on the left depicts a set of robots with the role of beacon taxis, namely illuminated robots. In the center, we can see the rest of the robots, namely non-illuminated robots, that moves towards the light emitted from the beacon of the robots in front of them. The continuous combination of these two movements constitutes the emergent taxis behavior.

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 behavior.
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 4.1.2.

2.1.3. Features

In this work, by feature, we intend the measurable properties or characteristics 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 5.2. From [31], 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. [29] 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 [46] for a decentralized fault detection system.
Wei et al. [50] introduce the metrics for flocking behavior. Among them, we can find:

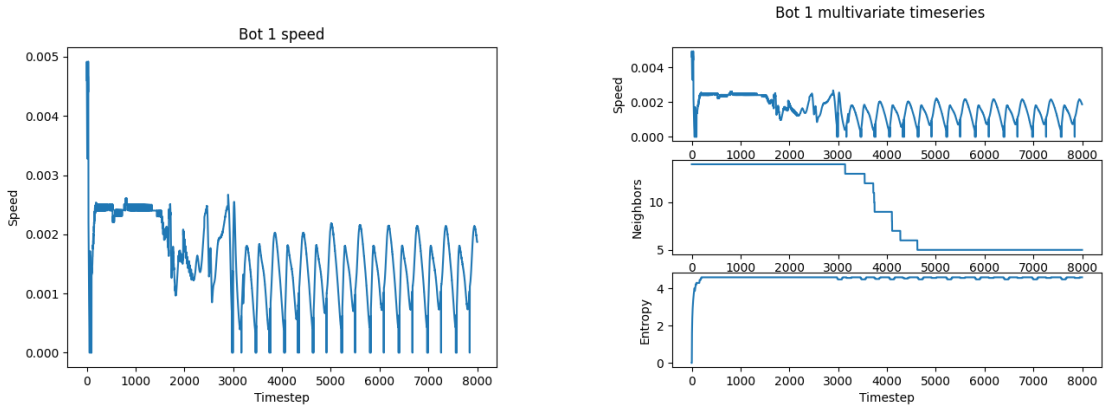
- " The agents of the swarm should always face approximately the same direction ", that suggested we analyze the direction of each agent.
- " The agent should remain close to each other ", meaning the average distance from all the other agents can enclose insightful information.

Khalastchi et al. [28] 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.

2.2. Machine Learning Techniques for Fault Detection

2.2.1. Data-Driven Fault Detection

Data-driven approaches can be divided into statistical approaches and machine learning approaches [26]. Kalman filters [2] or particle filters [49] 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 can find neural networks integrated with adaptive quasi-continuous second-order sliding mode controller [48] or deep learning minimally supervised methods [3]. All these techniques share



(a) Univariate time series.

(b) Multivariate time series

Figure 2.2: 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.

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. [7] propose a definition for univariate and multivariate time series:

Definition 2.2.1. *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 2.2.2. *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 2.2.

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. In the next sections, we will explain the main concepts

behind the techniques used in our work.

2.2.2. Decision Trees

Decision trees are a non-parametric supervised learning method used for classification and regression. Classification and Regression Trees (CART) [10] is considered to be the algorithm that regenerated interest in the subject of decision trees [32]. 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 2.3. Decision trees are built following decision tree inducers algorithms [42]. It has been shown that finding a minimal decision tree consistent with the training set [19] or finding the minimal equivalent decision tree for a given decision tree is NP-Hard [53]. 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.

2.2.3. 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 concerning 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 [22] propose an application of this technique with neural networks.

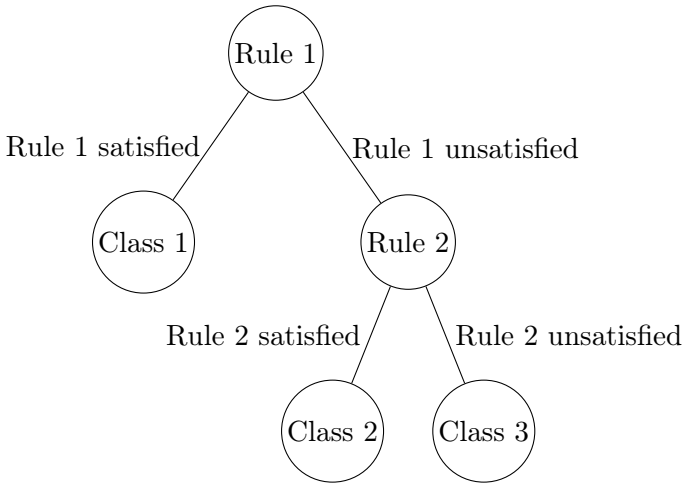


Figure 2.3: Decision tree example.

- 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 [17].

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 [5]: " 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 [41]. 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:

$$E = \sum_{i=1}^{N} e^{-y_i C_{(m-1)}(p_i) + \alpha_m k_m(p_i)} \tag{2.1}$$

where α_m and k_m are yet to be optimally determined.
Equation 2.1 can be rewritten as:

$$E = \sum_{i=1}^{N} w_i^{(m)} e^{-y_i \alpha_m k_m(p_i)} \tag{2.2}$$

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

2.2 can be split into:

$$E = \sum_{y_i = k_m(p_i)} w_i^{(m)} e^{-\alpha_m} + \sum_{y_i \neq k_m(p_i)} w_i^{(m)} e^{\alpha_m} \tag{2.3}$$

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 2.3 as $W_c e^{-\alpha_m}$ and the second as $W_e e^{\alpha_m}$, it can be simplified as:

$$E = W_c e^{-\alpha_m} + W_e e^{\alpha_m} \tag{2.4}$$

the first term is weighted total cost of all hits and the second term is the weighted total cost of all misses. Equation 2.4 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 $e_m = \frac{W_e}{W}$.

2.2.4. 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 [23].

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 $R_{j,m}$, $j = 1, 2, \ldots, J_m$.
7: **for** $j = 1, 2, \ldots, J_m$ **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:
   $$\frac{\delta}{\delta \ln \gamma} \left[y_i \ln(\gamma) + (1 - y_i) \ln(1 - \gamma) \right] = -y_i + \gamma$$
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.

2.2.5. Feature Importance

Despite its simplicity, it is quite difficult to interpret the real decision-making 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 [9]. 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 [11]. 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 can show more realistic results since it does not show bias towards numerical features and it is independent from the training set.

3 | Problem Formulation

3.1. Theoretical Formalization

Given a set of agents $N = \{0, 1, \ldots, n\}$, a fixed set of features $D = \{f_1, f_2, \ldots, f_d\}$ and a number of timesteps T with $T \in \mathbb{N}^+$, our problem consists in predicting a binary label $y_t \in \{0, 1\} : t \in [0, T]$ given a data sample

$$\mathbf{x}_t = (x_0, x_1, \ldots, x_d)$$

with d being the number of features and $x_i \in \mathbb{R} : i \in [0, d]$.
To analyze the prediction from a broader view we can say that our model predicts a set of labels $\mathbf{y}_i = (y_1, y_2, \ldots, y_T)$, with T as the total number of timesteps in the time series, given a set of data samples

$$\mathbf{X}_i = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$$

of samples $\mathbf{x}_t : t \in [0, T]$ with d-dimensions.
In the end our data is collected in the form

$$\mathcal{X} = (\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_n)$$

where $\mathbf{X}_i \in \mathbb{R}^{T \times d}$ is a time series of size $T \times d$ with T as the number of timesteps and d the number of features.
The goal of our work is to find which subset $D' \subseteq D$ gives more insight for the task of predicting y_t given $\mathbf{x}_t$ with $t \in [0, T]$.

3.2. Problem Setting

The problem of FDD in swarms robotics can be analyzed in different scenarios with different approaches. As we have seen in 2.1.1, it is possible to observe the task execution from a different point of view. Based on which interpretation we choose, we have to analyze some aspects instead of others. As explained in 4.1.2, the agents we use offer

several kinds of sensors from which to retrieve different forms of attributes or features. In our approach, we choose to analyze the problem from an external point of view with a centralized controller completely detached from the agents. This approach is coherent with the swarm concept since it does not require continuous communication with a central controller, the swarm paradigm allows only broadcast messages directed to other agents in the swarm. Considering this situation, we have to discard every kind of information that directly comes from the robot sensors, assuming they will not even be reachable, and we have to consider only information that can be gathered from an external observer.
The FDD procedure we propose starts from the injection of faults in simulators, proceeds on extracting the data of the simulation execution from the point of view of an external observer, and then elaborates the data to build the multivariate time series. In real-world scenarios of fault detection, or more generally anomaly detection, it is quite rare to have the chance of simulating the environment under analysis, nonetheless, simulations might be costly in order of time or money for equipment. Being able to simulate can be considered an advantage but has to be used with caution. Even if we can simulate an arbitrarily large number of agents and repeat the simulations several times, we have to keep in mind that the simulations must reflect real scenarios and not detach from concrete possible situations.
In this project, we have not analyzed all the possible faults with all the possible combinations of parameters, instead, we have focused on a subset of mostly real scenarios easy to implement. Our ultimate goal is to build a dataset that can encapsulate some situations that reflects real-world situation while being easy to simulate. The choice of faults types and length of simulations is also a consequence of restricting the size of the problem, as usual for problems with several variable settings, they can quickly explode and become unfeasible to analyze.

4 | Data collection

4.1. Simulation Instruments

In this section, we will present the instruments, the techniques, and the tasks that we have used to collect the simulation data for our tests.

4.1.1. Simulators

This project has been developed with data retrieved from two different simulators. The first simulator which has been used is ARGoS (Autonomous Robots Go Swarming), developed by Carlo Pinciroli [40], is an open-source multi-robot simulator with the focus on real-time simulation of large heterogeneous swarm of robots. Thanks to its modular and open-source approach we have been able to easily modify the source code in order to implement the necessary simulations. We can see a snapshot of the graphical user interface in Figure 4.1. In our simulations we used foot-bot agents which will be introduced in Section 4.1.2.

The second simulator which has been used is RAWSim-O (Robotic Automatic Warehouse Simulator (for) Optimization), developed by Marius Merschformann [34], that has been created to simulate a Robotic Mobile Fulfillment System (RMFS) and to study the impact on the performances of different decision strategies. We can see a snapshot of the graphical user interface in Figure 4.2. Merschformann does not specify a unique physical robotic agent for its environment but uses a conceptual implementation of the Kiva systems shown in Figure 4.4.

4.1.2. Robot Description

The foot-bot is shown in Figure 4.3, it can be referred to also as MarxBot [8]. They were developed within the EU-funded project Swarmanoid [13]. The foot-bot is a miniature mobile robot that addresses the needs of having a large battery life, being able to perceive its peers, and being able to interact with them. The robot measures 17 cm in diameter and 29 cm in height with a weight of 1.8 kg. The base module provides rough-terrain mo-

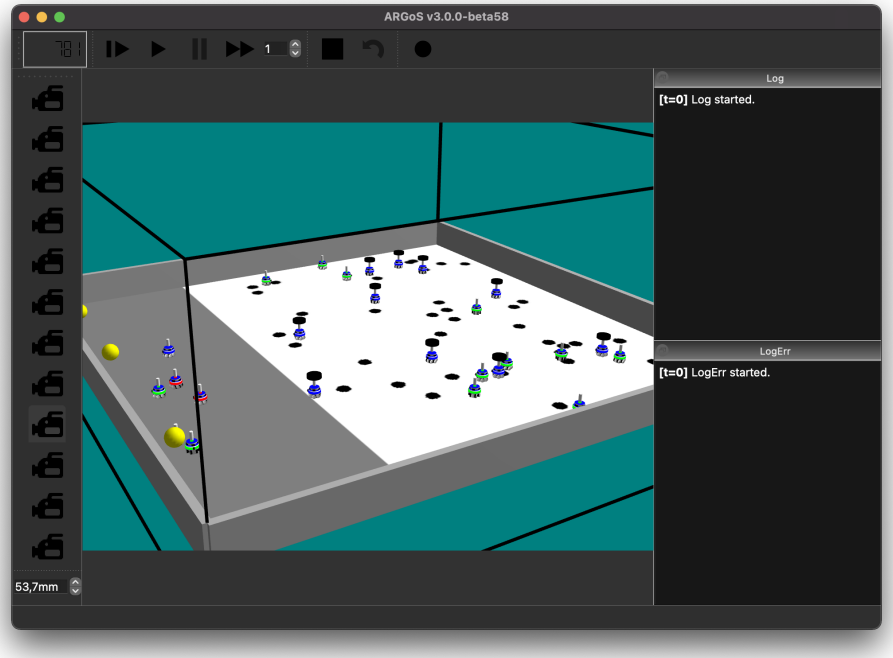


Figure 4.1: Snapshot of ARGoS graphical user interface.

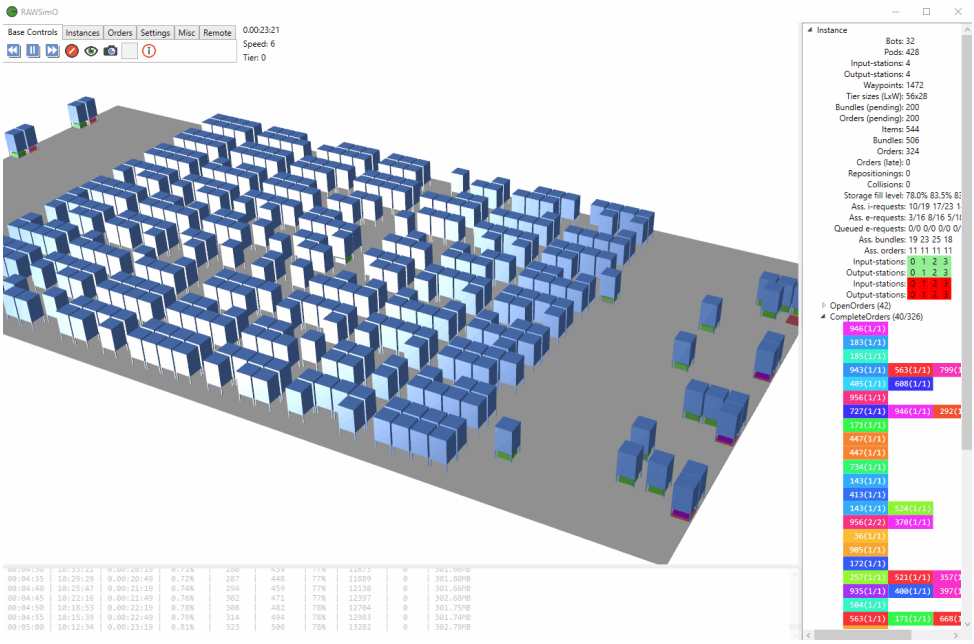


Figure 4.2: Snapshot of RAWSim-O graphical user interface.

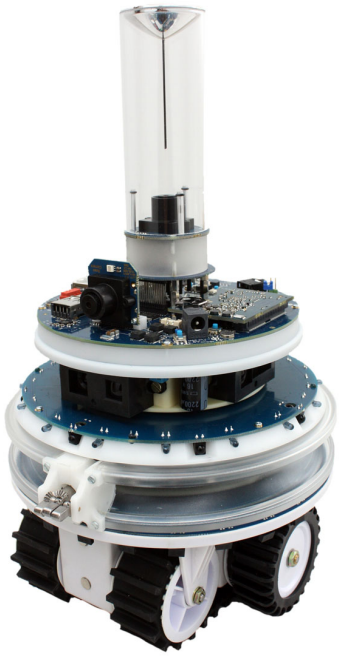
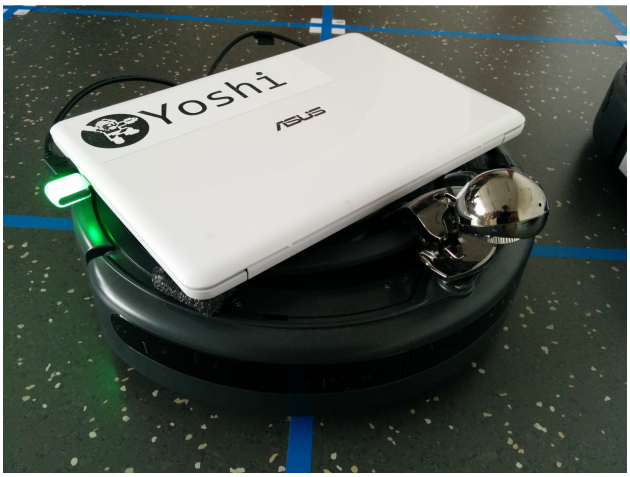
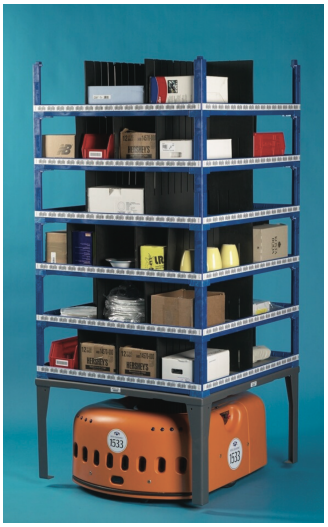


Figure 4.3: Photo of Foot-bot.

bility thanks to a combination of tracks and wheels. This module contains also proximity sensors, a 3-axis gyroscope, and a 3D accelerometer. A range and bearing module allows the robot to compute a rough estimate of the direction and the distance of the neighboring robots. A distance scanner allows the robot to build a 2D map of its environment. The on-board computer drives two cameras, one looking front and one oriented towards an omnidirectional hyperbolic mirror. The RAWSim-O simulator requires a robot that is able to carry heavy inventory pods and the Foot-bot would not be adequate for this task. The RMFS simulated by RAWSim-O is similar to the management system of the Kiva systems [14], known also as Amazon Robotics. An early version of Kiva's robots (Figure 4.4b)is composed of 5 main parts: navigation system, lifting mechanism, power system, collision detection system, and driving system. [18]. The navigation system is composed of a camera facing upwards, in charge of reading bar codes under inventory racks for identification, and a camera facing down reads barcodes on the floor for navigation. The lifting mechanism consists of a large screw that lifts inventory racks 5 centimeters from the ground. The power system is composed of four batteries that are charged at the charging station. The collision detection system is composed of infrared sensors and touch-sensitive bumpers. The driving system is made of two brushless dc motors that control independent neoprene rubber wheels making the robots move at 1.3 $\frac{m}{s}$. Physical experimental simulations of small warehouse environments have been deployed using a vacuuming robot [33]. The robots are equipped with ASUS Eee PCs, webcams for line-following, a blinking



(a) RAWSim-O demonstrator bot.



(b) Kiva robot carrying inventory pod.

Figure 4.4: RAWSim-O bots.

light for visual feedback, and RFID tag readers for waypoint recognition. The maximum velocity of the vacuuming robot is 0.21 $\frac{m}{s}$, while a complete turn takes 5.5 s. Maximum acceleration and deceleration are 0.5 $\frac{m}{s^2}$ and -0.5 $\frac{m}{s^2}$. A photo of the Kiva can be seen in Figure 4.4b. A photo of the demonstrator setup can be seen in Figure 4.4a.

In the following sections, we will explain the mechanism behind each task behavior and show the principal characteristics of the task.

4.1.3. Flocking

The flocking task we refer to is the one introduced by Ferrante et al. in [16]. By its definition, flocking is the " cohesive and ordered motion of a group of individuals in a common direction ". Ferrante et al. proposes a magnitude dependent motion control (MDMC) to " achieve flocking with minimal components ". A snapshot of flocking execution can be seen in Figure 4.5.

The method used to achieve flocking is composed of different components: proximal control, alignment control and motion control. The main concept behind this method is that each robot computes a flocking vector

$$\mathbf{f} = \mathbf{p} + \mathbf{a} + \mathbf{g}$$

where:

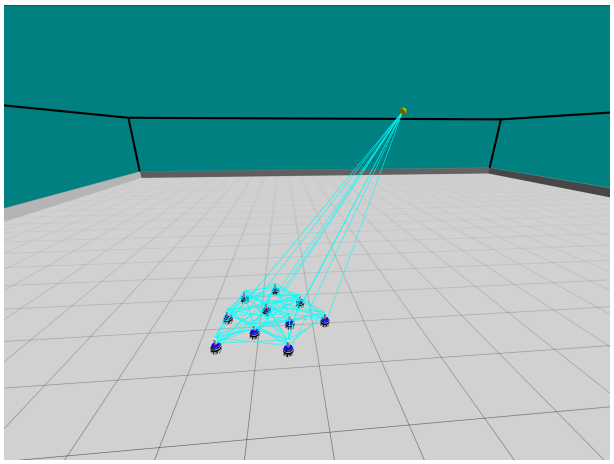


Figure 4.5: Snapshot of flocking task execution.

- **p** is the proximal control vector in charge of the attraction and repulsion forces.
- **a** is the alignment control vector in charge of the alignment rule.
- **g** is the goal direction vector in in charge of the goal following behavior.

Proximal Control

Proximal control computes all the forces needed to keep each robot at a certain distance from its neighbors. The proximal control rule encodes the attraction rules, to shorten the distance between neighbors, and the repulsion rules, to move further away robot that are too close. Proximal vector is computed as:

$$\mathbf{p} = \sum_{i=1}^{m_p} p_i(d_i)e^{j\phi_i}.$$

Where:

- $p_i(d_i)e^{j\phi_j}$ is a vector in the complex plane, its formula is defined in [16].
- m_p is the number of neighboring robots perceived in the range D_p.
- d_i is the relative range computed from the body-fixed reference frame.
- ϕ_i is the bearing of the i-th neighboring robot.

Alignment Control

Alignement control computes the average orientation of its neighbors and adjusts its own orientation accordingly. The alignment control method uses the following formula:

$$\mathbf{a} = \frac{\sum_{i=0}^{m_a} e^{j\theta_i}}{\left|\left|\sum_{i=0}^{m_a} e^{j\theta_i}\right|\right|}.$$

Where:

- θ_0 is the orientation of the focal robot, namely the current robot.
- m_a is the number of neighbors within a range D_a of the focal robot.
- $\theta_i : i \in \{1, \ldots, m_a\}$ are the orientations of the neighbors.
- The operator $||\cdot||$ denotes the length of a vector.

Motion Control

Motion control has to convert the flocking vector into forwarding speed u and angular speed ω. The model proposed by Ferrante et al. identifies as $\mathbf{f}_x$ and $\mathbf{f}_y$ the projections of the vector $\mathbf{f}$ on the x-axis and the y-axis respectively. $\mathbf{f}_x$ determines the forward speed u, while $\mathbf{f}_y$ determines the angular speed ω. The model equations are:

$$u = K_1\mathbf{f}_x + U$$
$$\omega = K_2\mathbf{f}_y.$$

Where:

- U is a forward biasing speed.
- K_1 and K_2 are linear and angular gains respectively.

4.1.4. Foraging

To describe the foraging we refer to [51] and the code used in the ARGoS simulator. The foraging task is broadly defined as searching for, collecting, and returning resource items to a point of delivery or nest area. This behavior can be seen as a heterogeneous task composed of several actions such as exploration, harvesting, resource transportation, and homing. Foraging robots are mobile robots capable of searching and transporting objects to a collection point. This task can be executed by single robots or multiple

robots operating collectively. A snapshot of the foraging task can be seen in Figure 4.1. The finite state machine represented in Figure 4.6 summarizes the control algorithm used in [39]. The robot is always in one of four states: resting (Rest), exploring (Expl), homing (Home), searching for space in nest (Nest). In our experiments, there are multiple target objects, namely resources to collect, which continuously regenerate, and a unique nest area where collected objects are deposited. Each robot has two internal variables: `RestToExploreProb` defines the probability of transitioning from resting state to exploring state; `ExploreToRestProb` defines the probability of transitioning from exploring state to resting state.

Resting

This is the initial state where all the robots start from the nest area. It is a state where the robots are in a sleepy state, they can only perceive the communications received from their neighbors and update the internal variables. After a robot has spent an amount of time greater than `MinimumRestingTime`, if the random extraction from a uniform probability function is greater than `RestToExploreProb`, then the robot changes state to exploring, otherwise it stays in resting state.

Exploring

In this state, the robot is performing a random walk while performing obstacle avoidance. During this behavior the robots can not sense the resources sparse in the arena, they can not communicate with their neighbors where other resources are, and it has not any specific place to reach. The random walk behavior is self-explanatory, the robot does not follow a predefined direction but randomly changes where it is headed. If the robot passes over some resource then it grabs it and transitions to homing state. After a robot has spent an amount of time greater than `MinimumUnsuccessfulExploreTime`, if the random extraction from a uniform probability function is greater than `ExploreToRestProb`, then the robot changes state to homing, otherwise, it stays in exploring state.

Homing

In this state, the robot follows the signal emitted by the beacon that indicates the position of the nest area. The nest area is reached whenever the two ground sensor on the back perceives the nest area color on the floor. Whenever the robot reaches the nest area it drops the collected resource and transitions to searching for place in nest state.

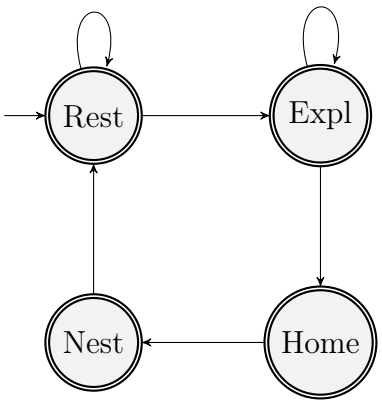


Figure 4.6: Foraging task finite state machine

Searching for Place in Nest

To avoid the positioning of all the robots on the border between the nest area and the area where resources are scattered, the algorithm forces the robots to spend some time searching for place in the nest. When a robot has spent `MinimumSearchForPlaceInNestTime` searching for a place in the nest, it communicates the result of its last exploration attempt to its neighbors and it changes its state to resting.

4.1.5. Robot Mobile Fulfillment System

The main innovation that RMFSs have brought to the field of warehouse management is to bring the shelves to the operators instead of the operators going to the shelves. RMFS can be summarized in a few simple components:

- Movable shelves, called *pods*, where inventory is stored.
- Storage locations where it is possible to store pods.
- Workstations where the pick order items are retrieved from the pods (*pick stations*) or replenishment order items are stored in the pods (*replenishment stations*).
- Mobile *robots* which can move underneath pods and carry them to workstations.

The storage and retrieval process can be seen in Figure 4.7. When a robot gets a replenishment, order it carries a selected pod to a replenishment station to fill. When a robot gets a pick order, it carries a selected pod to a pick station to deliver the requested items. The administration of an RMFS requires the solution of a set of optimization and allocation problems. In the approach proposed by Merschformann M. et al the solution of the decision problems is decomposed in a series of subproblems:

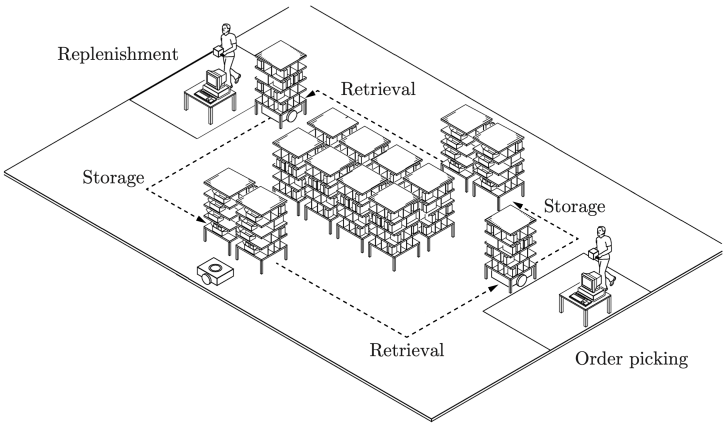


Figure 4.7: The central process of a RMFS.

- **Order assignment:** divided into replenishment order assignment (ROA) and pick order assignment (POA).
- **Task creation:** divided into pod selection, which in turn is divided into replenishment pod selection (RPS) and pick pod selection (PPS), and pod storage assignment (PSA).
- **Task allocation:** assignment of tasks to robots.
- **Path Planning:** planning of the paths.

For more detailed information about all the subproblems and their description, the reader can refer to [34]. A graphical representation of the simulation process can be seen in Figure 4.8. The simulation has to manage the update of the agents, the outsourcing of decisions to controllers, and the delivery of information to a visualizer. The framework of the simulation is composed of three layers of configuration: layout specification, scenario specification, and controller configuration. The *layout specification* specifies all the positions and individual characteristics of all stations, the waypoint system for robot direction, and the robot themselves. The *scenario specification* specifies the information about the scenario to simulate, like the simulation duration and settings about the inventory and order emulation. The *controller configuration* specifies which setting to use for each decision problem and their parameters.
The robot movements are simulated with linear acceleration and deceleration for straight movements and they're constrained to predefined paths, nevertheless, they can not perform curved movements, each robot must rotate on itself in order to change direction. We can see a picture that depicts a RMFS simulation made with RAWSim-O in Figure 4.2.

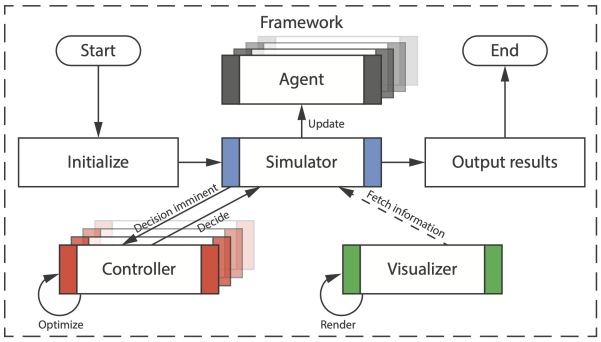


Figure 4.8: Overview of the simulation process in a RMFS.

4.2. Fault Implementation

Different faults have been implemented in the two simulators: in ARGoS we have implemented a rotating fault in which the bots rotate around a fixed point in the arena due to one wheel going faster than the other one; in RAWSim-O we have implemented a slowing fault where bots were constrained on traveling at decreased speeds.

4.2.1. ARGoS Fault Injection

Since the ARGoS simulator is built with a physical simulation purpose, it is easy to inject different faults in the core code of the controllers or define new unexpected behavior. The basic functioning of ARGoS is based on the information provided by three modules: an XML configuration file (`.argos`) which specifies the configuration of the experiment and the robot controllers; controller files (`.cpp`) compiled in one or more libraries that define the functioning of the robot and their components; loop functions files (`.cpp`) compiled in one library that defines external functions (i.e. the resource management in the foraging task). Our faults are implemented at the controller level, when a robot computes its flocking vector and translates it into the wheels' speed, the wheel speeds are hardcoded to fixed speeds. In our experiments, we forced the robots' wheels to stay at 2.0 $\frac{m}{s}$ for the left wheel and at 0.5 $\frac{m}{s}$ for the right wheel.

RAWSim-O Fault Injection

RAWSim-O is not oriented to simulate the physical interactions with the environment as much as ARGoS is. The basic functioning of RAWSim-O is based on three configuration files and the source code of the simulator. The configuration files are: a layout configuration file (`.xlayo`) that specifies settings like the number of robots and the aisles arrangement; a scenario configuration file (`.xsett`) that specifies settings about the sim-

ulation like its duration or how the orders are generated; and a controller configuration file (`.xconf`) that specifies the settings about the mechanisms to use to solve decision problems and their parameters. Our fault is implemented in the core code of the simulator, when a robot instance is constructed in the simulator, its physical engine gets initiated with the default speed reduced to a fraction of it. This implies that the robot is malfunctioning from the beginning of the simulation. Even though this fault seems easy to detect it is not as trivial as a complete stop, nevertheless, a more complex fault could be even more likely to be detected. For example, from the physical point of view, one more complex fault we could implement is the bot going back and forth from two adjacent positions. This fault, without considering the difficulty of implementing it, can be considered as trivial as a complete stop since the robot will stop visiting new areas and the area coverage feature would be discriminant between nominal and faulty agents. On the other hand, some more complex fault, like spinning in circles, would be detected by the catastrophic consequences they would have. Considering the fragile environment in which the agents are operating, if any of them would start wandering around not respecting the predefined trajectories, it could deeply damage the warehouse environment and thus force the system to a complete stop.

5 | Fault Detection

In this chapter, we will explain the construction process behind the datasets that have been used to train the model and observe the feature's performance. The process is composed of three main steps: running simulations, translating simulation log files into time series, computing robot features time series. The simulations are executed through the simulators introduced in Section 4.1.1 and will be explained in more detail in Section 5.1.1. The translation of log files into time series data objects is explained in Section 5.2. Finally, the computation of the features' time series is described in Section 5.3.

5.1. Data Retrieval

Simulations data is collected through comma separated value files (`.csv`) that are directly written by the simulators. These files are then processed with `python` and all the robots' positions are translated into time-series objects.

5.1.1. Running Examples

In our experiments, we run several simulations with different settings. In this section, we will list the settings used for the simulations of each task and show an example of task execution.

ARGoS Flocking

For the flocking task, we have used a simulation environment of a 30×30 m^2 arena surrounded by walls and with 15 bots. Even though the walls have never worked as a constraint measure, they are necessary to put physical bounds to the arena. The coordinate system of the simulation is considered to have the $(0, 0, 0)$ position in the middle of the arena at ground height. The simulations have a duration of 800 seconds, this value has been chosen arbitrarily after some tests to allow the swarm to complete the task. The goal of the task is to reach a light positioned in the center of the arena at a height of 4 m from the ground, the light coordinates are $(0, 0, 4)$ and it is oriented toward $(0, 0, 0)$. The

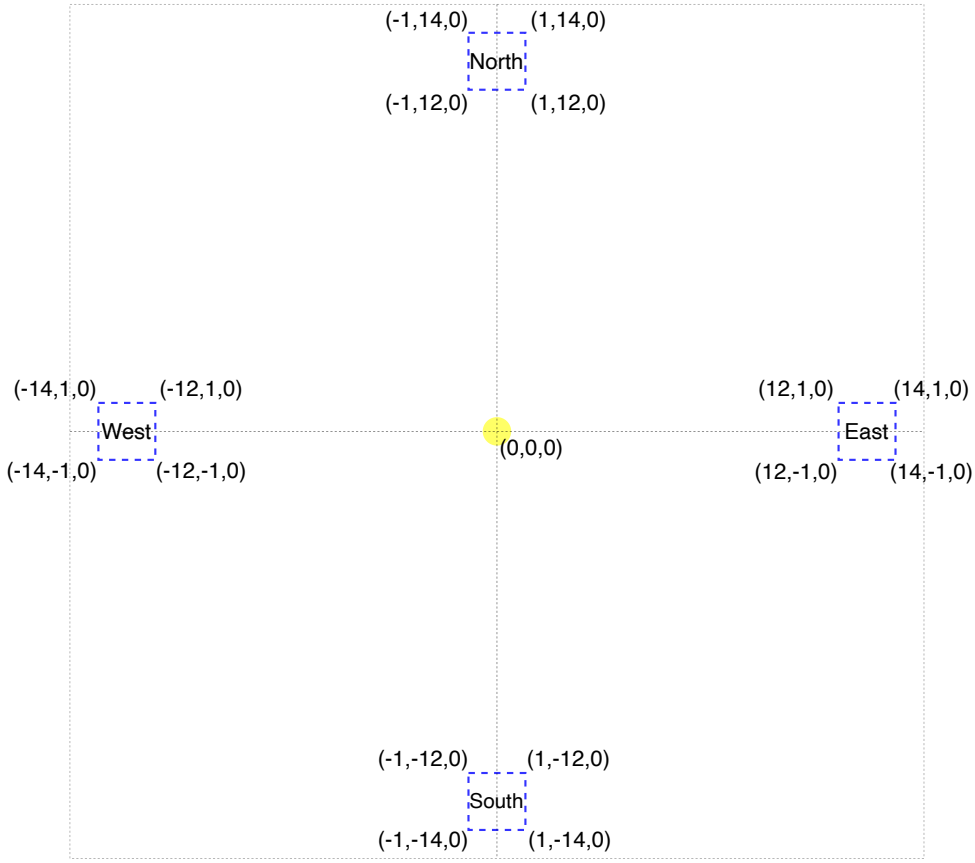
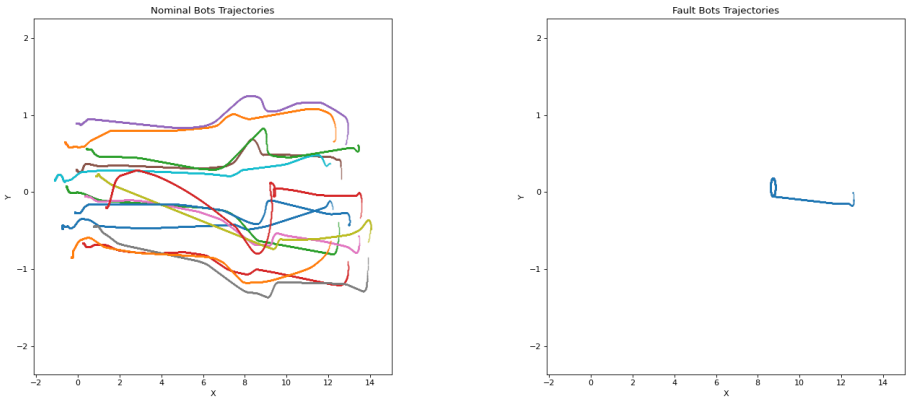


Figure 5.1: Flocking task robots starting positions.

flocking task presumes that the robots manage to reach the light while keeping a formation, more precisely they have to keep a distance of 75 *cm* from all their neighbors. In our `.argos` settings we specified a `gain` parameter of 1000 for the flocking controller. The robots begin each flocking simulation from one of the 4 different starting areas, namely North, South, East, and West. We can see the starting positions depicted in Figure 5.1 with their corresponding names and the light in the center of the arena.
Faults have been injected in the 10%, 20%, or 33% of the swarm population and at 0 s, 50 s, 150 s, or 400 s from the beginning of the simulation. Completely nominal simulations are run together with faulty simulations in order to have a complete dataset of a real situation. An example of the robot trajectories can be seen in Figure 5.2 where nominal and faulty agents are separated in two different graphs.

ARGoS Foraging

For the foraging task, we have used incremental settings that adapts with the increase of the arena size. The arena size starts from 5×5 m^2 and ends in 8×8 m^2 going trough each possible combination with sides of size $\{5, 6, 7, 8\}$ m thus using 16 different arena



(a) ARGoS flocking nominal bots trajectories.

(b) ARGoS flocking faulty bots trajectories.

Figure 5.2: ARGoS flocking plotted trajectories.

sizes. In this task, the arena is surrounded by walls and they're fundamental to keep the bot inside the arena while they're performing the random walk. The goal of the task is to retrieve the items in the resource area and bring them to the nest area. There are 4 lights to signal the position of the nest on the nest side of the arena, they're positioned at a height of 1 m from the ground. The lights' coordinates change with the size of the arena. At each simulation, they are positioned at the 10%, 30%, 70%, and 90% of the wall length in order to be adequately distanced. The robot number starts from 20 agents and scales in a directly proportional way with the arena size. The simulations last 800 seconds, this value has been chosen arbitrarily but the task execution does not depend on it as it is for flocking. The foraging task requires some parameters for the agents' foraging behavior, these values are scaled with the arena size in order to have coherent simulations among different arenas. The foraging task parameters are:

- `minimum_resting_time` parameter: initialized at 5 s.
- `minimum_unsuccessful_explore_time` parameter: initialized at 60 s.
- `minimum_search_for_place_in_nest_time` parameter: is initialized at 15 s.

Each foraging simulation requires a number of items to be scattered around in the resources area, this value is initialized at 15 items and scales with the arena size. An example of the foraging arena can be seen in Figure 5.3.
For each arena size, a number of nominal simulations are executed together with a number of simulations with injected faults. For the faulty simulations, we have considered 10%, 20%, or 33% of the swarm population to be infected at 0, 100, or 400 s from the task

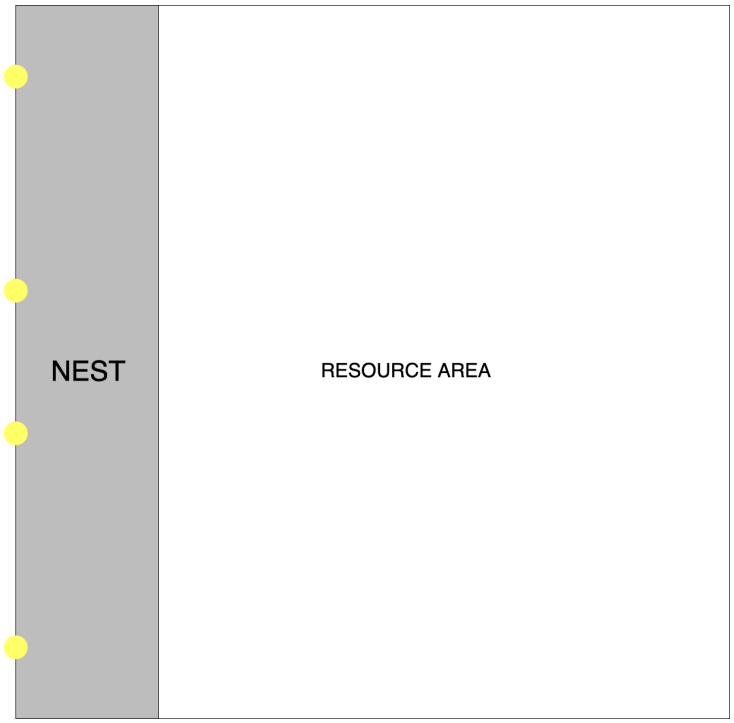


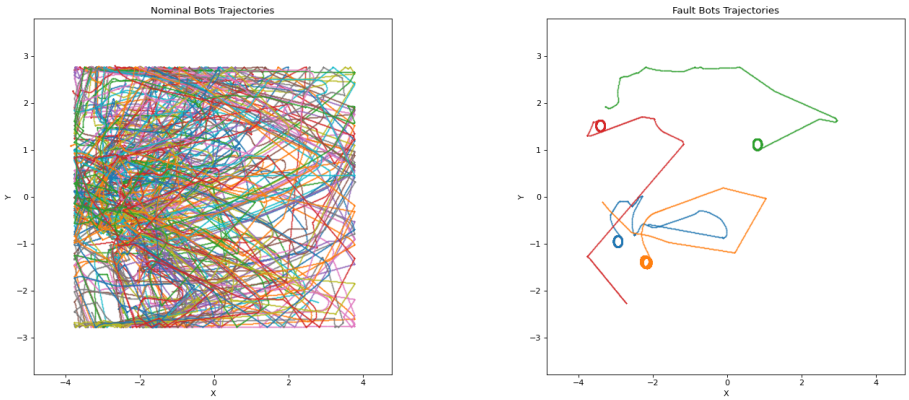
Figure 5.3: Foraging arena example.

start. An example of the robot trajectories can be seen in Figure 5.4 where nominal and faulty agents are separated in two different graphs.

RAWSim-O Fulfillment System

The RMFS task is set in a warehouse sized by the number of horizontal and vertical aisles that the robots use to move. In our experiments we used different arena sizes starting from 5×5 horizontal aisles $\times$ vertical aisles size through 8×8 horizontal aisles $\times$ vertical aisles, using each combination of the aisles number in $\{5, 6, 7, 8\}$, thus 16 different warehouse sizes. The environment objects are placed by the simulators, between each aisle there is a group of 10 pods locations that do not have to be all occupied. The simulations have a duration of 80000 seconds which approximately simulates the duration of a workday. The goal of the task is to follow the orders of the RMFS and carry pods from the storage area to the pick stations and replenishment stations. Based on how many horizontal aisles are included in the warehouse there is a corresponding number of pick stations and replenishment stations, the rule the compute the stations' number is:

$$\text{stations number} = \lfloor \frac{\text{horizontal aisles number}}{2} \rfloor.$$



(a) ARGoS foraging nominal bots trajectories.

(b) ARGoS foraging faulty bots trajectories.

Figure 5.4: ARGoS foraging plotted trajectories.

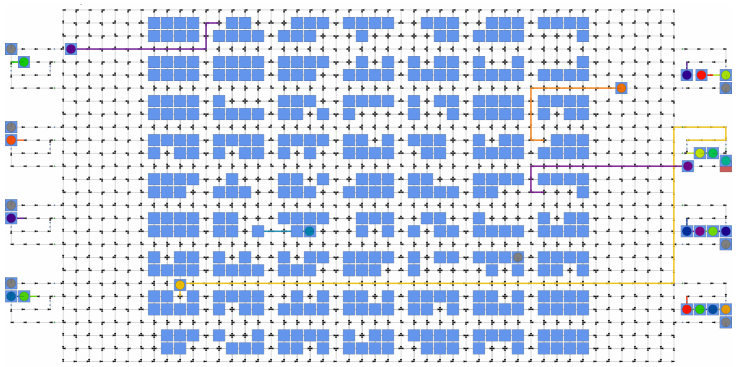
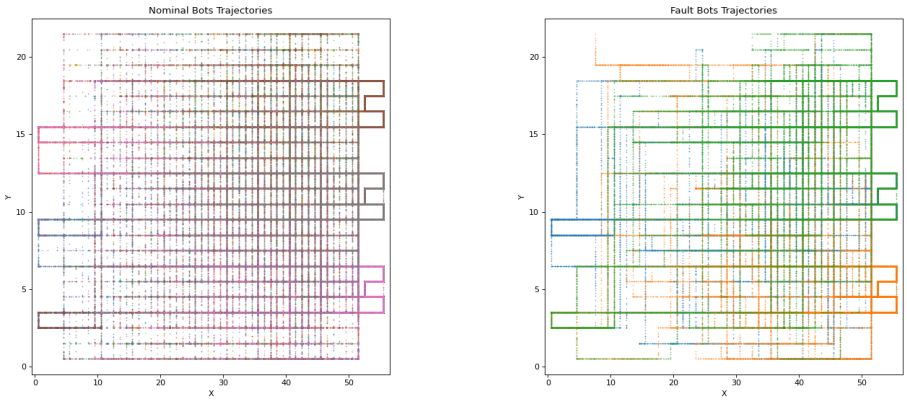


Figure 5.5: RMFS warehouse example.

The robot number starts from 15 agents and scales in a directly proportional way with the arena size. We can see an example of the warehouse environment in Figure 5.5.
For each warehouse size, a number of nominal simulations have been executed together with a number of simulations with injected faults. For the faulty simulations, we have considered 5%,10%, 20%, or 33% of the swarm population to be infected from the start of the task with a reduced speed of 10% or 5% of the maximum speed. An example of the robot trajectories can be seen in Figure 5.6 where nominal and faulty agents are separated in two different graphs.

5.2. Simulation Data Translation

During each simulation, the simulator is responsible for writing a `.csv` file where the information of every bot at every timestep is written. At the end of the simulation, we



(a) RAWSim-O nominal bots trajectories. (b) RAWSim-O faulty bots trajectories.

Figure 5.6: RAWSim-O plotted trajectories.

are provided with a `.csv` file with a structure identical to that of Table 5.1 but filled with values for each timestep and robot identification code. This step is supposed to simulate the external controller that observes the execution of the task and identifies each agent's position. The Fault value is necessary to perform supervised learning, in real-world scenarios it will not be provided.

timestep	ID	PosX	PosY	Fault
1	0	12.5606	-0.0025857	0
1	1	12.2117	-0.222327	0
...	...	...	...	...
1501	0	8.72475	-0.0530649	1
...	...	...	...	...
8000	0	8.66732	-0.0364418	1
...	...	...	...	...
8000	14	-0.0665298	0.290923	0

Table 5.1: Comma separated values file structure.

Feature	Description
Position	Coordinates of the agent in a single timestep
Direction	Directional vectors on x and y axis
Speed	Velocity of the agent between two consecutive timesteps
Cumulative speed	Sum of the last n timesteps velocity values
Neighbors number	Number of agent within a specified radius
Neighbors average distance	Average distance from all the bots in the swarm
Centroid distance	Distance from the coordinates of the swarm centroid
Cumulative centroid distance	Sum of the last n centroid distance values
Position entropy	Entropy measure of the last n positions
Area coverage	Percentage of area covered by the agent
Speed area coverage	Area coverage between two consecutive timesteps
Swarm position	Coordinates of the swarm centroid
Swarm speed	Velocity of the centroid between two consecutive timestep
Swarm area coverage	Percentage of area covered by the whole swarm

Table 5.2: Single bot list of features.

5.3. Feature Computation

The feature list has is shown in Table 5.2. In this section we will explain how each feature has been computed.

5.3.1. Position, Direction, Speed, Cumulative Speed

The position feature is the transposition of the `PosX` and `PosY` values into a `numpy` vector of size [timestep number, 2]. For future references, we will refer to this feature as $pos_{x,i}$ where x represents the x-axis and $i \in [0, \dots, T]$ is the timestep in a simulation of length T.
The direction feature is computed by subtracting the current position value from the previous position value for the x-axis and y-axis:

$$dir_{x,i} = pos_{x,i-1} - pos_{x,i}.$$

The speed feature is computed as the rate of change of position with respect to time:

$$v_i = \frac{\sqrt{(pos_{x,i-1} - pos_{x,i})^2 + (pos_{y,i-1} - pos_{y,i})^2}}{t_i - t_{i-1}} \tag{5.1}$$

with $v_0 = 0\frac{m}{s}$.
The cumulative speed feature for agent i at timestep t is computed as:

$$\text{cum-}v_{i,t} = \sum_{j=t-(\texttt{time-window})}^{T} v_{i,j}$$

5.3.2. Neighbors Number, Neighbors Average Distance

The neighbors number feature holds the information about how many other agent are within a predefined radius r. It is computed with the Algorithm 2 with N as the swarm cardinality, T as the total number of timestep. The neighbors average distance feature is

Algorithm 2 Neighbors Number Computation.

```
for t = 1 to T do
    temp-number=0
    for i = 1 to N do
        Compute d=distance between current agent and agent_i.
        if d < r then
            temp-number+=1
        end if
    end for
    neighbors-number_t=temp-number
end for
```

computed as the mean value among the distance from the current agent and all the other agents. It is computed with the Algorithm 3 with N as the swarm cardinality, T as the total number of timestep.

Algorithm 3 Neighbors Average Distance Computation.

```
for t = 1 to T do
    distances={}.
    for i = 1 to N do
        Compute d=distance between current agent and agent_i.
        Add d to distances.
    end for
    avg-d_t = (sum_{d in distances} d) / N
end for
```

5.3.3. Position Entropy

The position entropy feature computes the entropy of the positions in the last `time-window`×10 timesteps. The entropy is computed with `scipy.stats.entropy` [25] module that uses the following formula:

$$S = -\sum_{v \in Values} (P(v) * log(P(v)))$$

The entropy computation procedure is stated in Algorithm 4.

Algorithm 4 Position Entropy Computation.

for $t = 1$ to T **do**
  if $t <$ `time-window` **then**
    Compute vector of probabilities P for each in position in the last t positions.
    $entropy_t = -\sum_{p \in P} (p * log(p))$
  else
    Compute vector of probabilities P for each in position in the last `time-window` positions.
    $entropy_t = -\sum_{p \in P} (p * log(p))$
  end if
end for

5.3.4. Centroid Distance,Cumulative Centroid Distance

The centroid distance feature is the distance of the current agent from the centroid of the swarm. The centroid of the swarm is computed as the mean of the position of all the agents. Once the swarm centroid time series has been computed, each agent computes the distance between its current position and the position of the swarm centroid. The procedure is shown in Algorithm 5. The cumulative centroid distance feature for agent i

Algorithm 5 Centroid Distance Computation.

for $t = 1$ to T **do**
  $centroid\text{-}pos_t = \frac{\sum_{i \in N} pos_i}{|N|}$
end for
for $i = 1$ to N **do**
  for $t = 1$ to T **do**
    $centroid\text{-}distance_{i,t} = centroid\text{-}pos_t - pos_{i,t}$
  end for
end for

at timestep t is computed as:

$$\text{cum-}centroid\text{-}distance_{i,t} = \sum_{j=t-(\texttt{time-window})}^{t} centroid\text{-}distance_{i,j}$$

5.3.5. Area Coverage, Speed Area Coverage

The area coverage feature represents the area percentage covered by the agent until that exact moment. It is computed with 4 different levels of detail: 4, 16, 64, and 256 subdivision of the total area available. Area subdivisions are computed with the maximum and minimum positions covered by any robot in each axis. An approximate algorithm of the procedure is shown in Algorithm 6, the real implementation is optimized using `numpy` array and python functional programming with list comprehension.
Once the area borders have been computed, these values are used to compute the area coverage percentage of each agent. The area coverage computation procedure is shown in Algorithm 7.
The area coverage speed feature holds the information about the value increment in the area coverage timeseries for each timestep. The procedure to compute the area coverage speed feature is shown in Algorithm 8.

5.3.6. Swarm Position, Swarm Speed, Swarm Area Coverage

The swarm position feature represents the coordinates of the entire swarm, for this feature we've used the centroid of the positions of all the robots. The swarm position computation can be seen from line 1 to line 3 of Algorithm 5.
The swarm speed is computed as the speed feature for the single agent, in fact, we can refer to Equation 5.1 to compute the swarm speed considering the swarm centroid instead of the single agent position.
The swarm area coverage is considered as an overall area coverage performed by the whole swarm. The computation of the swarm area coverage feature is similar to the one shown in Algorithm 6 and Algorithm 7 with the only difference that, when we check if an area partition is visited, we check if any robot in the whole swarm is covering that area.

Algorithm 6 Area bounds computation.

Initialize variables to store positions. In this algorithm we will consider the order (agent-id, timestep-number,coordinate-axis) for the subscripts of *pos*
min-pos-x = $pos_{1,1,x}$
min-pos-x = $pos_{1,1,y}$
max-pos-x = $pos_{1,1,x}$
max-pos-x = $pos_{1,1,y}$
for $i = 1$ to N **do**
  for $t = 1$ to T **do**
    if min-pos-x $> pos_{i,t,x}$ **then**
      min-pos-x = $pos_{i,t,x}$
    end if
    if min-pos-y $> pos_{i,t,y}$ **then**
      min-pos-y = $pos_{i,t,y}$
    end if
    if max-pos-x $< pos_{i,t,x}$ **then**
      max-pos-x = $pos_{i,t,x}$
    end if
    if max-pos-y $< pos_{i,t,y}$ **then**
      max-pos-y = $pos_{i,t,y}$
    end if
  end for
end for
for $split \in \{2, 4, 8, 16\}$ **do**
  for $i = 1$ to $split$ **do**
    Compute segment$_{x,i}$ coordinates from the subdivision of segment [min-pos-x, max-pos-x] in equal *split* parts
    Compute segment$_{y,i}$ coordinates from the subdivision of segment [min-pos-y, max-pos-y] in equal *split* parts
    Save segments coordinates in an AreaPartition object.
    Save AreaPartition object in the area-subdivision list.
  end for
  Save area-subdivision list object in the area-partitions list.
end for

Algorithm 7 Area coverage computation.

Consider current agent i.
for area-subdivisions list in area-partitions list **do**
  timestep=0.
  while any AreaPartition not visited & timestep $< T$ **do**
    for area-partition in not visited area-partitions **do**
      if $pos_{i,t}$ is in area-partition **then**
        area-partition.visited=True.
      end if
    end for
    $\text{area-coverage}_{i,t} = \frac{\sum_{\text{area} \in \text{area-partitions-list}} I(\text{area.visited()==True})}{|\text{area-partitions-list}|}$
    timestep+=1.
  end while
  if iteration ends before visiting all timesteps **then**
    fill area-coverage_i timeseries with the last computed value.
  end if
end for

Algorithm 8 Area coverage speed computation.

Consider current agent i.
for $t = 1$ to T **do**
  $\text{coverage-speed}_i = \text{area-coverage}_{i,t} - \text{area-coverage}_{i,t-1}$
end for
Repeat for each area-subdivision level.

6 | Experiments

In this chapter, we will analyze the experiment we have made and discuss the measurements we have collected. The experiments will be analyzed from three different perspectives: feature analysis, performance metrics, and model performances. At the beginning we will analyze the feature time series values of nominal robots against faulty robots, we will then proceed on explaining which metrics we have used to judge the model performances, and, in the end, we will evaluate the model performances themselves.

6.1. Data Sets Analysis

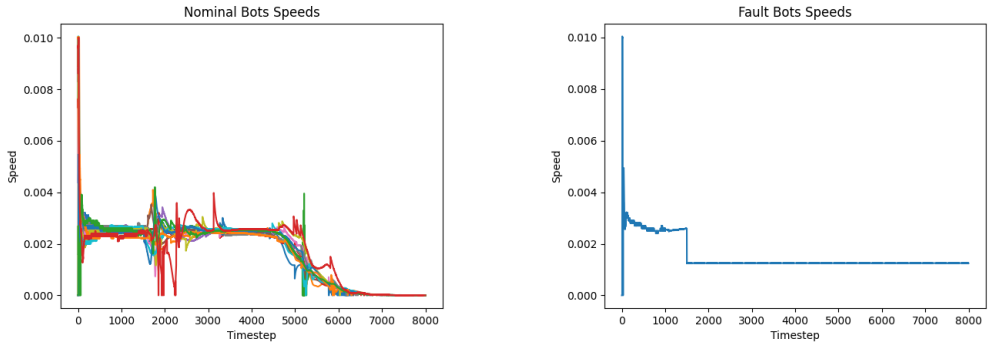
In this section we will graphically show the values assumed by each feature and if they present some distinguishing characteristics between nominal and faulty agents.

6.1.1. Flocking Features Analysis

The flocking task is explained in Section 4.1.3. The robot trajectories can be seen in Figure 5.2. From now on, we will analyze the features explained in Section 5.3. The direction feature will not be analyzed since it is unfeasible to show it graphically and useless to numerically analyze it. In this example, we will analyze a simulation of 15 agents with 1 faulty agent. In this section, we will analyze a flocking task starting from the East position. The fault is injected at 150 s and the robot rotates around a fixed point until the end of the simulation. For each second we have 10 timesteps.

Speed

The speed feature is one of the most characteristics since all of our faults influence this measure. It is possible to immediately distinguish the faulty robot behavior from the nominal robots. In the graphs in Figure 6.1 we can see that the faulty robot does not change speed after the fault is injected.



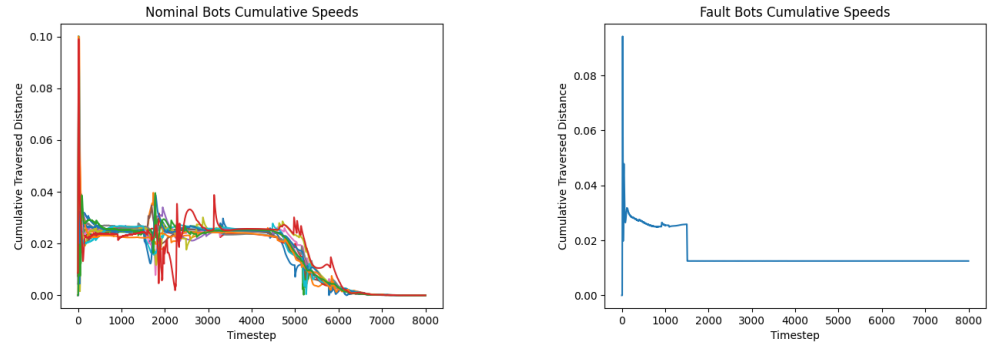
(a) Flocking nominal agents speed feature values.

(b) Flocking faulty agents speed feature values.

Figure 6.1: Flocking speed feature values.

Cumulative Speed

The cumulative speed feature is a rounded version of the speed feature that holds some information about the timestep in the `time-window`. This feature has identical characteristics to the speed feature and it is possible to distinguish the faulty robot at first sight. The `time-window` parameter is set to 10 and the values of the feature show the correlation between the two features.



(a) Flocking nominal agents cumulative speed feature values.

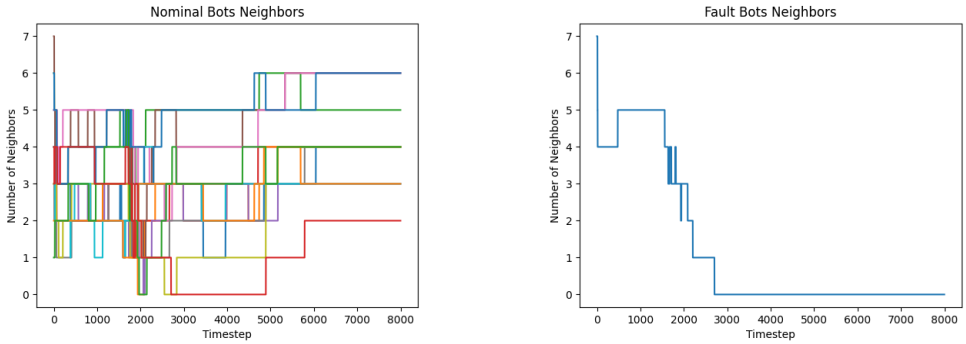
(b) Flocking faulty agents cumulative speed feature values.

Figure 6.2: Flocking cumulative speed feature values.

Neighbors Number

The neighbors number feature is very interesting for the flocking task since the agents have to keep a formation at a certain distance from one another. In Figure 6.3 we can

see that the faulty agent has a high number of neighbors at the start of the simulation, however, soon after the fault is injected, this value drastically drops denoting a separation of the agent from the swarm.
In this situation the neighborhood radius value has been set to 2 m. This value has to be fine tuned in order to have significant results.



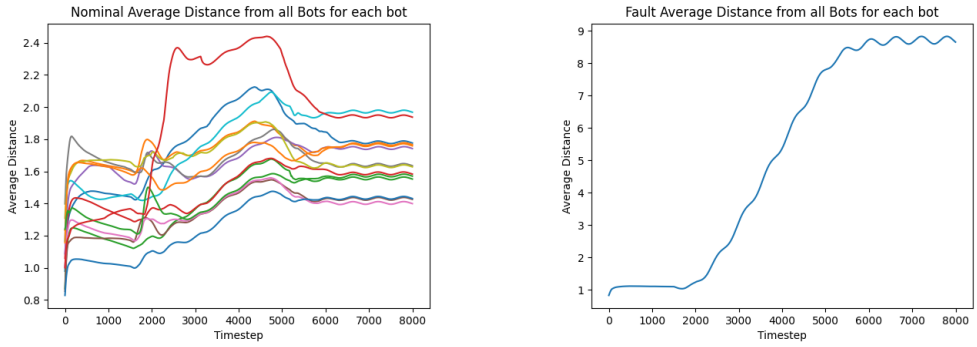
(a) Flocking nominal agents neighbors number feature values.

(b) Flocking faulty agents neighbors number feature values.

Figure 6.3: Flocking neighbors number feature values.

Neighbors Average Distance

The neighbors average distance feature highlights information similar to the neighbors number feature with opposite values, in this situation, the high value of the feature denotes a separation of the agent from the swarm. This feature is interesting for the flocking task, as the neighbors number feature, because it describes the behavior of the agent with respect to the other agents. In Figure 6.4 we can see that the faulty agent detaches from the swarm after the fault is injected.



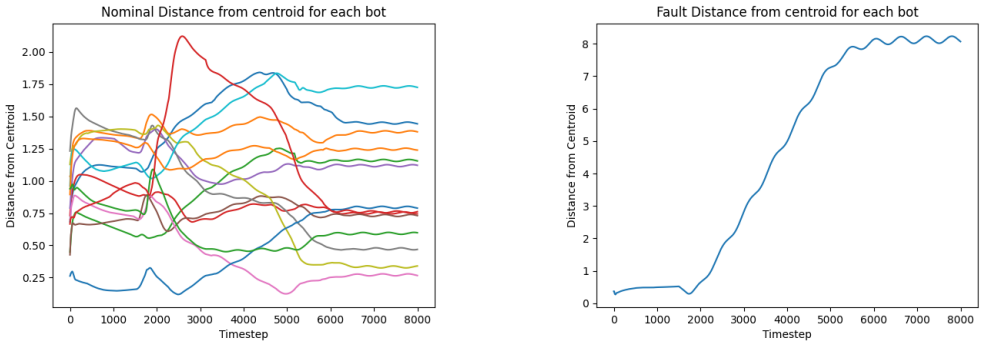
(a) Flocking nominal agents neighbors average distance feature values.

(b) Flocking faulty agents neighbors average distance feature values.

Figure 6.4: Flocking neighbors average distance feature values.

Centroid Distance

The centroid distance feature is a simplified version of the neighbors average distance feature since it is easier to compute. From the graphs in Figure 6.5 we can see that the values have a similar shape to the values shown in Figure 6.4 and show similar values, this is because the distance from the centroid can be seen as a mean value of the distance from the other agents.



(a) Flocking nominal agents centroid distance feature values.

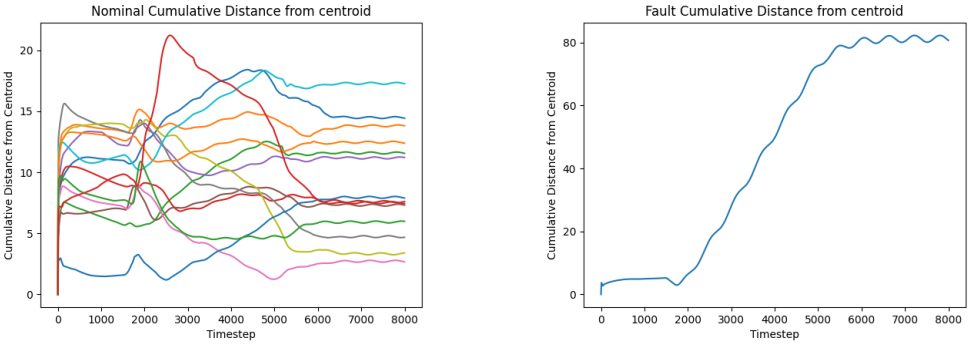
(b) Flocking faulty agents centroid distance feature values.

Figure 6.5: Flocking centroid distance feature values.

Cumulative Centroid Distance

The cumulative centroid distance feature is a feature that holds some information about the past values. The concept behind this feature is similar to the one of the cumulative

speed feature. Since this feature is a sum of the past values of the centroid distance feature, these two feature show high correlation and the values in the Figure 6.6 depicts the values of Figure 6.5 multiplied by the `time-window` parameter.



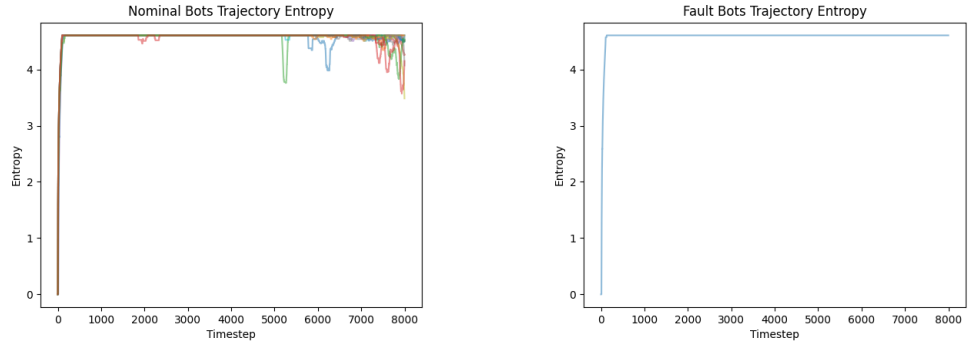
(a) Flocking nominal agents centroid distance feature values.

(b) Flocking faulty agents centroid distance feature values.

Figure 6.6: Flocking centroid distance feature values.

Position Entropy

The position entropy feature aims at identifying how many different positions are covered by the agent in the last `time-window` timesteps. The graphs shown in Figure 6.7 do not show any interesting information about the task execution since the faulty agent doesn't have different values than nominal agents. Like we said for the `neighborhood-radius` parameter, the `time-window` parameter has to be fine tuned for each different task such that the position entropy feature shows interesting values.



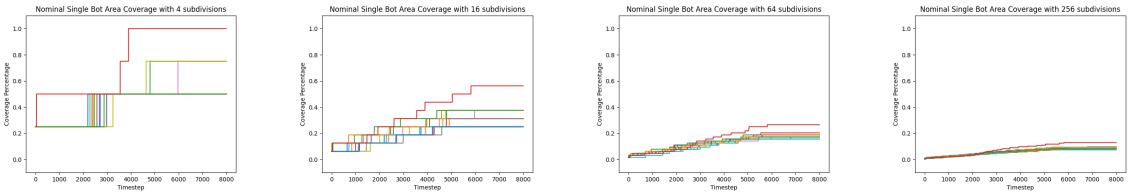
(a) Flocking nominal agents position entropy feature values.

(b) Flocking faulty agents position entropy feature values.

Figure 6.7: Flocking position entropy feature values.

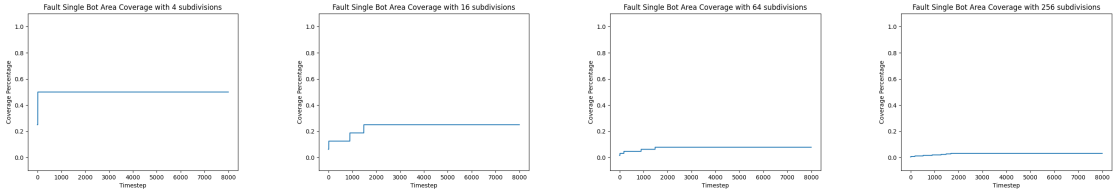
Area Coverage

The area coverage feature is divided into 4 different time series. The time series shown in Figure 6.8 depict the values assumed by the area coverage feature of the nominal agents with different levels of area partitioning. From the left, we can see the values in the case of 4 subdivisions, 16 subdivisions, 64 subdivisions, and then 256 subdivisions. We can see that the maximum value reached by the feature decreases with the increase of the area subdivisions, this is because, with the increase of the definition in area partitioning, some areas are too small to be covered by any agent.
The time series shown in Figure 6.9 depict the values assumed by the area coverage feature of the faulty agents. The graphs are organized like in Figure 6.8. The most important difference between nominal and faulty agents is that the faulty agent feature usually reaches lower values than the nominal agents.



(a) 4 subdivisions. (b) 16 subdivisions. (c) 64 subdivisions. (d) 256 subdivisions.

Figure 6.8: Flocking nominal agents area coverage feature values.

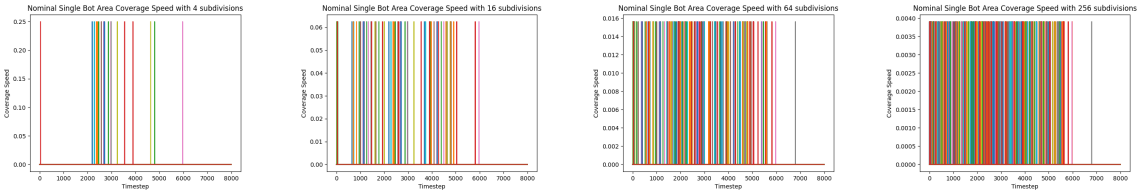


(a) 4 subdivisions. (b) 16 subdivisions. (c) 64 subdivisions. (d) 256 subdivisions.

Figure 6.9: Flocking faulty agents area coverage feature values.

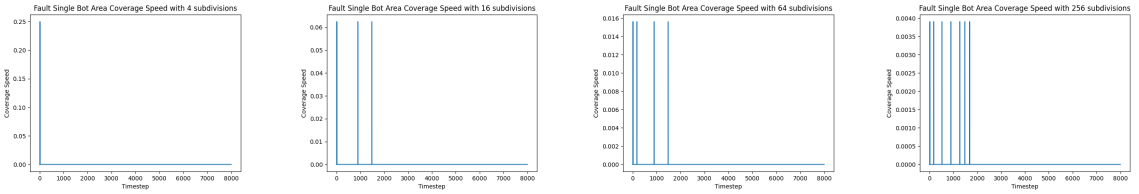
Area Coverage Speed

The area coverage speed feature holds information about the value increments of the area coverage feature. The graphs are organized like the ones of the area coverage feature. The most distinguishing aspect of this feature is that we can observe the faulty robot that stops exploring earlier than the nominal agents due to the injection of the fault.



(a) 4 subdivisions. (b) 16 subdivisions. (c) 64 subdivisions. (d) 256 subdivisions.

Figure 6.10: Flocking nominal agents area coverage speed feature values.



(a) 4 subdivisions. (b) 16 subdivisions. (c) 64 subdivisions. (d) 256 subdivisions.

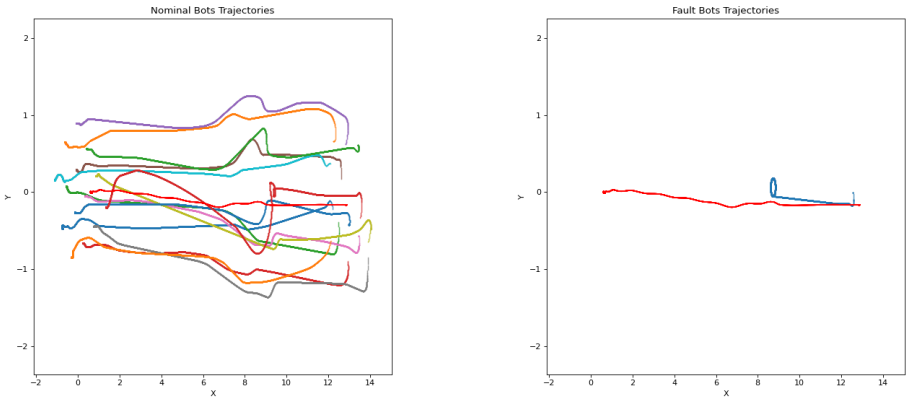
Figure 6.11: Flocking faulty agents area coverage speed feature values.

Global Features

The global features are those features that contain information about the overall behavior of the swarm in its entirety. These features should depict information about the execution of the task but do not contain information about the single agent, this characteristic is important because this kind of feature can not be considered fundamental for the objective of fault detection. The presence of these features is important whenever we use a model that can infer complex information from different features of the sample, i.e. convolutional neural networks.

Swarm Position

The swarm position feature is the coordinates of the centroid of the swarm. We can see the trajectory followed by the swarm centroid in Figure 6.12 as the bright red line in the middle of the plot among nominal agents (Figure 6.12a) and faulty agents (Figure 6.12b). The swarm position feature



(a) Swarm trajectory among nominal agents.

(b) Swarm trajectory among faulty agents.

Figure 6.12: Swarm trajectory.

Swarm Speed

The swarm speed feature represents the velocity of the swarm centroid. In Figure **??**, we can see that the values have a high spike at the beginning when the robots move to get in flocking formation. The feature value stays constant until 1500 timesteps, namely 150 s, and then slowly drops to lower values after the fault is injected. This feature behavior perfectly mirrors the swarm centroid movement that is slowed down by the faulty agent.

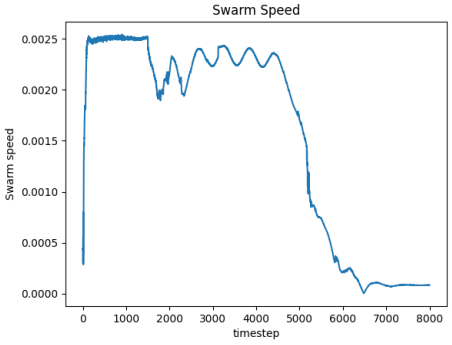
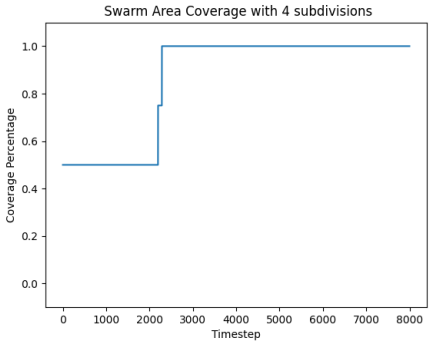


Figure 6.13: Swarm speed.

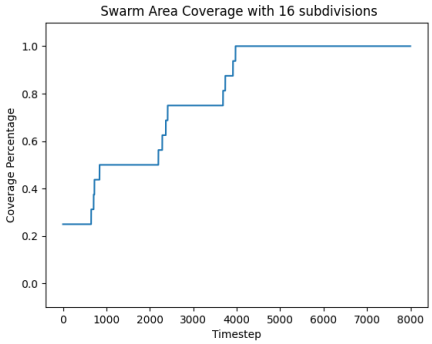
Swarm Area Coverage

The swarm area coverage features illustrate the percentage of area covered by the entirety of the swarm considered each agent. This feature is similar to the area coverage feature seen before in section Area Coverage. As stated above, this is a global feature and does

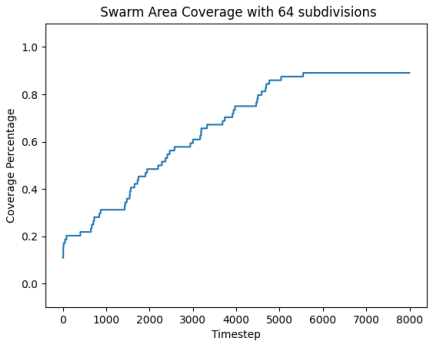
not have the objective to directly identify the fault injection. Nonetheless, the graphs shown in Figure 6.14 do not shown any sign of the presence of the fault injected after 150 s.



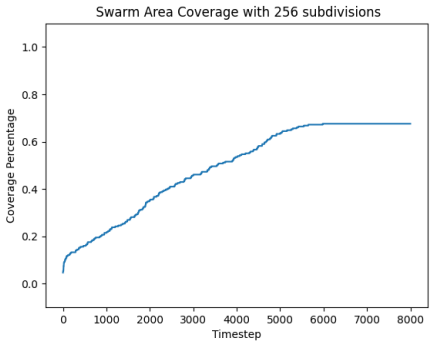
(a) Flocking swarm 4 subdivisions area coverage feature values.



(b) Flocking swarm 16 subdivisions area coverage feature values.



(c) Flocking swarm 64 subdivisions area coverage feature values.



(d) Flocking swarm 256 subdivisions area coverage feature values.

Figure 6.14: Flocking swarm area coverage feature values.

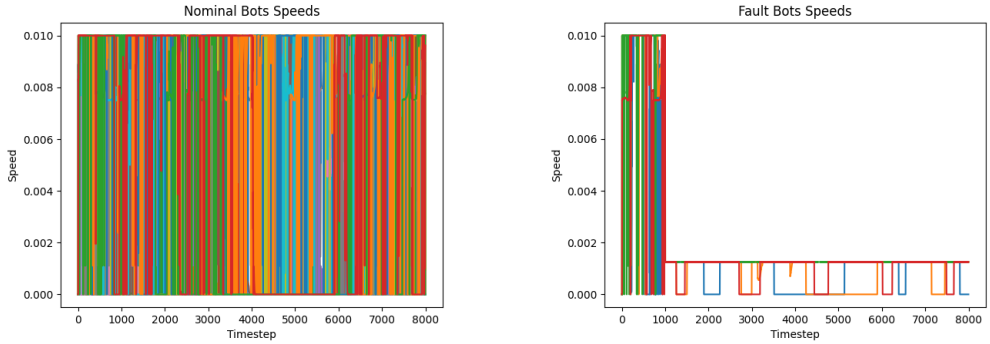
6.1.2. Foraging Features Analysis

The foraging task is explained in Section 4.1.4. The robot trajectories can be seen in Figure 5.4. In this example, we will analyze a simulation of 38 agents with 4 faulty agents. The fault is injected at 150 s and the robots rotate around a fixed point until the end of the simulation. For each second we have 10 timesteps.

Speed

As stated for the flocking task, this feature is highly significant due to the nature of the faults injected. In Figure 6.15 we can see that the faulty robots rotate do not reach high

speeds after the fault is injected but their speed oscillates at values lower than the nominal agents. The oscillations of faulty agents' speed can be traced back to obstacle avoidance maneuvers or blockages in the nest area.



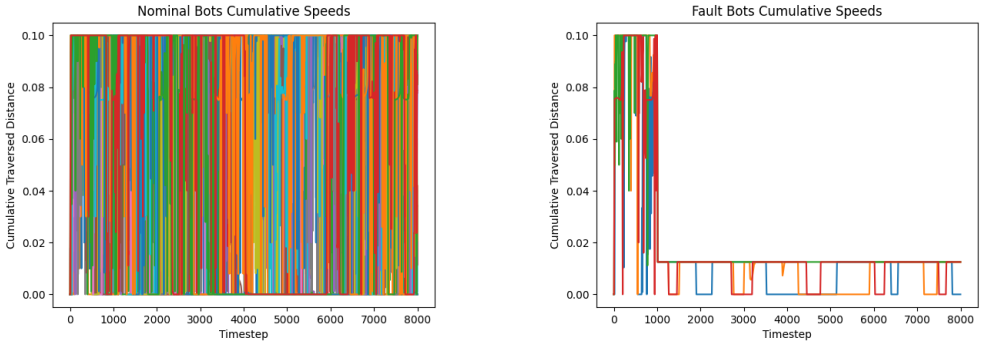
(a) Foraging nominal agents speed feature values.

(b) Foraging faulty agents speed feature values.

Figure 6.15: Foraging speed feature values.

Cumulative Speed

This feature has identical characteristics to the speed feature and it is possible to distinguish the faulty robot at first sight. The `time-window` parameter is set to 10 and the features values show the correlation between the two features.



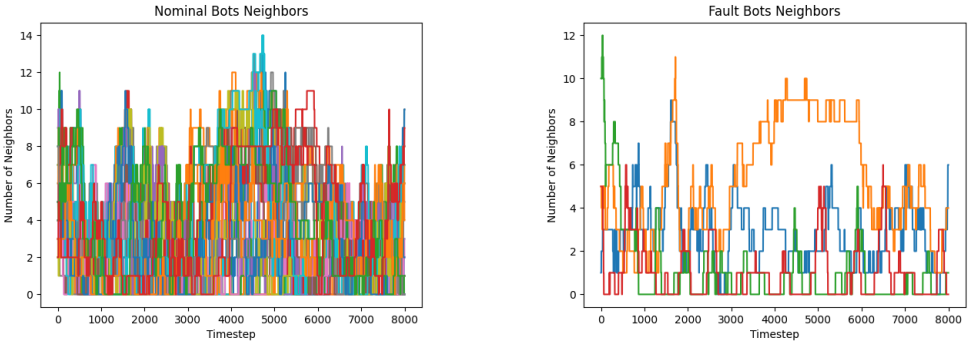
(a) Foraging nominal agents cumulative speed feature values.

(b) Foraging faulty agents cumulative speed feature values.

Figure 6.16: Foraging cumulative speed feature values.

Neighbors Number

The neighbors number feature is not very interesting for the foraging task since the agents spend most of the time wandering around. In Figure **??** we can see that the faulty agents have similar values with respect to the nominal agents.
In this situation the neighborhood radius value has been set to 2 m. This value has to be fine tuned in order to have significant results.



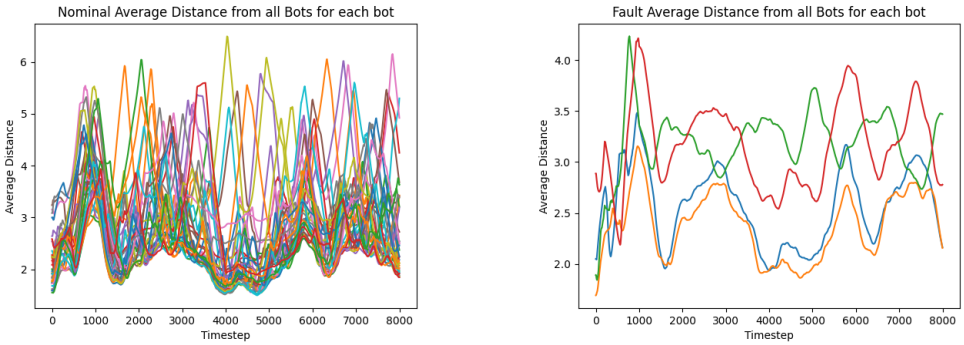
(a) Foraging nominal agents neighbors number feature values.

(b) Foraging faulty agents neighbors number feature values.

Figure 6.17: Foraging neighbors number feature values.

Neighbors Average Distance

The neighbors average distance feature highlights information similar to the neighbors number feature with opposite values, in this situation, the high value of the feature denotes a separation of the agent from the swarm. As stated before, this feature does not contain interesting information due to the nature of the task. In Figure 6.18 we see that faulty agent and nominal agents have mostly similar values.



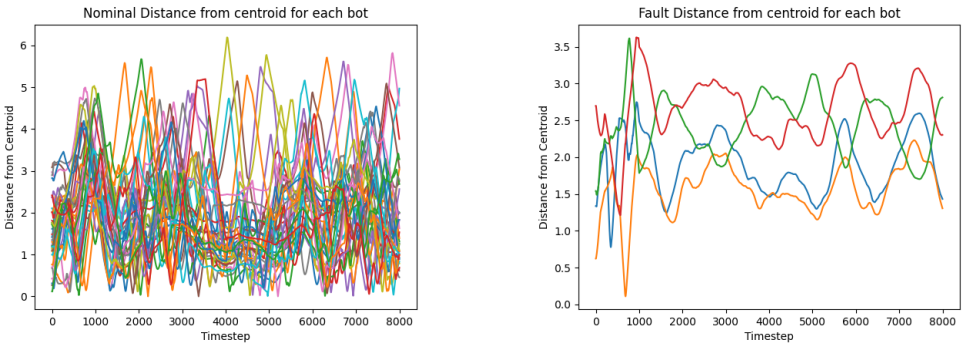
(a) Foraging nominal agents neighbors average distance feature values.

(b) Foraging faulty agents neighbors average distance feature values.

Figure 6.18: Foraging neighbors average distance feature values.

Centroid Distance

From the graphs in Figure 6.19 we can see that the values have a similar shape to the values shown in Figure 6.18 and show similar values, this is because the distance from the centroid can be seen as a mean value of the distance from the other agents.



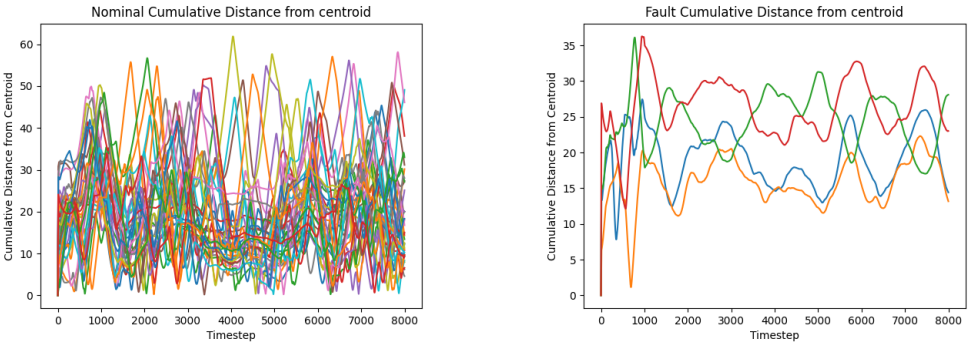
(a) Foraging nominal agents centroid distance feature values.

(b) Foraging faulty agents centroid distance feature values.

Figure 6.19: Foraging centroid distance feature values.

Cumulative Centroid Distance

Similarly to the flocking task, these two feature show high correlation and the values in the Figure 6.20 depicts the values of Figure 6.19 multiplied by the `time-window` parameter.



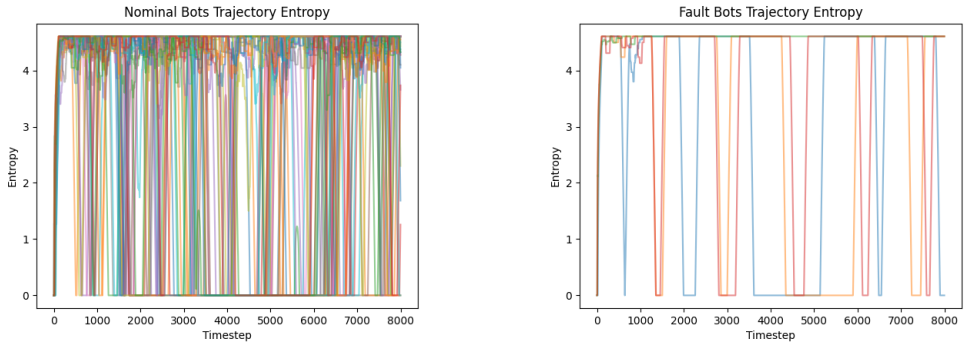
(a) Foraging nominal agents centroid distance feature values.

(b) Foraging faulty agents centroid distance feature values.

Figure 6.20: Foraging centroid distance feature values.

Position Entropy

The graphs shown in Figure 6.21 shows more different values than the graph in Figure 6.7. The values of this features depends on the tuning of the `time-window` parameter.



(a) Foraging nominal agents position entropy feature values.

(b) Foraging faulty agents position entropy feature values.

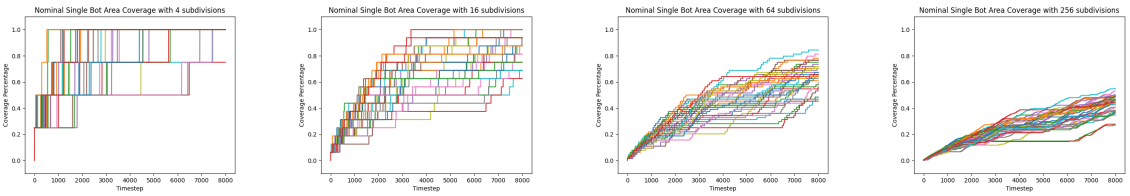
Figure 6.21: Foraging position entropy feature values.

Area Coverage

The area coverage feature is identically organized as the flocking task. We can see that the maximum value reached by the feature decreases with the increase of the area subdivisions, this is because, with the increase of the definition in area partitioning, some areas are too small to be covered by any agent.

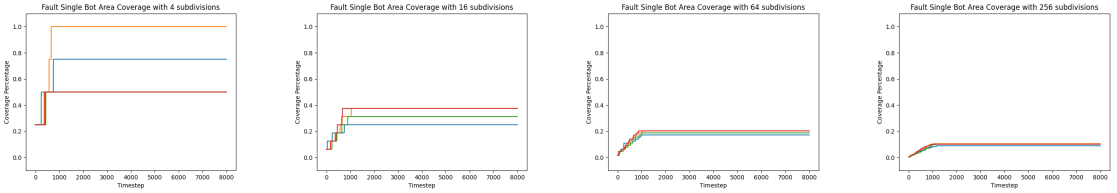
The most important difference between nominal and faulty agents is that the faulty agent

feature reaches lower values than the nominal agents.



(a) 4 subdivisions. (b) 16 subdivisions. (c) 64 subdivisions. (d) 256 subdivisions.

Figure 6.22: Foraging nominal agents area coverage feature values.

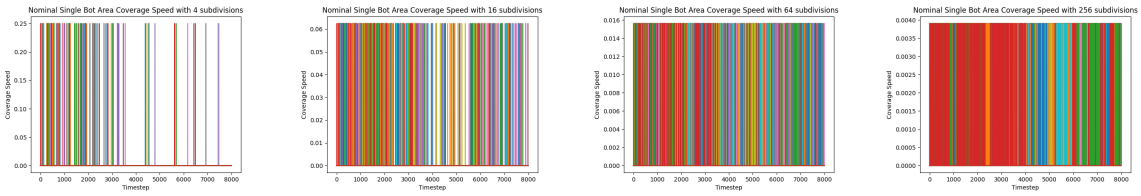


(a) 4 subdivisions. (b) 16 subdivisions. (c) 64 subdivisions. (d) 256 subdivisions.

Figure 6.23: Foraging faulty agents area coverage feature values.

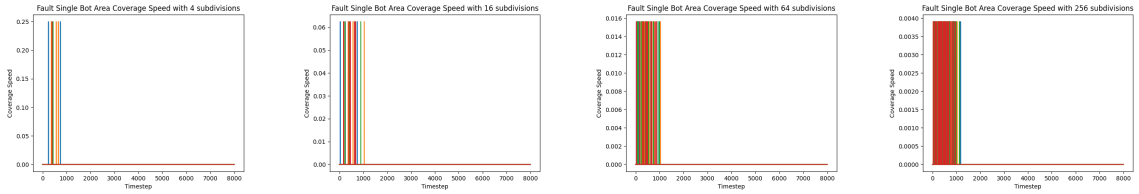
Area Coverage Speed

The graphs are organized like the ones of the area coverage feature. The most distinguishing aspect of this feature is that we can observe the faulty robot that stops exploring earlier than the nominal agents due to the injection of the fault.



(a) 4 subdivisions. (b) 16 subdivisions. (c) 64 subdivisions. (d) 256 subdivisions.

Figure 6.24: Flocking nominal agents area coverage speed feature values.



(a) 4 subdivisions. (b) 16 subdivisions. (c) 64 subdivisions. (d) 256 subdivisions.

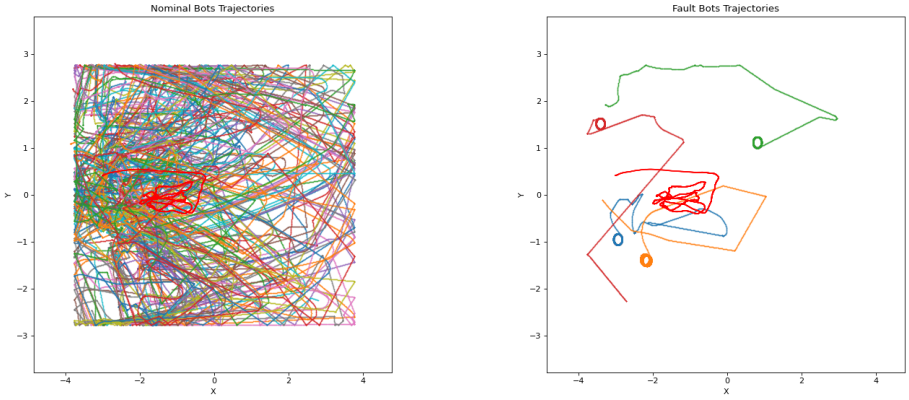
Figure 6.25: Foraging faulty agents area coverage speed feature values.

Global Features

The global features have been described in the previous task.

Swarm Position

The swarm position feature is the coordinates of the centroid of the swarm. We can see the trajectory followed by the swarm centroid in Figure 6.12 as the bright red line in the middle of the plot among nominal agents (Figure 6.12a) and faulty agents (Figure 6.12b).

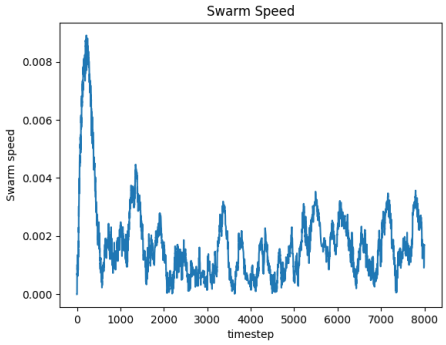


(a) Swarm trajectory among nominal agents. (b) Swarm trajectory among faulty agents.

Figure 6.26: Swarm trajectory.

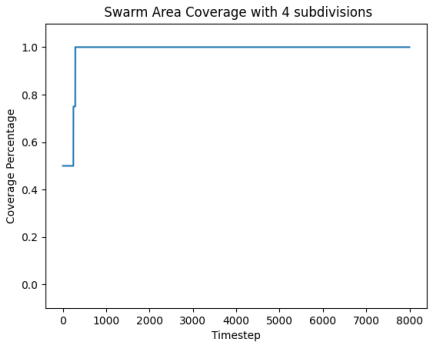
Swarm Speed

The swarm speed feature represents the velocity of the swarm centroid. In Figure **??** we can see that the values have a high spike at the beginning, when most of the robots start exploring. Despite this feature showing exactly the fault injection in Figure **??**, in this situation we are unable to observe the fault injection moment due to the nature of the task that does not have a homogeneous behavior.

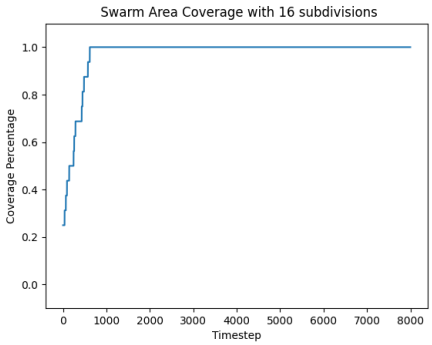


Swarm Area Coverage

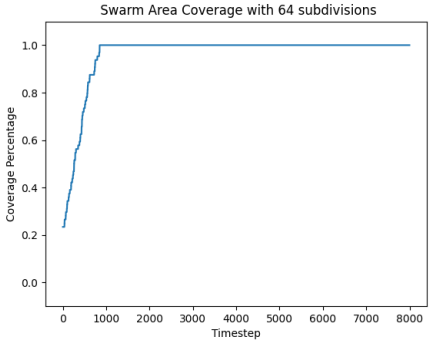
The swarm area coverage features illustrate the percentage of area covered by the entirety of the swarm considered each agent. This feature is similar to the area coverage feature seen before in section Area Coverage. As stated above, this is a global feature and does not have the objective to directly identify the fault injection. Nonetheless, the graphs shown in Figure 6.14 do not shown any sign of the presence of the fault injected after 150 s.



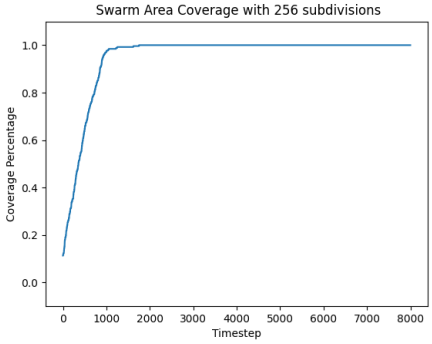
(a) Foraging swarm 4 subdivisions area coverage feature values.



(b) Foraging swarm 16 subdivisions area coverage feature values.



(c) Foraging swarm 64 subdivisions area coverage feature values.



(d) Foraging swarm 256 subdivisions area coverage feature values.

Figure 6.27: Foraging swarm area coverage feature values.

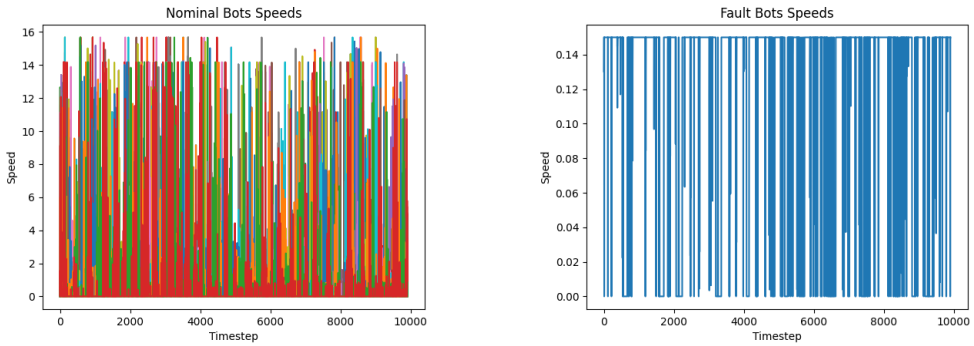
6.1.3. RMFS Features Analysis

The RMFS task of the RAWSim-O simulator is explained in Section 4.1.5. The robot trajectories can be seen in Figure 5.6. In this example, we will analyze a simulation of 25 agents with 1 faulty agent. The fault is injected from the beginning of the simulation and the faulty robot travels at 5% of the maximum speed until the end of the simulation. The positions of all the robots are recorded only when a robot executes its `Act()` function, since this function is not called at every timestep, the time series data points are not evenly distributed in time.

Speed

As stated before, the speed feature is highly discriminating and we can observe this characteristic in the graph in in Figure 6.28. We can see that the faulty robot does not

reach high speeds like nominal robots.



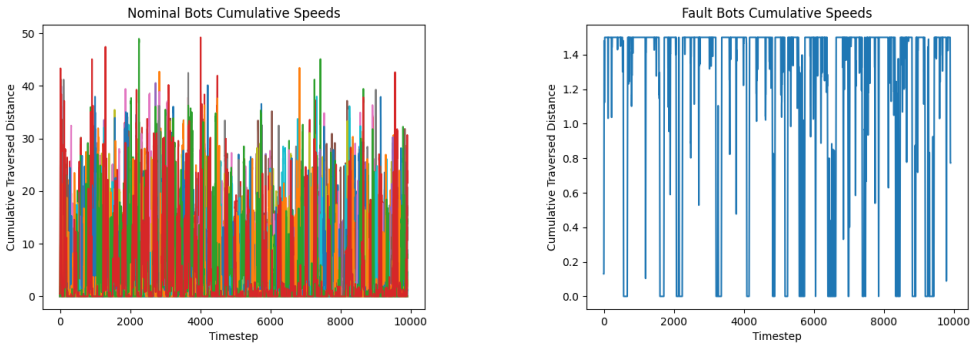
(a) RMFS nominal agents speed feature values.

(b) RMFS faulty agents speed feature values.

Figure 6.28: RMFS speed feature values.

Cumulative Speed

From the graphs in Figure 6.29 we can appreciate the same characteristic of the graphs in Figure 6.28 with fewer spikes and higher values. The speed graphs do not show any discriminating information other than the maximum speed value a robot can reach.



(a) RMFS nominal agents cumulative speed feature values.

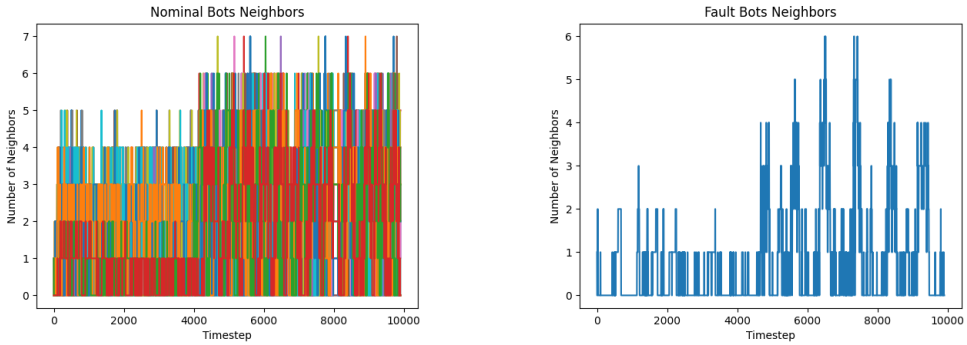
(b) RMFS faulty agents cumulative speed feature values.

Figure 6.29: RMFS cumulative speed feature values.

Neighbors Number

The neighbors number feature does not show any particular difference between nominal robots and faulty robots. Like we have seen in Figure **??** for the foraging task, the faulty robots have similar values to nominal ones. The RMFS task does not have the same

behavior as the foraging task where robots wander around performing a random walk, in this situation the agents have to follow specific routes and reach specific locations, however, the continuous movements in the warehouse force the robots to visit crowded locations with the same chance of visiting locations with any other robot.
In this situation the neighborhood radius value has been set to 2 m. This value has to be fine tuned in order to have significant results.



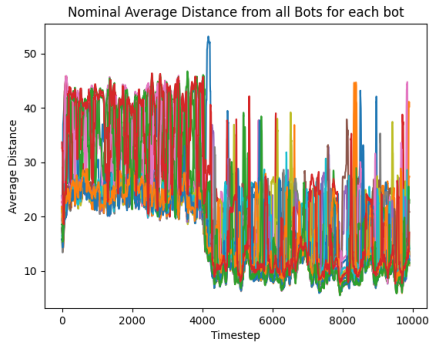
(a) RMFS nominal agents neighbors number feature values.

(b) RMFS faulty agents neighbors number feature values.

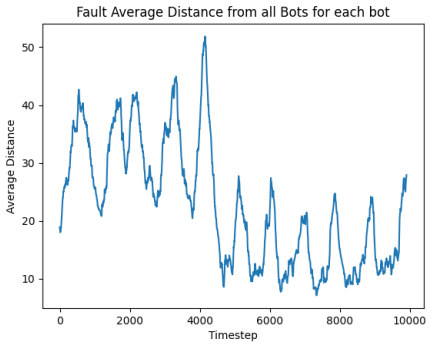
Figure 6.30: RMFS neighbors number feature values.

Neighbors Average Distance

The graphs shown in Figure 6.31 have some similarities with the neighbors number feature. Neighbors average distance and the number of neighbors feature are inversely correlated but depict the same behaviors. In Figure 6.31b we can assume that the lowest values are reached when the robot approaches the pick stations. Instead, the higher values depict situations in which the robot is searching for a pod in less visited regions. From the two graphs, we can observe two phases, a first phase in which the feature assumes higher values and a second phase in which the feature assumes on average lower values. This behavior should be analyzed with a complete recording of the simulation, however, we can assume that these values depict a first phase in which the robots visited both pick and replenishment stations, ending in covering a broader area of the warehouse, and a second phase where the robots were closer together may be due to a congestion on the replenishment station or due to an excess of replenishment orders.



(a) RMFS nominal agents neighbors average distance feature values.

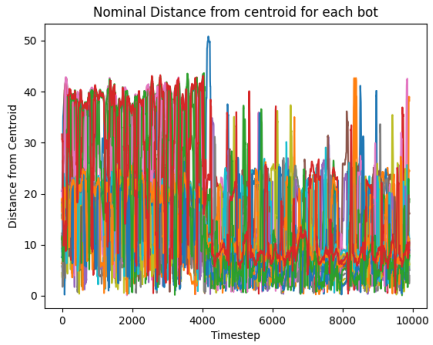


(b) RMFS faulty agents neighbors average distance feature values.

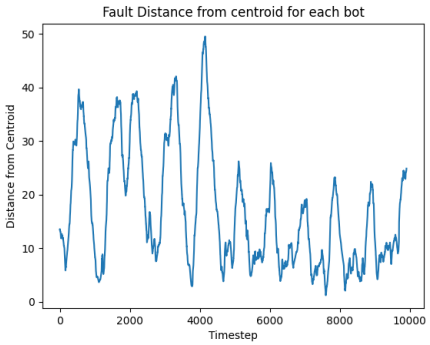
Figure 6.31: RMFS neighbors average distance feature values.

Centroid Distance

In Figure 6.32 we can observe the same behavior of the neighbors average distance feature shown in Figure 6.31. The only difference we can see between the two features is that the centroid distance shows higher oscillations and less distinguished phases. We could note that the faulty agents have fewer spikes with respect to the nominal agents, we can assume this is a consequence of the lower speeds of faulty robots but it is necessary to implement a model able to interpret time series to exploit these characteristics.



(a) RMFS nominal agents centroid distance feature values.

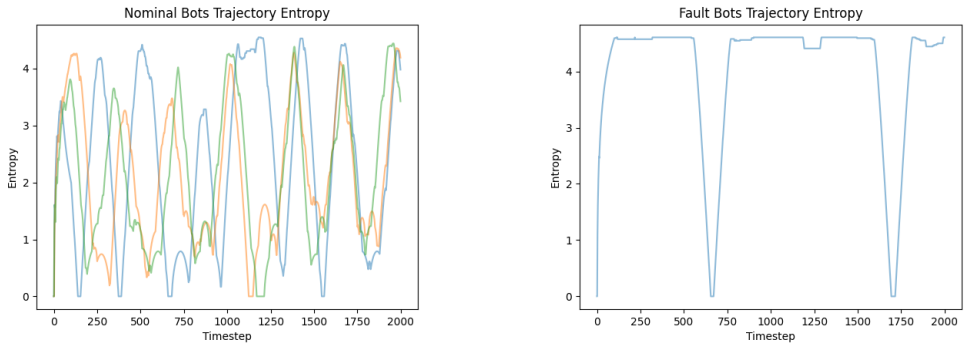


(b) RMFS faulty agents centroid distance feature values.

Figure 6.32: RMFS centroid distance feature values.

Position Entropy

In Figure 6.33 we analyze a reduced time series with fewer nominal bots to underline the differences between nominal and faulty agents. In particular, we can see that the nominal robots have a higher number of spikes, as stated before this is a consequence of the difference in speed. In Figure 6.33 we can see that the faulty agent reaches higher values on average than its nominal counterparts. This is an undesired behavior since the slowest robot should visit fewer locations and it should not have high entropy values. However, this feature characteristic is a direct consequence of the `time-window` parameter, confirming once again that each parameter has to be fine tuned for each specific task.



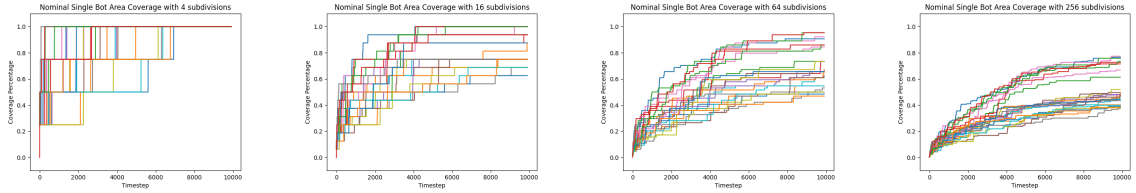
(a) RMFS nominal agents position entropy feature values.

(b) RMFS faulty agents position entropy feature values.

Figure 6.33: RMFS position entropy feature values.

Area Coverage

The area coverage feature does not depict any new behavior with respect to the graphs analyzed in Figure 6.8 or Figure 6.22. The only clear difference we can appreciate is between Figure 6.34c and Figure 6.35c or between Figure 6.34d and Figure 6.35d where the faulty agent reaches lower values on average than the nominal agents.



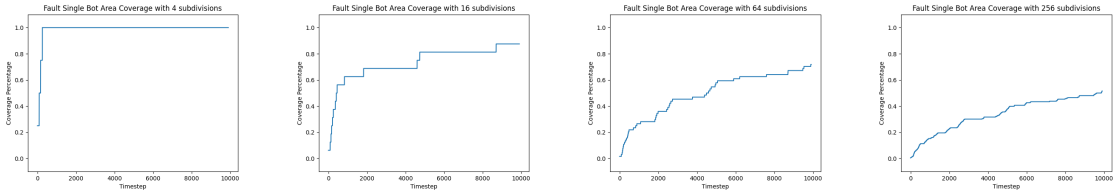
(a) 4 subdivisions.

(b) 16 subdivisions.

(c) 64 subdivisions.

(d) 256 subdivisions.

Figure 6.34: RMFS nominal agents area coverage feature values.



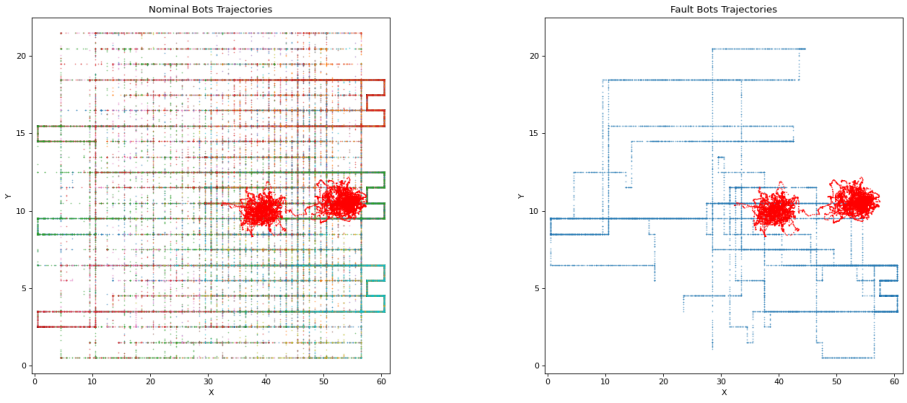
(a) 4 subdivisions. (b) 16 subdivisions. (c) 64 subdivisions. (d) 256 subdivisions.

Figure 6.35: RMFS faulty agents area coverage feature values.

Global Features

Swarm Position

The swarm position feature is the coordinates of the centroid of the swarm. We can see the trajectory followed by the swarm centroid in Figure **??** as the bright red line in the middle of the plot among nominal agents (Figure 6.36a) and faulty agents (Figure 6.36b). Since the data is not evenly distributed in time and we do not have a continuous representation of the robot movements, the swarm position is presented as a set of points mixed with robots trajectories. In particular, we can see that are two main points aggregations, one closer to the middle of the warehouse and one closer to the pick stations. These positions reflect the situation observed for the neighbors average distance graphs in Figure 6.31.



(a) Swarm trajectory among nominal agents. (b) Swarm trajectory among faulty agents.

Figure 6.36: RMFS Swarm trajectory.

6.2. Metrics

In this section, we will introduce the metrics used to evaluate the model performance and how to interpret them.

6.2.1. Confusion Matrix

The distinction between nominal and faulty agents is a classification task that falls into the anomaly detection category. Most of the time, anomaly detection implies the presence of an unbalanced set since it is easy to imagine that the anomalies should be less frequent than the nominal data samples. In this situation, analyzing the model performances solely from the accuracy (Equation 6.1) values can be misleading.

$$\text{classification accuracy} = \frac{\text{correct predictions}}{\text{total predictions}} \tag{6.1}$$

If we imagine a dataset with very few samples belonging to the faulty class, a trivial model that predicts only the nominal class should obtain high accuracy performance even though it is not rightfully classifying the faulty data samples. To overcome this difficulty we use the confusion matrix which highlights the number of samples classified correctly for each class. The goal of the confusion matrix is to show how confused is our model while making predictions on the test dataset. The confusion matrix used in our experiments is organized with the actual class instances in the rows and the predicted class instances in the columns, we consider faulty agents for the positive class and negative agents for the negative class. We can see an example of the confusion matrix structure in Table 6.1. More precisely, we can see each cell in the matrix as:

- **True positive:** correctly predicted event values.
- **False positive:** incorrectly predicted event values.
- **True negative:** for correctly predicted no-event values.
- **False negative:** for incorrectly predicted no-event values.

In particular, a good performance indicator would be to have the majority of the classified samples on the diagonal of the matrix, namely in the true negative and true positive regions.
Furthermore, to extract more information from the confusion matrix, we compute the

True Label	Predicted Label: n	Predicted Label: p
n′	True Negative	False Positive
p′	False Negative	True Positive

Table 6.1: Confusion matrix structure.

recall, precision, and f1-score indexes. The precision is defined as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positive} + \text{False Positives}}.$$

This measure is used to " evaluate the fraction of correct classified instances among the ones classified as positive " [15, pg.52]. The precision values range from 0.0 to 1.0 with 0.0 denoting precision absence and 1.0 denoting perfect precision.
The recall is defined as:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positive} + \text{False Negatives}}$$

and it quantifies the number of correct positive predictions retrieved from all the positive predictions that are possible. The recall values range from 0.0 to 1.0 and have the same behavior of precision values.
The use of these two metrics depends on what is the objective of the classifier, if the classifier has to minimize the number of false positives, then it must have high precision, if the classifier has to minimize the number of false negatives, then it must have a high recall.
In order to combine the precision and recall measure, we compute the F1-score. This metric is the harmonic mean of precision and recall and it is computed with the following formula:

$$\text{F1-score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

The F1-score can assume values from 0.0 to 1.0 and, like all the previous measures, 0.0 identifies poor performances while 1.0 identifies perfect performances.

6.2.2. Precision Recall Curve

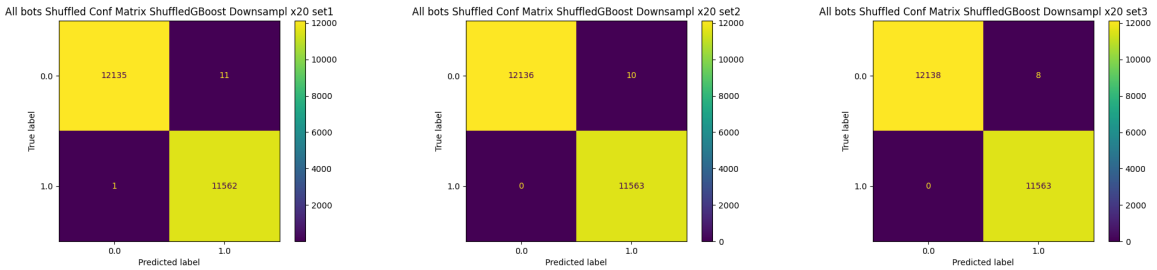
In classification tasks, it is useful to measure how the performances change when using different thresholds to distinguish a sample between positive and negative classes. In binary classification, we can imagine the threshold as the numeric boundary that draws a line between the negative and positive classes. The receiver operating characteristic (ROC) curve and the precision recall curve are two useful metrics to evaluate the classification performance of a model when using different thresholds. In this section, we will explain how the precision-recall curve works and how it can be interpreted.
We have to underline that precision and recall are not directly correlated. If we try to increase the precision of a model by lowering the threshold to identify more positive events, this does not imply that the number of false negatives will decrease so that the recall will increase.
The precision recall curve graphs will also indicate the average precision (AP) measure. The average precision is defined as follows:

$$AP = \sum_{n} (R_n - R_{n-1}) P_n$$

where P_n and R_n are the precision and recall at the n-th threshold. This measure identifies the area subtended by the precision recall curve. The AP will range from 0.0 to 1.0, values lower than 0.5 usually mean that the model is performing worse than a random guesser or a trivial model predicting only one class, values higher than 0.5 mean that the model is able to classify and distinguish a fair amount of samples. A model with perfect performances will have an AP value around 1.0 while a model with values around 0.0 usually presents some errors in the parameters or is trying to classify corrupted data.

6.3. Gradient Boosting Performance Evaluation

In this section, we will analyze the performance of our model on the fault detection for the 3 different tasks introduced in Section 5.1.1. The performances are divided into 3 different sets of features, these sets aim at collecting different points of view. Even though the goal of our work is to perform fault detection from an external point of view, it is useful to analyze different sets in order to understand which effects and consequences they could have in other approaches.
The first set represents the point of view of the single agent, in this set we have included the following features: position, speed, direction, cumulative speed, position entropy, area coverage, and area coverage speed.



(a) Feature set 1. (b) Feature set 2. (c) Feature set 3.

Figure 6.37: Confusion matrices of the gradient boosting model performances on the flocking task.

The second set represents the point of view of the single with the acknowledgment of being surrounded by other robots. This feature set includes the following features: position, speed, direction, cumulative speed, neighbors number, neighbors average distance, centroid distance, cumulative centroid distance, position entropy, area coverage, and area coverage speed.
The third and last set represents the point of the single agent with complete knowledge of its surroundings and the agents of the swarm. This feature set includes the following features: position, speed, direction, cumulative speed, neighbors number, neighbors average distance, centroid distance, cumulative centroid distance, position entropy, area coverage, area coverage speed, swarm centroid positions, swarm speed, and swarm area coverage.

6.3.1. Flocking Fault Detection

The flocking task simulations we will analyze in this section are described in Section 4.1.3. In section 5.1.1, we have described a summary of the simulation parameters we have used to collect all the simulations used in the dataset. In general, the overall procedure to execute all the simulations is explained in Algorithm 9. It is implied that every time a simulation is executed its log files are automatically collected. Once the simulation positions with the corresponding fault flag have been collected they are processed the build each feature time series.

Confusion Matrix

A

Algorithm 9 Neighbors Number Computation.

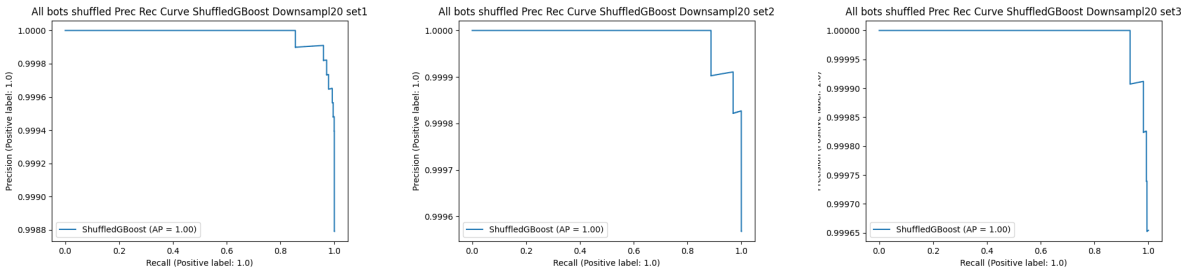
```
nominal-repetitions=15.
positions= { North, South, East, West }.
for i = 1 to nominal-repetitions do
    for position in positions do
        execute nominal simulation from position.
    end for
end for
faults= {10%, 20%, 33%}.
single-bot-fault-repetitions = 4.
for i = 1 to single-bot-fault-repetitions do
    for position in positions do
        execute single bot fault simulation from position.
    end for
end for
for fault-percentage in faults do
    for position in positions do
        execute fault-percentage simulation from position.
    end for
end for
```

Precision Recall Curve

A

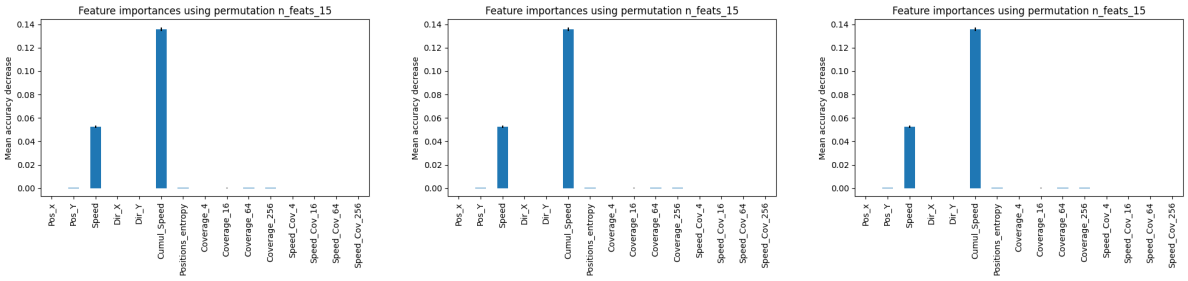
Feature Permutation Results

A



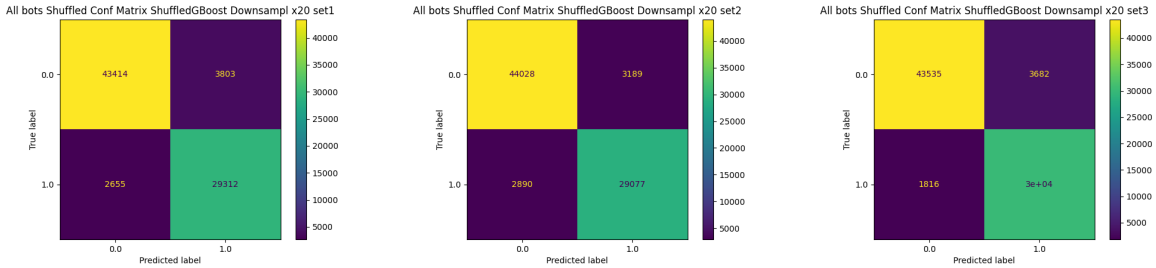
(a) Feature set 1. (b) Feature set 2. (c) Feature set 3.

Figure 6.38: Precision recall curves of the gradient boosting model performances on the flocking task.



(a) Feature set 1. (b) Feature set 2. (c) Feature set 3.

Figure 6.39: Feature permutation results on the flocking task.



(a) Feature set 1. (b) Feature set 2. (c) Feature set 3.

Figure 6.40: Confusion matrices of the gradient boosting model performances on the foraging task.

6.3.2. Foraging Fault Detection

A

Confusion Matrix

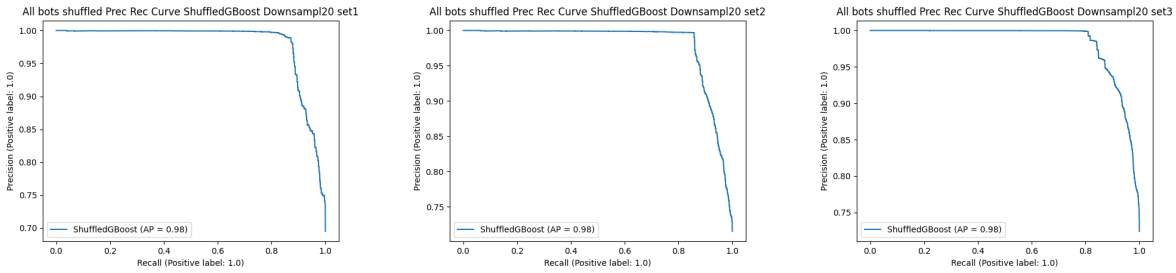
A

Precision Recall Curve

A

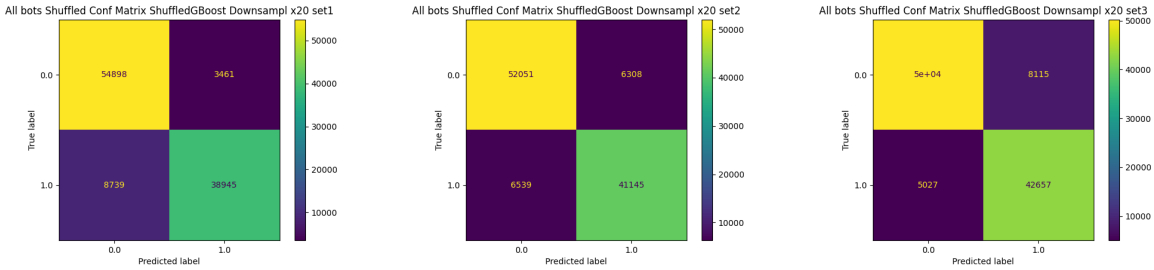
6.3.3. RMFS Fault Detection

A



(a) Feature set 1. (b) Feature set 2. (c) Feature set 3.

Figure 6.41: Precision recall curves of the gradient boosting model performances on the foraging task.



(a) Feature set 1. (b) Feature set 2. (c) Feature set 3.

Figure 6.42: Confusion matrices of the gradient boosting model performances on the RMFS task.

Confusion Matrix

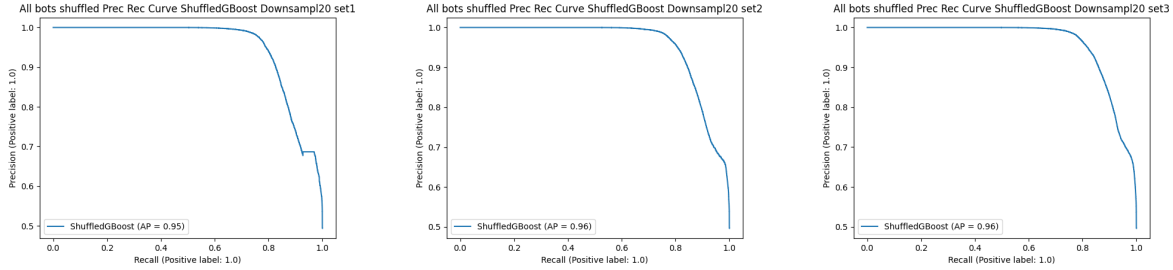
A

Precision Recall Curve

A

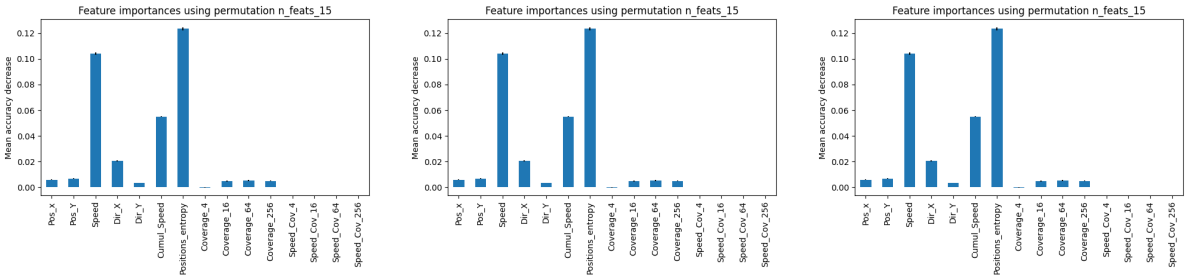
Feature Permutation Results

A



(a) Feature set 1. (b) Feature set 2. (c) Feature set 3.

Figure 6.43: Precision recall curves of the gradient boosting model performances on the RMFS task.



(a) Feature set 1. (b) Feature set 2. (c) Feature set 3.

Figure 6.44: Feature permutation results on the RMFS task.

7 | Conclusions and Future Works

7.1. Results

7.2. Future Works

- Analyze model performances on different kind of faults: is it able to detect a short fault at the end of the task? how long does it take to detect a fault?
- Time series exploitation
- Single feature importance on different faults and different tasks (i.e. stretch entropy calculation more back in timeseries and see different fault performances)
- Feature importance on task completion identification
- When two features are correlated and one of the features is permuted, the model will still have access to the feature through its correlated feature. This will result in a lower importance value for both features, where they might actually be important. One way to handle this is to cluster features that are correlated and only keep one feature from each cluster. This strategy is explored in the following example: Permutation Importance with Multicollinear or Correlated Features

Bibliography

[1] D. Albani, J. IJsselmuiden, R. Haken, and V. Trianni. Monitoring and mapping with robot swarms for agricultural applications. In *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6. IEEE, 2017.

[2] M. H. Amoozgar, A. Chamseddine, and Y. Zhang. Experimental Test of a Two-Stage Kalman Filter for Actuator Fault Detection and Diagnosis of an Unmanned Quadrotor Helicopter. *Journal of Intelligent & Robotic Systems*, 70(1):107–117, 2013. ISSN 1573-0409. doi: 10.1007/s10846-012-9757-7. URL `https://doi.org/10.1007/s10846-012-9757-7`.

[3] D. Azzalini, L. Bonali, and F. Amigoni. A minimally supervised approach based on variational autoencoders for anomaly detection in autonomous robots. *IEEE Robotics and Automation Letters*, 6(2):2985–2992, 2021. doi: 10.1109/LRA.2021.3062597.

[4] K. Bader, B. Lussier, and W. Schön. A fault tolerant architecture for data fusion: A real application of kalman filters for mobile robot localization. *Robotics and Autonomous Systems*, 88:11–23, 2017. ISSN 0921-8890. doi: https://doi.org/10.1016/j.robot.2016.11.015. URL `https://www.sciencedirect.com/science/article/pii/S0921889015302943`.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

[6] J. D. Bjerknes and A. F. T. Winfield. On fault tolerance and scalability of swarm robotic systems. In A. Martinoli, F. Mondada, N. Correll, G. Mermoud, M. Egerstedt, M. A. Hsieh, L. E. Parker, and K. Støy, editors, *Distributed Autonomous Robotic Systems: The 10th International Symposium*, pages 431–444. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. doi: 10.1007/978-3-642-32723-0_31. URL `https://doi.org/10.1007/978-3-642-32723-0_31`.

[7] A. Blázquez-García, A. Conde, U. Mori, and J. A. Lozano. A review on out-

lier/anomaly detection in time series data. *CoRR*, abs/2002.04236, 2020. URL `https://arxiv.org/abs/2002.04236`.

[8] M. Bonani, V. Longchamp, S. Magnenat, P. Rétornaz, D. Burnier, G. Roulet, F. Vaussard, H. Bleuler, and F. Mondada. The marxbot, a miniature mobile robot opening new perspectives for the collective-robotic research. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4187–4193. IEEE, 2010.

[9] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[10] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[11] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[12] A. L. Christensen, R. OGrady, and M. Dorigo. From fireflies to fault-tolerant swarms of robots. *IEEE Transactions on Evolutionary Computation*, 13(4):754–766, 2009. doi: 10.1109/TEVC.2009.2017516.

[13] M. Dorigo. Swarmanoid, 2006. URL `https://www.swarmanoid.org/index.php.html`.

[14] J. J. Enright and P. R. Wurman. Optimization and coordinated autonomy in mobile fulfillment systems. In *Proceedings of the 9th AAAI Conference on Automated Action Planning for Autonomous Mobile Robots*, AAAIWS'11-09, page 33–38. AAAI Press, 2011.

[15] A. Fernández, S. García, M. Galar, R. C. Prati, B. Krawczyk, and F. Herrera. *Learning from imbalanced data sets*, volume 10. Springer, 2018.

[16] E. Ferrante, A. Turgut, C. Huepe, A. Stranieri, C. Pinciroli, and M. Dorigo. Self-organized flocking with a mobile robot swarm: a novel motion control method. *Adaptive Behavior*, 20:460–477, 12 2012. doi: 10.1177/1059712312462248.

[17] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1): 119–139, 1997. ISSN 0022-0000. doi: https://doi.org/10.1006/jcss.1997.1504. URL `https://www.sciencedirect.com/science/article/pii/S002200009791504X`.

[18] E. Guizzo. Kiva systems: three engineers, hundreds of robots, one warehouse, 2010. URL `https://spectrum.ieee.org/three-engineers-hundreds-of-robots-one-warehouse`.

[19] T. Hancock, T. Jiang, M. Li, and J. Tromp. Lower bounds on learning decision lists and trees. *Information and Computation*, 126(2):114–122, 1996. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.1996.0040. URL `https://www.sciencedirect.com/science/article/pii/S0890540196900401`.

[20] F. Harrou, B. Khaldi, Y. Sun, and F. Cherif. Monitoring robotic swarm systems under noisy conditions using an effective fault detection strategy. *IEEE Sensors Journal*, 19(3):1141–1152, 2019. doi: 10.1109/JSEN.2018.2877183.

[21] F. Harrou, B. Khaldi, Y. Sun, and F. Cherif. An efficient statistical strategy to monitor a robot swarm. *IEEE Sensors Journal*, 20(4):2214–2223, 2020. doi: 10.1109/JSEN.2019.2950695.

[22] S. Hashem and B. Schmeiser. Improving model accuracy using optimal linear combinations of trained neural networks. *IEEE Transactions on Neural Networks*, 6(3):792–794, 1995. doi: 10.1109/72.377990.

[23] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009. URL `http://www-stat.stanford.edu/~tibs/ElemStatLearn/`.

[24] R. Isermann and P. Ballé. Trends in the application of model-based fault detection and diagnosis of technical processes. *Control Engineering Practice*, 5(5):709–719, 1997. ISSN 0967-0661. doi: https://doi.org/10.1016/S0967-0661(97)00053-1. URL `https://www.sciencedirect.com/science/article/pii/S0967066197000531`.

[25] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL `http://www.scipy.org/`.

[26] E. Khalastchi and M. Kalech. On fault detection and diagnosis in robotic systems. *ACM Comput. Surv.*, 51(1), Jan. 2018. ISSN 0360-0300. doi: 10.1145/3146389. URL `https://doi.org/10.1145/3146389`.

[27] E. Khalastchi and M. Kalech. Fault detection and diagnosis in multi-robot systems: A survey. *Sensors (Switzerland)*, 19, 2019. ISSN 14248220. doi: 10.3390/s19184019.

[28] E. Khalastchi, M. Kalech, G. A. Kaminka, and R. Lin. Online data-driven anomaly detection in autonomous robots. *Knowledge and Information Systems*, 43(3):657–

688, 2015. ISSN 0219-3116. doi: 10.1007/s10115-014-0754-y. URL https://doi.org/10.1007/s10115-014-0754-y.

[29] B. Khaldi, F. Harrou, F. Cherif, and Y. Sun. Monitoring a robot swarm using a data-driven fault detection approach. *Robotics and Autonomous Systems*, 97:193–203, 2017. ISSN 0921-8890. doi: https://doi.org/10.1016/j.robot.2017.06.002. URL https://www.sciencedirect.com/science/article/pii/S0921889017300854.

[30] L. Kunze, N. Hawes, T. Duckett, M. Hanheide, and T. Krajník. Artificial intelligence for long-term robot autonomy: A survey. *IEEE Robotics and Automation Letters*, 3 (4):4023–4030, 2018. doi: 10.1109/LRA.2018.2860628.

[31] X. Li and L. E. Parker. Sensor analysis for fault detection in tightly-coupled multi-robot team tasks. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3269–3276, 2007. doi: 10.1109/ROBOT.2007.363977.

[32] W.-Y. Loh. Classification and regression trees. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 1(1):14–23, 2011.

[33] Massachusetts Institute of Technology. iRobot Create 2, 2014. URL https://www.irobot.com/about-irobot/stem/create-2.aspx.

[34] M. Merschformann, L. Xie, and H. Li. Rawsim-o: A simulation framework for robotic mobile fulfillment systems. *Logistics Research*, 11, 08 2018. doi: 10.23773/2018_8.

[35] R. Micalizio, P. Torasso, and G. Torta. On-line monitoring and diagnosis of multi-agent systems: A model based approach. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, page 848–852, NLD, 2004. IOS Press. ISBN 9781586034528.

[36] MIT Senseable City Lab. Seaswarm, 2010. URL http://senseable.mit.edu/seaswarm/index.html.

[37] J. Nembrini. Minimalist coherent swarming of wireless networked autonomous mobile robots. Technical report, University of the West of England, 2005.

[38] S. Y. Nof. *Handbook of industrial robotics*. John Wiley & Sons, 1999.

[39] C. Pinciroli. Argos examples, 2015. URL https://github.com/ilpincy/argos3-examples/blob/master/controllers/footbot_foraging/footbot_foraging.cpp.

[40] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and

M. Dorigo. ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence*, 6(4):271–295, 2012.

[41] R. Rojas et al. Adaboost and the super bowl of classifiers a tutorial introduction to adaptive boosting. *Freie University, Berlin, Tech. Rep*, 2009.

[42] L. Rokach and O. Maimon. Top-down induction of decision trees classifiers—a survey. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 35:476 – 487, 12 2005. doi: 10.1109/TSMCC.2004.843247.

[43] E. Şahin. Swarm robotics: From sources of inspiration to domains of application. In E. Şahin and W. M. Spears, editors, *Swarm Robotics*, pages 10–20, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-30552-1.

[44] J.-H. Shin and J.-J. Lee. Fault detection and robust fault recovery control for robot manipulators with actuator failures. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, volume 2, pages 861–866 vol.2, 1999. doi: 10.1109/ROBOT.1999.772398.

[45] E. Skoundrianos and S. Tzafestas. Finding fault - fault diagnosis on the wheels of a mobile robot using local model neural networks. *IEEE Robotics Automation Magazine*, 11(3):83–90, 2004. doi: 10.1109/MRA.2004.1337829.

[46] D. Tarapore, A. L. Christensen, and J. Timmis. Generic, scalable and decentralized fault detection for robot swarms. *PLOS ONE*, 12(8):1–29, 08 2017. doi: 10.1371/journal.pone.0182058. URL `https://doi.org/10.1371/journal.pone.0182058`.

[47] M. ten Hompel, H. Bayhan, J. Behling, L. Benkenstein, J. Emmerich, G. Follert, M. Grzenia, C. Hammermeister, H. Hasse, D. Hoening, C. Hoppe, S. Kerner, P. Klokowski, B. Korth, G. Kuhlmann, J. Leveling, D. Lünsch, R. Mendel, F. Menebröker, A. Nettsträter, J. Pieperbeck, C. Prasse, S. Roeder, M. Roidl, M. Rotgeri, D. Sparer, S. Walter, and O. Wolf. Technical report: Loadrunner®, a new platform approach on collaborative logistics services. *Logistics Journal : nicht referierte Veröffentlichungen*, 2020(10), 2020. ISSN 1860-5923. doi: 10.2195/lj_NotRev_tenhompel_en_202010_01. URL `https://www.logistics-journal.de/not-reviewed/2020/10/5114`.

[48] M. Van and H.-J. Kang. Robust fault-tolerant control for uncertain robot manipulators based on adaptive quasi-continuous high-order sliding mode and neural network. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 229(8):1425–1446, 2015. doi: 10.1177/0954406214544311. URL `https://doi.org/10.1177/0954406214544311`.

[49] V. Verma, G. Gordon, R. Simmons, and S. Thrun. Real-time fault diagnosis [robot fault diagnosis]. *IEEE Robotics Automation Magazine*, 11(2):56–66, 2004. doi: 10.1109/MRA.2004.1310942.

[50] H. Wei, J. Timmis, and R. Alexander. Evolving test environments to identify faults in swarm robotics algorithms. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 929–935, 2017. doi: 10.1109/CEC.2017.7969408.

[51] A. F. Winfield. *Foraging Robots*, pages 3682–3700. Springer New York, New York, NY, 2009. ISBN 978-0-387-30440-3. doi: 10.1007/978-0-387-30440-3_217. URL `https://doi.org/10.1007/978-0-387-30440-3_217`.

[52] A. F. Winfield and J. Nembrini. Safety in numbers: Fault-tolerance in robot swarms. *International Journal of Modelling, Identification and Control*, 1(1):30–37, 2006. ISSN 17466180. doi: 10.1504/IJMIC.2006.008645.

[53] H. Zantema and H. Bodlaender. Finding small equivalent decision trees is hard. *International Journal of Foundations of Computer Science*, 11(2):343–354, 2000. ISSN 0129-0541. doi: 10.1142/S0129054100000193.

List of Figures

2.1 Emergent taxis behavior illustration. 8
2.2 Univariate and multivariate timseries examples. 10
2.3 Decision tree example. 12

4.1 Snapshot of ARGoS graphical user interface. 22
4.2 Snapshot of RAWSim-O graphical user interface. 22
4.3 Photo of Foot-bot. 23
4.4 RAWSim-O bots. 24
4.5 Snapshot of flocking task execution. 25
4.6 Foraging task finite state machine 28
4.7 The central process of a RMFS. 29
4.8 Overview of the simulation process in a RMFS. 30

5.1 Flocking task robots starting positions. 34
5.2 ARGoS flocking plotted trajectories. 35
5.3 Foraging arena example. 36
5.4 ARGoS foraging plotted trajectories. 37
5.5 RMFS warehouse example. 37
5.6 RAWSim-O plotted trajectories. 38

6.1 Flocking speed feature values. 46
6.2 Flocking cumulative speed feature values. 46
6.3 Flocking neighbors number feature values. 47
6.4 Flocking neighbors average distance feature values. 48
6.5 Flocking centroid distance feature values. 48
6.6 Flocking centroid distance feature values. 49
6.7 Flocking position entropy feature values. 49
6.8 Flocking nominal agents area coverage feature values. 50
6.9 Flocking faulty agents area coverage feature values. 50
6.10 Flocking nominal agents area coverage speed feature values. 51
6.11 Flocking faulty agents area coverage speed feature values. 51

6.12 Swarm trajectory. 52
6.13 Swarm speed. 52
6.14 Flocking swarm area coverage feature values. 53
6.15 Foraging speed feature values. 54
6.16 Foraging cumulative speed feature values. 54
6.17 Foraging neighbors number feature values. 55
6.18 Foraging neighbors average distance feature values. 56
6.19 Foraging centroid distance feature values. 56
6.20 Foraging centroid distance feature values. 57
6.21 Foraging position entropy feature values. 57
6.22 Foraging nominal agents area coverage feature values. 58
6.23 Foraging faulty agents area coverage feature values. 58
6.24 Flocking nominal agents area coverage speed feature values. 58
6.25 Foraging faulty agents area coverage speed feature values. 59
6.26 Swarm trajectory. 59
6.27 Foraging swarm area coverage feature values. 61
6.28 RMFS speed feature values. 62
6.29 RMFS cumulative speed feature values. 62
6.30 RMFS neighbors number feature values. 63
6.31 RMFS neighbors average distance feature values. 64
6.32 RMFS centroid distance feature values. 64
6.33 RMFS position entropy feature values. 65
6.34 RMFS nominal agents area coverage feature values. 65
6.35 RMFS faulty agents area coverage feature values. 66
6.36 RMFS Swarm trajectory. 66
6.37 Confusion matrices of the gradient boosting model performances on the flocking task. 70
6.38 Precision recall curves of the gradient boosting model performances on the flocking task. 71
6.39 Feature permutation results on the flocking task. 72
6.40 Confusion matrices of the gradient boosting model performances on the foraging task. 72
6.41 Precision recall curves of the gradient boosting model performances on the foraging task. 73
6.42 Confusion matrices of the gradient boosting model performances on the RMFS task. 73

6.43 Precision recall curves of the gradient boosting model performances on the RMFS task. 74
6.44 Feature permutation results on the RMFS task. 74

List of Tables

2.1 Internal hazards for a single bot. 7

5.1 Comma separated values file structure. 38
5.2 Single bot list of features. 39

6.1 Confusion matrix structure. 68

Acknowledgements

I want to thank: Professor Francesco Amigoni and Ing. Davide Azzalini for their help and contribution to the development of this thesis.
Carlo Pinciroli and Marius Merschformann for their help on how to use their simulators.
The open-source community without whom this work would have never been possible.
My family that has always supported me throughout my studies.
My friends that are and have been moral support during the years of my academic life.