DESIGN AND IMPLEMENTATION OF A BEHAVIOR RECOGNITION AND CLONING SYSTEM BASED ON LEARNING FROM OBSERVATION

(DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE RECONOCIMIENTO Y CLONACIÓN DE COMPORTAMIENTOS BASADO EN APRENDIZAJE POR OBSERVACIÓN)

Trabajo de Fin de Grado
para acceder al
GRADO EN INGENIERÍA INFORMÁTICA

Autor: Carlos Ortiz Sobremazas

Director: Jose Luis Montaña Arnaiz

Co-Director: Cristina Tîrnăucă

Septiembre - 2015
## Contents

1 Introduction ............................................ 6  
   1.1 Historical Perspective of Learning from Observation .................. 6  
   1.2 Potential Applications of Learning from Observation .................. 7  

2 Methodology ............................................. 9  
   2.1 Machine Learning Tools ..................................... 10  

3 Behavior Recognition and Behavior Cloning from Observation ........... 14  
   3.1 Training a Probabilistic Finite Automaton ............................. 14  
   3.2 Evaluation Metrics ............................................ 15  
      3.2.1 Evaluation Metrics for Behavior Recognition ................... 15  
      3.2.2 Evaluation Metrics for Behavior Cloning ...................... 15  

4 Experiments .............................................. 17  
   4.1 Training Maps .............................................. 17  
   4.2 Agent Strategies ........................................... 18  
   4.3 Traces and Performance Evaluation .................................. 19  
   4.4 Behavior Recognition Experimentation ................................ 19  
   4.5 Behavior Cloning Experimentation .................................. 21  
      4.5.1 Predictive Accuracy. ...................................... 21  
      4.5.2 Monte Carlo Distance ..................................... 23  

5 Implementation .......................................... 24  
   5.1 JAVA Programming Language .................................... 24  
   5.2 Python Programming Language ................................... 27  
   5.3 Knime Analytics Platform ....................................... 27  

6 Concluding Remarks and Future Work ................................... 29
Figures

1. Training examples for LfO ............................................. 10
2. Example of a Probabilistic Neuronal Network ....................... 12
3. Example of a Multilayer Feedforward Neural Network ............. 13
4. Training Maps .......................................................... 17
5. Distance Matrix for Testing Group 1 ................................ 20
6. Confusion Matrix for Testing Group 2 ............................... 21
7. Predictive Accuracy for the generated maps ........................ 22
8. Monte Carlo distance between original and cloned behavior .... 23
9. Class Diagram ............................................................ 26
10. Knime Model ............................................................. 28
Abstract

Imagine an agent that performs tasks according to different planned strategies. Behavior Recognition aims to identify which of the available strategies are carried out by the agent by simply observing the agent’s actions and the environmental conditions during a certain period of time. The objective of Behavior Cloning is a bit more ambitious, since the learner must be able to act-alike the agent. In both problems, the only assumption is that the learner has access to a training set that contains, for each strategy, several labeled traces of observations. The goal of this project is to implement a simulated learning environment (in this case a Roomba vacuum cleaner robot), and to find appropriate models and machine learning tools that allow, on one hand, a correct identification of the robot’s strategy, and on the other hand, a reasonable imitation of the robot’s behavior.

Key Words

Learning from Observation, Behavior Recognition, Behavior Cloning, Probabilistic Finite Automata, Classification.
Resumen

Imaginemos un agente que realiza tareas siguiendo distintos tipos de estrategias. El reconocimiento del comportamiento tiene como finalidad descubrir cuál es la estrategia que el agente está realizando simplemente mediante la observación de las acciones que ejecuta y las condiciones ambientales en un determinado instante. El propósito de la clonación de conductas es más ambicioso, ya que el sistema debe ser capaz de actuar igual que el agente bajo la única hipótesis es que el sistema de aprendizaje tiene acceso a un conjunto de datos que contiene, para cada estrategia, una serie de trazas etiquetadas de observaciones. El objetivo del proyecto es implementar un entorno simulado de aprendizaje (en nuestro caso es el robot aspirador Roomba), y encontrar el modelo apropiado y las herramientas de aprendizaje automático que pueden permitirnos, por una parte, la correcta identificación de la estrategia del robot y, por otra parte, una adecuada imitación del comportamiento del robot.

Palabras Clave

Aprendizaje por Observación, Reconocimiento del comportamiento, Clonación de conducta, Autómata Finito Probabilístico, Clasificación.
Acknowledgments

First of all, thanks to my family: because of them and of their efforts I have been able to get here.

Thanks to my friends for encouraging me to follow at all times throughout the university.

Thanks to my classmates also for their help over these four years.

Thanks to José Luis for helping me and giving me the opportunity to realize this project.

Finally, thanks to Cristina for all her effort, patience and dedication over this year.
1 Introduction

Modern training, entertainment and education applications make use of autonomously controlled virtual agents or physical robots. In these applications, the agents must display complex intelligent behaviors that until recently were only shown by humans. Driving simulations, for example, require having vehicles moving in a realistic way in the simulation, while interacting with other virtual agents as well as humans. Likewise, computer games require artificial characters or opponents that display complex intelligent behaviors to enhance the entertainment factor of the games.

Creating those complex behaviors is usually difficult, since it requires significant and costly resources and because much of the knowledge is tacit in nature. In fact, humans that are proficient in the task have difficulties articulating the knowledge in an effective manner for inclusion in the agent’s behavior model. For example, when asked how hard to apply the brakes in a speeding car while approaching a traffic light, most human practitioners would be unable to find appropriate words to describe the experience. On the other hand, showing how to do it is a lot easier.

An attractive and promising alternative to handcrafting behaviors is to automatically generate them through machine learning techniques. The problem of automatically generating behaviors has been studied in artificial intelligence (AI) from two different perspectives:

1. Reinforcement learning (RL) focuses on learning from experimentation.
2. Learning from Observation (LfO) focuses on learning by observing sample traces of correct behavior.

Reinforcement Learning is, by far, the better studied of the two, but presents several open problems like scalability and generalization. Furthermore, only behaviors that are not human-like can be created through RL. We believe that Learning from Observation can offer a more promising and computationally tractable approach that delivers human-like behaviors quickly, with high fidelity, at an affordable cost, and with a reasonable level of generalization. LfO has already shown some success in learning implicit knowledge. Nevertheless, LfO requires further research to realize its full and extensive potential.

1.1 Historical Perspective of Learning from Observation

One of the pioneers of this field is M. Bauer [1], who shows how to make use of knowledge about variables, inputs, instructions and procedures in order to learn programs, which basically amounts to learning strategies to perform abstract computations by demonstration, a topic that was especially popular in robotics [9]. Another early mention of LfO
comes from Michalski et al. [10], who define it merely as unsupervised learning. González et al [6] discussed LfO at length, but provided no formalization nor suggested an approach to realize it algorithmically. More recent work on the more general LfO subject came nearly simultaneously but independently from Sammut et al [19] and Sidani [21]. Fernlund et al. [4] used LfO to build agents capable of driving a simulated automobile in a city environment. Pomerleau [16] developed the ALVINN system that trained neural networks from observation of a road-following automobile in the real world. Moriarty and Gonzalez [12] used neural networks to carry out LfO for computer games. König and Laird [8] introduced LfO in complex domains with the SOAR system, by using inductive logic programming techniques. Other significant work done under the label of learning from demonstration has emerged recently in the Case-Based Reasoning community. Floyd et al. [5] present an approach to learn how to play RoboSoccer by observing the play of other teams. Ontañón et al. [14] use learning from demonstration for real-time strategy games in the context of Case-Based Planning. Finally, another related area is that of Inverse Reinforcement Learning [13], where the focus is on reconstructing the reward function given optimal behavior (i.e., given a policy, or a set of trajectories). One of the main problems here is that different reward functions may correspond to the observed behavior, and heuristics need to be devised to only consider families of reward functions that are interesting.

1.2 Potential Applications of Learning from Observation

Some application domains, like complex computer games or robotics, require the creation of artificial agents that show different very complex behaviors, and whose programming cost is usually quite high. Moreover, programming the artificial intelligence behind the game can also be time consuming. For example, the artificial intelligence for the two virtual characters in the game Façade took more than five person-years to develop. Thus, if the creation of artificial life is difficult, programming an autonomous robot that adapts to a new environment is an even more complex task. Learning from Observation is an effective way to solve this behavior problem.

A technical form to implement Learning from Observation in an effective, efficient and practical way would represent a tremendous benefit to the above applications in the virtual and in the physical world, for example some simulations or physical robots. Over time, the primary objective of these simulations is to obtain a result as close as possible to the behavior of a human, but getting it is a very complicated task. The main reason that implement human’s behavior is very difficult is because human behavior is highly complex and context-dependent. The United States Department of Defense, or US DoD, was one of the first to recognize the value of intelligent agents that could reliably act as intelligent opponents, friendly forces and neutral bystanders in training simulations. The availability of such agents permitted them avoid using human experts, a scarce and expensive resources, who had been used to manually control such entities in the training sessions in the past.
Semi-Automated Forces (SAF) were the first attempt at developing such software agents, where simulated enemy and friendly entities inhabit the virtual training environment and act to oppose or support war-fighters in training sessions. Recent advances in the closely related area of Computer-Generated Forces (CGF) indicate important progress in simulating behaviors that are more complex, but at the cost of operational efficiency. However, CGF models are still very difficult to build, validate and maintain.

LfO could be applied in other daily-life tasks that require the same behavior by humans, such as making a cup of coffee or running a washing machine, can be implemented by intelligent agents that know the environment around them. In this case, LfO brings intelligence to our environments and make those environments responsive to us. In computing, this electronic environments that are sensitive to the presence of humans are called Ambient Intelligence (AmI). As stated by Philips Research Technologies\(^1\), a future in which “People living comfortably in digital environments in which the electronics are sensitive to their needs, personalized to their requirements, anticipatory to their behavior and responsive to their presence” could become a reality.

Many AmI systems incorporate only low-level type intelligence. Very often, they are built in the absence of Artificial Intelligence (AI), while the emphasis is concentrated on their hardware component: with sensors, communication protocols, and, in general, ubiquitous computing. As an example, many smart cities projects consist primarily in the distribution of thousands of sensors over the urban space. This limited amount of intelligence represents an inconvenient, at least in early stages of an AmI system, but it is rapidly being solved with emerging software Machine Learning (ML) applications that will result in a balanced combination of ubiquitous technologies and AI methodologies. ML has received attention inside the AI community from the beginning. Neural Networks, Inductive Learning from Examples, Case-Based Reasoning, Decision Trees (DTs), Bayesian Networks (BNs), Support Vector Machines and other Data Mining techniques like k-clustering contribute decisively in the whole process of knowledge discovery and representation. Nowadays, ML is widely used, so AmI will likely also need to handle this technology. Moreover, one main requirement for AmI is to learn by observing users. For example, several systems understand user commands, but they are not intelligent enough to avoid doing things that the user does not want to do. Basic ML methods will enable AmI systems to learn by observing users, thus making these systems more adapted to them.

Our goal is devoted to the study of basic ML methods that will enable AmI systems to learn by observing users. Imagine a human who performs a task. In our formulation of the problem of behavioral learning we want to recognize, and more generally, to clone, the way in which the task is performed by the agent just by observing the actions it performs. We think that this point of view is general enough to deal with relevant specific applications in AmI such as the automatic generation of the software needed by intelligent autonomous agents like robots that perform several tasks.

\(^1\)http://www.research.philips.com/technologies/projects/ami/
2 Methodology

In Learning from Observation there is a learner that observes one or several agents performing a task in a given environment, and recording the agent’s behavior in the form of traces. Then, those traces are used to discover some properties of the behavior, for instance the particular task or, for the same task, the kind of strategy the agent is using to perform it. Most LfO work assumes that the agent does not have access to a description of the task during learning, and thus, the features of the task and the way it is achieved must be learned purely by unobtrusive observation of the behavior of the agent. Let $B$ be the behavior of an agent $A$. By behavior, we mean the control mechanism, policy, or algorithm that an agent or a learner uses to determine which actions to execute over time.

Our formalization is founded on the principle that behavior can be modeled as a stochastic process, and its elements as random variables depending on time. Our model includes the following variables: its perception $X$ of the environment taking values in some space $\mathcal{X}$, its unobservable internal state $C \in \mathcal{C}$, and the perceptible actions it executes, $Y \in \mathcal{Y}$ (We use the following convention: if $X_t$ is a variable, then we use a calligraphic $X$ to denote the set of values it can take, and lower case $x \in \mathcal{X}$ to denote specific values it takes). We interpret the agent’s behavior as a discrete-time process $Z = \{Z_1, \ldots, Z_k, \ldots\}$ (which can be either deterministic or stochastic), with state space $\mathcal{I} = \mathcal{X} \times \mathcal{C} \times \mathcal{Y}$; $Z_t = (X_t, C_t, Y_t)$ is a variable that captures the state of the agent at time $t$.

The observed behavior of an agent in a particular execution defines a trace $T$ of observations: $T = [(x_1, y_1), \ldots, (x_m, y_m)]$, where $x_t$ and $y_t$ represent the specific perception of the environment and action of the agent at time $t$. The pair of variables $X_t$ and $Y_t$ represents the observation of the agent $A$, i.e., $O_t = (X_t, Y_t)$. Thus, for simplicity, we can write a trace as $T = [o_1, \ldots, o_m]$. We assume that the random variables $X_t$ and $Y_t$ are multidimensional discrete variables. Under this statistical model, we distinguish three types of behaviors: type 1 (that includes strict imitation behavior) corresponding to a process that eventually depends on time (independent of previous states and actions); type 2 (reactive behavior) where $Y_t$ eventually depends on time $t$, present state $X_t$ and non-observable internal state $C_t$; type 3 (planned behavior) when the action $Y_t$ eventually depends on time $t$, previous states $X_1, \ldots, X_t$ and actions $Y_1, \ldots, Y_{t-1}$, and non-observable internal state $C_t$. Note that we use eventually to include the case in which there is in fact no dependency.

When a behavior does not explicitly depend on time, we say that it is a stationary behavior. Also, we distinguish between deterministic and stochastic behavior.

In this project we model only stationary behaviors that do not explicitly depend on the internal state. We focus on behaviors of type 2 and 3, and we limit the “window” of knowledge in the case of planned behavior to one previous observation (the experiments can be easily extended to allow a more generous “memory”, with the obvious drawback of an increased number of features). This gives rise to three possibilities for the current
action $Y_t$:

- $Y_t$ depends only on current state $X_t$ (Model 1)
- $Y_t$ depends on previous and current state: $X_t, X_{t-1}$ (Model 2)
- $Y_t$ depends on previous and current state: $X_t, X_{t-1}$, and on previous action $Y_{t-1}$ (Model 3)

Note that the actual strategies of the agent may be of more complex types (see Section 4.2), but the cloned strategy is restricted to one of the three models presented above.

An example of the kind of information available for each of the three models is presented in Figure 1.

<table>
<thead>
<tr>
<th>Model 1</th>
<th>Model 2</th>
<th>Model 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>1st</td>
<td>1st</td>
</tr>
<tr>
<td>2nd</td>
<td>2nd</td>
<td>2nd</td>
</tr>
<tr>
<td>Class</td>
<td>Class</td>
<td>Class</td>
</tr>
<tr>
<td>$x_1$</td>
<td>$x_1$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$y_1$</td>
<td>$y_1$</td>
<td>$y_1$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$x_2$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$y_2$</td>
<td>$y_2$</td>
<td>$y_2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$x_m$</td>
<td>$x_m$</td>
<td>$x_m$</td>
</tr>
<tr>
<td>$y_m$</td>
<td>$y_m$</td>
<td>$y_m$</td>
</tr>
</tbody>
</table>

Figure 1: Training examples for LfO

In the next section we describe the kind of learning machines that we propose for modeling reactive and planned behaviors. Note that the only information we have is a trace with pairs (state, action): we do not know if the trace was produced by a deterministic or a stochastic agent, or whether it uses an internal state. But we would like to have a mechanism that predicts, in each state, the action to perform.

### 2.1 Machine Learning Tools

Since one of the objectives of this project is the recognition of a strategy based on a series of observations, we opted to train a probabilistic finite automaton (PFA) using the available information. The role of the PFA is twofold: on one hand, it is a model that allows us to compute the probability of a new previously unseen trace (a feature needed for behavior recognition), and on the other hand, it offers us a good estimation for the conditional probability of a certain action in a given state (a feature that we need for behavior cloning).

**Probabilistic Finite Automata**

Although PFAs have been introduced since the 60s by M. O Rabin (see [17]), they are still used in several fields of science and technology for modeling stochastic processes such as DNA sequencing analysis, image and speech recognition, human activity recognition and
environmental problems among others. A reference covering the basic PFA properties and explaining the relations with other Markovian models is [3].

Formally, a PFA (with finite states) is a 5-tuple \( A = (\Sigma, Q, \phi, \iota, \gamma) \) where \( \Sigma \) is a finite alphabet (that is, a discrete set of symbols), \( Q \) is a finite collection of states, \( \phi : Q \times \Sigma \times Q \rightarrow [0, 1] \) is a function defining the transition probability (i.e., \( \phi(q, a, q') \) is the probability of emission of symbol \( a \) while transitioning to state \( q' \) from state \( q \)), \( \iota : Q \rightarrow [0, 1] \) is the initial probability function and \( \gamma : Q \rightarrow [0, 1] \) is the final probability function. In addition, the following functions, defined over words \( \alpha = (a_1, \ldots, a_m) \in \Sigma^* \) and state paths \( \pi = (q_1, \ldots, q_m) \in Q^* \), must be probability distributions (Equation (1) when using final probabilities and Equation (2) otherwise):

\[
P_A(\alpha, \pi) = \iota(q_1) \left( \prod_{i=1}^{m-1} \phi(q_i, a_i, q_{i+1}) \right) \gamma(q_m)
\]

(1)

\[
\hat{P}_A(\alpha, \pi) = \iota(q_1) \prod_{i=1}^{m-1} \phi(q_i, a_i, q_{i+1})
\]

(2)

This implies in particular that the two following functions are probability distributions over \( \Sigma^* \):

\[
P_A(\alpha) = \sum_{q, q'} \iota(q) \phi(q, \alpha, q') \gamma(q')
\]

(3)

\[
\hat{P}_A(\alpha) = \sum_{q, q'} \iota(q) \phi(q, \alpha, q')
\]

(4)

Here \( \phi(q, \alpha, q') \) is the extension of function \( \phi \) to words with the obvious meaning: the probability of reaching state \( q' \) from state \( q \) while generating word \( \alpha \), \( P_A \) is the probability of generating word \( \alpha \) and \( \hat{P}_A \) is the probability of generating a word with prefix \( \alpha \) (see [3] for a detailed explanation of Equations (3) and (4)). In many real situations we are interested in PAs with no final probabilities, and in this case we use Equation (4).

Note that the conditional probability of a symbol \( a \) given a state \( q \) can be defined in terms of the transition probability \( \phi \):

\[
\mathbb{P}(a \mid q) = \sum_{q' \in Q} \phi(q, a, q')
\]

(5)

Thus, PFAs can be used in BC as a stochastic model: when the agent is in a state \( q \), the action to perform can be obtained by sampling according to the probability distribution \( \{\mathbb{P}(a \mid q)\}_{a \in \Sigma} \).
Next, we describe the other ML tools we used for BC, all of them having something in common: based on a set of labeled examples, these tools are able to predict (with higher or lower accuracy) the label of previously unseen examples. These tools are commonly known as classifiers.

**Classifiers**

All classifiers described below use one, two or three features, depending on which of the three models is used (see Figure 1), and have as the class label the type of action undertaken by the agent.

- **Decision Tree**: A decision tree (DT) is a classifier that uses a branching method to illustrate every possible outcome of a decision. DTs can be drawn by hand or created with a graphics program or specialized software. A DT has some entries which can be an object or a situation described by a set of attributes and returns an answer which is a decision that is taken from entries. The values that can take the inputs and outputs can be discrete or continuous values; discrete values are more used because of its simplicity (see [20] for more details about the actual algorithm that we use).

- **K-Nearest Neighbor (KNN)**: KNN is a simple algorithm that stores all available cases and classifies new cases based on a similarity measure. A case is classified by a majority vote of its neighbors (see [7]).

- **Naive Bayes**: A Naive Bayes classifier (NB) is a simple probabilistic classifier based on applying Bayes’ theorem (from Bayesian statistics) with strong (naive) independence assumptions (see [7]).

- **Probabilistic Neuronal Network** [2]: A probabilistic neural network (PNN) is predominantly a classifier that maps any input pattern to a number of classifications. PNN can be forced into a more general function approximator.

  ![Figure 2: Example of a Probabilistic Neuronal Network.](image)

- **Multilayer Feedforward Neural Network** [18]: A multilayer feedforward neural net-
work (RProp) is an interconnection of nodes in which data and calculations flow in a single direction, from the input data to the outputs. The number of layers in a neural network is the number of layers of perceptrons.

Figure 3: Example of a Multilayer Feedforward Neural Network.
3 Behavior Recognition and Behavior Cloning from Observation

Having formally defined the tools employed in this project, we proceed by presenting the way we used them to achieve our goals.

3.1 Training a Probabilistic Finite Automaton

We propose to train a PFA $A = (\Sigma, Q, \phi, \iota, \gamma)$ to model an unknown behavior by observing its trace $T$. To this end, we define the alphabet $\Sigma$ of automaton $A$ to be the set of all actions $\mathcal{Y}$ that the agent can perform. The state space $Q$ depends on the model: it is either $\mathcal{X}$ (Model 1), or $\mathcal{X} \times \mathcal{X}$ (Model 2) or $\mathcal{X} \times \mathcal{Y} \times \mathcal{X}$ (Model 3).

Training the automaton $A$ from a trace $T = [(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)]$ consists of determining the initial probability values $\{\iota(q) \mid q \in Q\}$ and the transition probability function values $\{\phi(q, a, q') \mid q, q' \in Q, a \in \Sigma\}$ (we opted for a model with no final probabilities). For any state $q \in Q$, let $\sharp q$ be the number of occurrences of symbol $q$ in trace $T$. Note that in the case of Models 2 and 3, $\sharp q$ is either $X \times Y \times X$ or $X \times Y \times X \times Y \times X$ (Model 3).

Model 1: $q = x, a = y, q' = x'$

$\sharp(q, a) = |\{i \in 1, m \mid x_i = x, y_i = y\}|$

$\sharp(q, a, q') = |\{i \in 1, m - 1 \mid x_i = x, y_i = y, x_{i+1} = x'\}|$

Model 2: $q = (x'', x), a = y, q' = (x, x')$

$\sharp(q, a) = |\{i \in 2, m \mid x_{i-1} = x'', x_i = x, y_i = y\}|$

$\sharp(q, a, q') = |\{i \in 2, m - 1 \mid x_{i-1} = x'', x_i = x, x_{i+1} = x', y_i = y\}|$

Model 3: $q = (x'', y'', x), a = y, q' = (x, y, x')$

$\sharp(q, a) = |\{i \in 2, m \mid x_{i-1} = x'', x_i = x, y_{i-1} = y'', y_i = y\}|$

$\sharp(q, a, q') = |\{i \in 2, m - 1 \mid x_{i-1} = x'', x_i = x, x_{i+1} = x', y_{i-1} = y'', y_i = y\}|$

Note that in the case of Models 2 and 3, $\sharp(q, a, q')$ is zero by definition if the last element of state $q$ is different than the first element of state $q'$. Next, we estimate the values of $\iota$ and $\phi$ with the following formulas (we use Laplace smoothing to avoid zero values in the testing phase for elements that never appeared in training).

$$\iota(q) := \frac{\sharp q + 1}{m + |Q|} \quad \phi(q, a, q') := \frac{\sharp(q, a, q') + 1}{\sharp q + |Q| \times |\Sigma|}$$
It is easy to see that $P(a | q)$ becomes $(q(a) + |Q|)/(|q + |Q| \times |\Sigma|)$, and this is precisely the probability of performing the action $a$ when in state $q$.

3.2 Evaluation Metrics

In order to evaluate our system, we used the evaluation metrics described below.

3.2.1 Evaluation Metrics for Behavior Recognition

In the case of Behavior Recognition, the goal is to identify which was the strategy used by the agent, given a learning trace $T = [(x_1, y_1), \ldots, (x_m, y_m)]$. To this end, we train a PFA $A_k$ for each available planned strategy, we compute the value $P_{A_k}(\alpha, \pi^M)$ and return $i := \text{argmax}_k \{P_{A_k}(\alpha, \pi^M)\}$, where $\alpha = (y_1, \ldots, y_m)$ and $\pi^M = (q_1^M, \ldots, q_m^M)$. The value of $q_i^M$ depends on the amount $M$ of memory used: $q_i^1 = x_i$ for Model 1, $q_i^2 = (x_i, x_{i-1})$ for Model 2 and $q_i^3 = (x_i, x_{i-1}, y_{i-1})$ for Model 3.

Next we measure the distance between an unknown behavior $B$ exhibited by agent $A$ and a given behavior $B'$ modeled by automaton $A_k$ by computing the negative log-probability

$$R_{A_k}^M(T) := -\frac{1}{m} \log P_{A_k}(\alpha, \pi^M),$$

where $T$ is the trace corresponding to behavior $B$ and $m$ is the number of observations in $T$. This value can be interpreted as a Monte Carlo approximation of the crossed entropy between behaviors $B$ and $B'$, known in the literature as Vapnik’s risk (see [15]).

3.2.2 Evaluation Metrics for Behavior Cloning

We propose two methods for evaluating the performance of the cloning algorithms.

**Predictive Accuracy.** This is a standard measure for classification tasks. Let $M$ be the model trained by one of the learning algorithms using the trace obtained by observing an agent $A$ (that follows a certain strategy) on a fixed set of maps. This model can be either deterministic (in which case, there is only one possible action at any point in time) or stochastic (the action is chosen randomly according to some probability distribution).

Now, let $T = [(x_1, y_1), \ldots, (x_m, y_m)]$ be the trace of the agent $A$ on a different previously unseen map. The predictive accuracy $\text{Acc}_{M}(T)$ is measured as follows:

$$\#(M, T, i) = \begin{cases} 1 & \text{if } M([x_{i-1}, y_{i-1}, x_i]) = y_i \\ 0 & \text{otherwise} \end{cases}$$

$$15$$
Acc_\mathcal{M}(T) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(\mathcal{M}, T, i) \tag{8}

where \mathcal{M}([x_{i-1}, y_{i-1}, x_i]) represents the action predicted by the model \mathcal{M} for the state \(x_i\), possibly knowing previous state and action. If \mathcal{M} is a stochastic model, \mathcal{M}([x_{i-1}, y_{i-1}, x_i]) is a random variable over \(\mathcal{Y}\) with the probability distribution \(\{P(a \mid q)\}_{a \in \mathcal{Y}}\), where \(q = x_i\) for Model 1, \(q = (x_i, x_{i-1})\) for Model 2 and \(q = (x_i, x_{i-1}, y_{i-1})\) for Model 3.

Monte Carlo Distance.

To assess the quality with which a model \(\mathcal{M}\) reproduces the behavior of an agent \(A\) we propose a Monte Carlo-like measure based on estimating the cross entropy between the probability distributions associated with both the model \(\mathcal{M}\) and the agent \(A\). We propose to estimate cross entropy from traces, for this reason it is a Monte Carlo estimation. More concretely, let \(T = [o_1, \ldots, o_n]\) be a trace generated\(^2\) according to model \(\mathcal{M}\) on a fixed map (different than the one used in training), and let \(T' = [o'_1, \ldots, o'_m]\) be the trace generated by the agent on the same map. We define the Monte Carlo distance between model \(\mathcal{M}\) and agent \(A\) as follows (estimated through traces \(T\) and \(T'\)):

\[
H(T, T') = -\frac{1}{n} \sum_{i=1}^{n} \log \left[ \frac{1}{m} \sum_{j=1}^{m} \mathbb{I}(o'_j)(a_i) \right] \tag{9}
\]

Here \(\mathbb{I}(o'_j)\) means the indicator function of set \(\{o'_j\}\). Obviously the previous measure in Equation (9) is empirical. It is assumed that for large enough traces it is closed to the true cross entropy between the behavior corresponding to model \(\mathcal{M}\) and the behavior exhibited by agent \(A\).

Using Laplace smoothing, the previous formula becomes:

\[
H(T, T') = -\frac{1}{n} \sum_{i=1}^{n} \log \left[ \frac{\sum_{j=1}^{m} \mathbb{I}(o'_j)(a_i) + 1}{m + |\mathcal{X} \times \mathcal{Y}|} \right] \tag{10}
\]

\(^2\)The model predicts the next action, but the next state is given by the actual configuration of the map; in the case that it is impossible to perform a certain action because of an obstacle, the agent does not change its location.
4 Experiments

We have run our experiments with a simulator of a simplified version of a Roomba, which is a series of autonomous robotic vacuum cleaners sold by iRobot\(^3\). The original Roomba vacuum cleaner uses a set of basic sensors that help it perform tasks. For instance, it is able to change direction whenever it encounters an obstacle. It uses two independently operating wheels that allow 360 turns in place. Additionally, it can adapt to perform other more creative tasks using an embedded computer in conjunction with the Roomba Open Interface.

4.1 Training Maps

The environment in which the agent moves is a 40 x 60 rectangle surrounded by walls, which may contain all sorts of obstacles. For testing, we have randomly generated obstacles on an empty map. In the sequel, we briefly explain the six maps used in the training phase. Each of them is meant to represent a real-life situation, as indicated by their title (the list is by no means exhaustive).

![Training Maps](image)

(a) Empty Map  (b) Messy Room  (c) Office
(d) Passage  (e) Museum  (f) Maze

Figure 4: Training Maps

**Empty Map.** The empty map consists of a big empty room with no obstacles whatsoever.

\(^3\)According to wikipedia, “iRobot Corporation is an American advanced technology company founded in 1990 and incorporated in Delaware in 2000. Roomba was introduced in 2002. As of Feb 2014, over 10 million units have been sold worldwide”.
Messy Room. The messy room simulates an untidy teenager bedroom, with all sorts of obstacles on the floor, and with a narrow entry corridor that makes the access to the room even more challenging for any ‘hostile intruder’.

The Office. The office map simulates a space in which several rooms are connected to each other by small passages. In this case, obstacles are representing big furniture such as office desks or storage cabinets.

The Passage. The highlight of this map is an intricate pathway that leads to a small room. The main room is huge and does not have any obstacle in it.

The Museum. This map is intended to simulate a room from a museum, with the main sculpture in the center, and with several other sculptures on the four sides of the room, separated by small spaces.

The Maze. The most part of this map consists of narrow pathways with the same width as the agent. There is also a little room which is difficult to find.

The six maps are represented in Figure 4.

4.2 Agent Strategies

We have designed a series of strategies with different complexities. When describing a strategy, we must define the behavior of the agent in a certain situation (which defines its state $X_t$) that depends on the configuration of obstacles in its vicinity (prefix RND is used for stochastic strategies).

Walk. The agent always performs the same movement in a given state. As an example, a possible strategy could be to go Right whenever there are no obstacles, and Up whenever there is only one obstacle to the right (stationary deterministic of type 2, it only depends on current state $X_t$).

Rnd_Walk. In this strategy, the next move is picked up randomly from the set of available moves. For example, an agent that has obstacles to the right and to the left can only move Up or Down, but there is no predefined choice between those two (stationary stochastic of type 2, it only depends on current state $X_t$).

Crash. In this strategy the robot should perform the same action as in the previous cell while there exists this possibility. Whenever it encounters a new obstacle in its way, the agent must choose a certain predefined action. Therefore, it needs to have information about its previous action in order to know where to move (stationary deterministic of type 3, it depends on current state $X_t$ and previous action $Y_{t-1}$).

Rnd_Crash. This strategy allows the agent to take a random direction when it crashes with an obstacle. The main difference with the Rnd_Walk is that in Rnd_Crash the agent does not change direction if it does not encounter an obstacle in its way (stationary stochastic of type 3, it depends on current state $X_t$ and previous action $Y_{t-1}$).
ZigZag. It consists of different vertical movements in two possible directions, avoiding the obstacles. It has an internal state which tells the robot if it should advance towards the left or the right side with this vertical movements: it initially goes towards the right side, and once it reaches one of the right corners, the internal state changes so that the robot will start moving toward the left side (stationary deterministic of type 3, it depends on current state $X_t$, previous action $Y_{t-1}$ and internal state $C_t$).

RND_ZigZag. This strategy is similar to the previous one, with the only difference that once it reaches a corner the internal state could either change its value or not, and this is randomly assigned (stationary stochastic of type 3, it depends on current state $X_t$, previous action $Y_{t-1}$ and internal state $C_t$).

4.3 Traces and Performance Evaluation

In our experiments the simulation time is discrete, and at each time step, the agent can take one out of these 5 actions: up, down, left, right and stand still, with their intuitive effect (if the agent tries to move into an obstacle, the effect is equivalent to the stand still action). The control variable $Y$ can take 5 different values: \{Up, Down, Left, Right, Stand\}. The agent perceives the world through the input variable $X$ having 4 different binary components (up, down, left, right), one of them identifying what can the vacuum cleaner see in each direction (obstacle, no obstacle).

We work with an agent represented by a fairly simplified version of a Roomba robot. In our implementation, although it is possible for the agent to start anywhere, the traces we generate are always with the agent starting in the top-left corner of the map. We use the strategies explained in the section 4.2 and the maps described in section 4.1. For each pair map/strategy we have generated a trace of 1500 observations. In the training phase, we used, for each strategy, a single file obtained by concatenating the traces from the above mentioned six maps. Therefore, each machine was trained using 9000 observations coming from different types of maps.

4.4 Behavior Recognition Experimentation

In order to evaluate our behavior recognition system, we performed two experiments. We have randomly generated three maps (Testing Group 1) for the first experiment and 100 maps (Testing Group 2) for the second experiment. Then, we generated a family of traces (each of them containing 1500 observations) for each pair strategy/map: \{$T_n^i\}_{i\in\{1,2,3\},n\in\{1,...,6\}}$ and \{$\bar{T}_n^i\}_{i\in\{1,...,100\},n\in\{1,...,6\}}$ for the first and the second experiment, respectively.

For the first experiment, we computed the log-normalized distance $R_{n,m}^M(T_n^i)$ between the observation trace $T_n^i$ and the automaton $A_m$, as described in Equation (6). The average value $R_{m,n}^M = \sum_{i=1}^{3} R_{n,m}^M(T_n^i)/3$ of these distances is reported in Figure 5, in
the \((m,n)\)-th cell of Table \(M\), with \(M\) in \(\{1,2,3\}\) corresponding to Model 1, Model 2 and Model 3, respectively. Our system classifies the testing task represented by column \(n\) as being generated by the automaton \(A_k\) such that \(k = \text{argmin}_m R_{m,n}^M\) (minimizing distance maximizes trace probability).

Table 1: \(R_{m,n}^M\) values for \(M = 1\)

<table>
<thead>
<tr>
<th>9000/1500 obs</th>
<th>Walk</th>
<th>Rnd_Walk</th>
<th>Crash</th>
<th>Rnd_Crash</th>
<th>ZigZag</th>
<th>Rnd_ZigZag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walk</td>
<td>8.8447</td>
<td>9.0881</td>
<td>9.554</td>
<td>7.2567</td>
<td>8.998</td>
<td>7.9354</td>
</tr>
<tr>
<td>Rnd_Crash</td>
<td>8.782</td>
<td>4.186</td>
<td>3.4454</td>
<td>3.454</td>
<td>4.9792</td>
<td>4.4259</td>
</tr>
<tr>
<td>ZigZag</td>
<td>10.0427</td>
<td>7.1226</td>
<td>11.6125</td>
<td>4.1299</td>
<td>4.0198</td>
<td>4.0415</td>
</tr>
</tbody>
</table>

Table 2: \(R_{m,n}^M\) values for \(M = 2\)

<table>
<thead>
<tr>
<th>9000/1500 obs</th>
<th>Walk</th>
<th>Rnd_Walk</th>
<th>Crash</th>
<th>Rnd_Crash</th>
<th>ZigZag</th>
<th>Rnd_ZigZag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rnd_Walk</td>
<td>10.0088</td>
<td>4.8185</td>
<td>4.9955</td>
<td>5.6885</td>
<td>6.1171</td>
<td>6.8666</td>
</tr>
<tr>
<td>Crash</td>
<td>10.009</td>
<td>8.0111</td>
<td>10.3333</td>
<td>7.0744</td>
<td>6.612</td>
<td>6.303</td>
</tr>
<tr>
<td>Rnd_Crash</td>
<td>10.0088</td>
<td>5.7422</td>
<td>5.1717</td>
<td>5.2526</td>
<td>6.9289</td>
<td>6.5429</td>
</tr>
<tr>
<td>ZigZag</td>
<td>10.0088</td>
<td>8.1036</td>
<td>11.6281</td>
<td>7.9913</td>
<td>5.7136</td>
<td>6.1689</td>
</tr>
<tr>
<td>Rnd_ZigZag</td>
<td>10.0088</td>
<td>8.0421</td>
<td>11.6978</td>
<td>7.8478</td>
<td>5.7096</td>
<td>5.7097</td>
</tr>
</tbody>
</table>

Table 3: \(R_{m,n}^M\) values for \(M = 3\)

<table>
<thead>
<tr>
<th>9000/1500 obs</th>
<th>Walk</th>
<th>Rnd_Walk</th>
<th>Crash</th>
<th>Rnd_Crash</th>
<th>ZigZag</th>
<th>Rnd_ZigZag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash</td>
<td>12.0089</td>
<td>11.3821</td>
<td>11.938</td>
<td>8.6991</td>
<td>8.3253</td>
<td>7.8532</td>
</tr>
<tr>
<td>Rnd_Crash</td>
<td>12.0089</td>
<td>10.7444</td>
<td>6.458</td>
<td>6.5235</td>
<td>8.3757</td>
<td>8.0651</td>
</tr>
<tr>
<td>ZigZag</td>
<td>12.0089</td>
<td>11.2901</td>
<td>11.9402</td>
<td>9.3499</td>
<td>6.7775</td>
<td>7.5367</td>
</tr>
</tbody>
</table>

Figure 5: Distance Matrix for Testing Group 1

The results of this testing group show that the PFA recognition system is able to correctly recognize the three random strategies (Rnd_Walk, Rnd_Crash and Rnd_ZigZag). However, the system fails when recognizing the respective underlying deterministic strategies (Walk, Crash and ZigZag). Also note that the deterministic versions of the random behaviors (Walk, Crash and ZigZag) are not confused one each other but each of them is most of the times classified by the system as its corresponding non deterministic version (Crash is classified as Rnd_Crash, ZigZag as Rnd_ZigZag).

For the second experiment, the numerical value \(C_{m,n}^m\) placed in the \((m,n)\)-th cell of the confusion matrix is the empirical probability of the \(n\)-th task to be classified as the \(m\)-th task, that is, the percentage of the learning traces produced using strategy \(n\) that are recognized as being produced by strategy \(m\).
More precisely,

\[
C_{m,n}^M = \frac{|\{i \in \{1, \ldots, 100\} \mid m = \arg\min_k R_{Ak}^M (T_n^i)\}|}{100}
\]

The diagonal of this matrix reflects the empirical probabilities of right classification and the sum of the other rows different from the diagonal element is the probability of error. We observe that this second experiment confirms the conclusions of the first one.

<table>
<thead>
<tr>
<th>Table 1: (C_{m,n}^M) values for (M = 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9000/1500 obs</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Walk</td>
</tr>
<tr>
<td>RND_Walk</td>
</tr>
<tr>
<td>Crash</td>
</tr>
<tr>
<td>RND_Crash</td>
</tr>
<tr>
<td>ZigZag</td>
</tr>
<tr>
<td>RND_ZigZag</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2: (C_{m,n}^M) values for (M = 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9000/1500 obs</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Det_Walk</td>
</tr>
<tr>
<td>RND_Walk</td>
</tr>
<tr>
<td>Crash</td>
</tr>
<tr>
<td>RND_Crash</td>
</tr>
<tr>
<td>ZigZag</td>
</tr>
<tr>
<td>RND_ZigZag</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3: (C_{m,n}^M) values for (M = 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>9000/1500 obs</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Walk</td>
</tr>
<tr>
<td>RND_Walk</td>
</tr>
<tr>
<td>Crash</td>
</tr>
<tr>
<td>RND_Crash</td>
</tr>
<tr>
<td>ZigZag</td>
</tr>
<tr>
<td>RND_ZigZag</td>
</tr>
</tbody>
</table>

Figure 6: Confusion Matrix for Testing Group 2

4.5 Behavior Cloning Experimentation

As already mentioned, we used a file of 9000 observations for each strategy in order to train our models. For testing, we used again the three maps of Testing Group 1.

4.5.1 Predictive Accuracy.

The numbers in the three tables of Figure 7 represent average values of the predictive accuracy (defined as in Equation (8)) computed for each of the three randomly generated
maps in Testing Group 1. Note that the only stochastic model is the PFA.

<table>
<thead>
<tr>
<th>Model 1</th>
<th>PFA</th>
<th>DT</th>
<th>PNN</th>
<th>KNN</th>
<th>RProp</th>
<th>NB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walk</td>
<td>0.999</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>0.558</td>
<td>0.506</td>
</tr>
<tr>
<td>RND_Walk</td>
<td>0.282</td>
<td>0.318</td>
<td>0.318</td>
<td>0.316</td>
<td>0.298</td>
<td>0.291</td>
</tr>
<tr>
<td>Crash</td>
<td>0.530</td>
<td>0.532</td>
<td>0.532</td>
<td>0.532</td>
<td>0.514</td>
<td>0.295</td>
</tr>
<tr>
<td>RND_Crash</td>
<td>0.424</td>
<td>0.441</td>
<td>0.441</td>
<td>0.441</td>
<td>0.428</td>
<td>0.367</td>
</tr>
<tr>
<td>ZigZag</td>
<td>0.472</td>
<td>0.350</td>
<td>0.350</td>
<td>0.350</td>
<td>0.348</td>
<td>0.474</td>
</tr>
<tr>
<td>RND_ZigZag</td>
<td>0.483</td>
<td>0.494</td>
<td>0.494</td>
<td>0.494</td>
<td>0.470</td>
<td>0.477</td>
</tr>
<tr>
<td>Avg.</td>
<td>0.532</td>
<td>0.523</td>
<td>0.522</td>
<td>0.522</td>
<td>0.436</td>
<td>0.402</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model 2</th>
<th>PFA</th>
<th>DT</th>
<th>PNN</th>
<th>KNN</th>
<th>RProp</th>
<th>NB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walk</td>
<td>0.363</td>
<td>0.502</td>
<td>0.005</td>
<td>0.006</td>
<td>0.502</td>
<td>0.005</td>
</tr>
<tr>
<td>RND_Walk</td>
<td>0.277</td>
<td>0.294</td>
<td>0.290</td>
<td>0.287</td>
<td>0.277</td>
<td>0.277</td>
</tr>
<tr>
<td>Crash</td>
<td>0.488</td>
<td>0.502</td>
<td>0.437</td>
<td>0.488</td>
<td>0.406</td>
<td>0.278</td>
</tr>
<tr>
<td>RND_Crash</td>
<td>0.431</td>
<td>0.395</td>
<td>0.389</td>
<td>0.395</td>
<td>0.361</td>
<td>0.372</td>
</tr>
<tr>
<td>ZigZag</td>
<td>0.558</td>
<td>0.492</td>
<td>0.489</td>
<td>0.484</td>
<td>0.482</td>
<td>0.481</td>
</tr>
<tr>
<td>RND_ZigZag</td>
<td>0.458</td>
<td>0.589</td>
<td>0.574</td>
<td>0.584</td>
<td>0.550</td>
<td>0.477</td>
</tr>
<tr>
<td>Avg.</td>
<td>0.439</td>
<td>0.462</td>
<td>0.364</td>
<td>0.374</td>
<td>0.430</td>
<td>0.315</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model 3</th>
<th>PFA</th>
<th>DT</th>
<th>PNN</th>
<th>KNN</th>
<th>RProp</th>
<th>NB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walk</td>
<td>0.358</td>
<td>0.338</td>
<td>0.338</td>
<td>0.336</td>
<td>0.501</td>
<td>0.337</td>
</tr>
<tr>
<td>RND_Walk</td>
<td>0.282</td>
<td>0.304</td>
<td>0.293</td>
<td>0.287</td>
<td>0.293</td>
<td>0.289</td>
</tr>
<tr>
<td>Crash</td>
<td>0.716</td>
<td>0.701</td>
<td>0.635</td>
<td>0.645</td>
<td>0.626</td>
<td>0.480</td>
</tr>
<tr>
<td>RND_Crash</td>
<td>0.952</td>
<td>0.958</td>
<td>0.914</td>
<td>0.934</td>
<td>0.914</td>
<td>0.929</td>
</tr>
<tr>
<td>ZigZag</td>
<td>0.919</td>
<td>0.961</td>
<td>0.892</td>
<td>0.944</td>
<td>0.896</td>
<td>0.857</td>
</tr>
<tr>
<td>RND_ZigZag</td>
<td>0.909</td>
<td>0.956</td>
<td>0.890</td>
<td>0.944</td>
<td>0.894</td>
<td>0.855</td>
</tr>
<tr>
<td>Avg.</td>
<td>0.689</td>
<td>0.703</td>
<td>0.660</td>
<td>0.682</td>
<td>0.687</td>
<td>0.625</td>
</tr>
</tbody>
</table>

Figure 7: Predictive Accuracy for the generated maps.

Analyzing the results, we can see that our hierarchy of models behaves as expected: type 3 behavior is very well captured by Model 3 (the one that uses information about both previous state and previous action), while type 2 behavior is better explained by Model 1 (in which we only take into account the current state). Note that even though intuitively, the more info we have, the best we can predict, in the case of type 2 behavior, using this extra information can do more harm than good. Another anticipated result that was experimentally confirmed is that Model 1 would be very good in predicting the Walk strategy, because the agent always performs the same action in a given state, and that RND_Walk would be the most unpredictable strategy of all (a notable exception is in the case of Model 2, for which, surprisingly, PNNs and KNNs have even worse accuracy for the Walk strategy).

It is noteworthy the high accuracy rates of Model 3 for RND_Crash, ZigZag and RND_ZigZag, all type 3 strategies. Also, this model gives somewhat lower, but still good enough, prediction rates for the Crash strategy.
If we were to choose a classifier, we would opt for DTs, which gives, with only one exception (Model 3, Walk), best results among deterministic models.

Using a PFA is, most of the times, either the best option, or very close to the best one, especially when predicting the behavior of more complex strategies. Note that PFAs are notably better for describing type 3 behaviors. Also, although intuitively a stochastic model should be better than a deterministic model in describing the behavior of a random process, in our experiments, this is not the case (PFAs outperform DTs mostly when learning deterministic strategies).

### 4.5.2 Monte Carlo Distance.

In order to compare the distance between the agent’s behavior and the behavior of a given learned strategy, we used the same three randomly generated maps of Testing Group 1. For each of the classifiers (DT, PNN, KNN, RProp, NB) and for each of the six predefined strategies (Walk, RND\_Walk, Crash, RND\_Crash, ZigZag, RND\_ZigZag), we generated the trace $T$ produced by an agent whose next action is dictated by the classifier on each of the three randomly generated maps. On the other hand, we generated the trace $T'$ of each of the six predefined strategies on those three maps. In Figure 8 we present the average of $H(T, T')$ for each pair classifier/strategy (see Equation (10)). For stochastic models, the next action, instead of being predefined, is obtained by sampling according to the probability distribution given by the trained model.

<table>
<thead>
<tr>
<th>Model 1</th>
<th>PFA</th>
<th>DT</th>
<th>PNN</th>
<th>KNN</th>
<th>RProp</th>
<th>NB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walk</td>
<td>0.752</td>
<td>0.752</td>
<td>0.752</td>
<td>0.752</td>
<td>7.348</td>
<td>7.348</td>
</tr>
<tr>
<td>RND_Walk</td>
<td>4.562</td>
<td>7.027</td>
<td>7.027</td>
<td>7.027</td>
<td>7.315</td>
<td>7.315</td>
</tr>
<tr>
<td>Crash</td>
<td>1.441</td>
<td>3.877</td>
<td>3.877</td>
<td>3.877</td>
<td>7.348</td>
<td>7.348</td>
</tr>
<tr>
<td>RND_Crash</td>
<td>5.349</td>
<td>6.734</td>
<td>6.734</td>
<td>6.734</td>
<td>7.352</td>
<td>7.352</td>
</tr>
<tr>
<td>ZigZag</td>
<td>1.774</td>
<td>4.863</td>
<td>4.863</td>
<td>4.863</td>
<td>7.322</td>
<td>7.322</td>
</tr>
<tr>
<td>RND_ZigZag</td>
<td>1.796</td>
<td>6.926</td>
<td>6.926</td>
<td>6.926</td>
<td>7.325</td>
<td>7.325</td>
</tr>
<tr>
<td>Avg.</td>
<td>2.612</td>
<td>5.030</td>
<td>5.030</td>
<td>5.030</td>
<td>7.335</td>
<td>7.335</td>
</tr>
</tbody>
</table>

Figure 8: Monte Carlo distance between original and cloned behavior

Analyzing the results, we can see that PFAs are the best in cloning the agent’s behavior for all six strategies, and that the deterministic approach only equals its performance for the Walk strategy. Also note that random behaviors are harder to reproduce than their deterministic counterparts.
5 Implementation

Now we will go through each of the steps followed in the implementation phase of this project, specifying the programming languages and the software used, and explaining the relationship between the JAVA code, the PFA Python files and the KNIME workflow.

5.1 JAVA Programming Language

The decision of choosing JAVA has been made based on several considerations. First, it is an object oriented language. Secondly, it is a language that gives you plenty of flexibility when it comes to upgrading the code, and lastly, it is a cross-platform and an open-source programming language. There are also lots of libraries and packages that allow its use in a comfortable way, with all the information about JAVA on the Internet.

Step One

The first implemented system consists of a map of cells, each of them being defined by its coordinates. The “Cell” class in the beginning was composed of the getter and setter methods for its coordinates.

Another component of this first system is the environment, which consists of a group of cells that compose a map. That map is created in a class named “Map”, structured as a matrix of 60x40 cells. In the “Map” class, we have to set an initial cell to know where the agent begins its route. In addition, it is necessary to set the position of the agent in a determinate moment, so we created the attribute “actual” in the “Map” class.

This gives us an empty map for the agent, in which we needed to add some obstacles. On one hand, we created the walls to prevent the agent from escaping the map with a method call “addWalls”. On the other hand, we created difficulties in the movement of the agent around the map by generating a number of obstacles that will appear on the map with the method “addObstacle”.

After adding the obstacles, the agent has to know where are its surrounding cells, if they are obstacles or not, and to know where to go according to its strategy. We have added a method for each of the situations in which the robot can be found, and which returns a Boolean value indicating whether the agent finds itself in that situation. For example, if we want to know if the agent has an obstacle on its right cell, we have to call the method “isObstacleRight”.

In some cases, it is important to know which was the last movement of the agent, so we added a “Movement” object that reflects it. This attribute is important because most of the strategies need to know the agent’s last movement.
We also created a class named “Strategy” containing all the agent’s predefined strategies. For each strategy, we created a method that tells the agent what to do in any possible situation. In this class we also have a method that chooses which strategy should the agent follow. This method is called “selectStrategy” and it has two parameters:

- **Strategy**: We choose one of the enumerate values from the class “Str_enum”.
- **Movement**: We have to pass the last movement of the agent to this method. The movement is also an enumerate value, in this case from the class “Movement”.

Because of the obstacles present in the maps, we needed to include a new attribute in the “Cell” class that defines the cell as a possible obstacle. As we did for the coordinates, we created a getter and setter method for the Boolean attribute obstacle. In addition, we added another Boolean value that allows us to know if one cell has been visited by the agent (this is useful for strategies that depend on all previously visited cells).

In the “Map” class we created six maps with the same dimension, where the agent can follow any possible strategy. We created the “selectMap” method, where we generate each of the maps by adding obstacles. For choosing the map we want, we have to set a “Map_enum” object as a parameter. The “Map_enum” class is composed by enumerate values, each of them representing one of the maps that we created in the “selectMap” method.

So far, the agent can move around the map following one of the strategies and avoiding obstacles.

**Step Two**

In a second iteration, we decided to add the possibility of generating random maps. We do so by adding a method in “Map” class that consists of given the number of obstacles we want to add, we distribute all of them around the map, obtaining a random map. Each time we use this method we obtain a different map. These random maps will be used to obtain testing sets of the movement of the agent following an strategy.

One utility that was missing in Step One was the ability of exporting information about the agent to a CSV file. According to the models of section 2, we have to export different information about the agent as it is said in Figure 1. We create the “WriteFile” class when we use the package “java.io” that help us with the input and output file system.

In this class we have two methods:

- **“WriteIn”**: This method consists of writing one CSV file per strategy with the movements of the agent. We will write 1500 observations per map, totaling 9000 observations, for all the strategies. These CSV files form the training set.

- **“WriteTesting”**: We write one CSV file per strategy according to the movements of the agent in a random map. This file will be used in Python or Knime as a testing set for a strategy.
After creating these CSV files we used them to train and test our models as specified in Section 4.

**Step Three**

The necessity of using generated random maps more than one time meant that we needed to implement a method that exports a map to a text file. This method exports the information of the whole map in a CSV file where each line represents one cell, identified by its coordinates. This method will appear in the “WriteFile” class as “writeMap” and it is necessary to pass a “Map” object as a parameter.

Apart from exporting information from the agent, we also want to be able to import information for a strategy (in Behavioral Cloning, for example). We created a new class named “ReadFile”. Using again the “java.io” package we could import the results obtained from the PFA and the classifiers to make new strategies. These new strategies will be implemented in “Strategy” class in the two methods where we have to pass as parameter a link to the CSV file that contains the strategy. In the “ReadFile” class we can also import a map from a CSV file. We need this method in order to use a random map more than once.

Finally, we obtain the following class diagram that is a brief representation of our JAVA project.

![Class Diagram](image)

Figure 9: Class Diagram
5.2 Python Programming Language

Python is a concise and compact dynamically typed programming language, which makes it much easier to use than JAVA. It promotes the use of functions, which are equivalent to JAVA methods. We used it to train the Probabilistic Final Automaton and to evaluate the models obtained as described in section 3.2. Also, all tables were automatically generated in Python.

Concretely, we implemented a function to import all the observations of the agent from the CSV files. When we import our data, we have to take into account which of the three models is used (see Section 2).

After importing all the observations, we implement a function that trains a PFA and another one that computes the probability of a given trace to have been generated by that PFA according to the formulas that appear in Section 2.1. Also, we implement functions that allow us to compute the Predictive Accuracy of Figure 7 and the Monte Carlo distance values of Figure 8.

5.3 Knime Analytics Platform

KNIME is a data mining tool that allows us to get, for each one of the models, the range of successful predictions of the next move of the agent according to its state, using the classifiers mentioned in section 2.1. It is an open source and licensed software, and also the tool used in the course of Machine Learning and Data Mining for the final project, which made it a perfect candidate for accomplishing this task.

KNIME projects are composed by nodes. Each node is defined and properly explained in KNIME, so it is easy to know which node serves for each task. To import the observations of the agent that we extract from the JAVA project, we use a node called “CSV Reader”. This node generates a database with all the information that appears in the CSV file.

For each one of the classifiers, we used two nodes “CSV Reader” because one is dedicated to training and the other for testing. After that, we have to classify the movement that the agent will do (from 1 to 4); most of the classifiers used String variables, so we had to group the values we wanted to classify from the database in sets composed by Strings; for example, the number 1 will be in the “Bin 1”. For this operation in KNIME we used a node called “Auto Binner”.

Then, after having all the database ready for prediction, we used different classifiers. Most of them are composed by two related nodes: a “Learner” node and a “Predictor” node. Training sets will go to the “Learner” node to train the classifier and testing values go to the “Predictor” node. Finally, we use the “Score” node to obtain the accuracy of the classifier, which tells us whether the classifier works properly. The values obtained are reported in Figures 7 and 8.
In the sequel we show the scheme of the KNIME workflow used for Decision Trees. The other workflows are very similar to this one.

Figure 10: Knime Model
6 Concluding Remarks and Future Work

The realization of this project has given me great satisfactions at both professional and personal level. On one hand, all proposed objectives defined were accomplished.

We have proposed a model for behavior recognition. Behaviors are identified with tasks (or strategies for solving a given problem). Our system uses PFAs and correctly identifies tasks performed by the actor whenever those tasks have a certain random component. One of the strengths of our system is that it captures the stochastic aspects of behaviors. On the other hand, another characteristic of our model is the difficulty to distinguish between a certain strategy and a similar strategy perturbed with some degree of randomness. The inference technique is based on the greatest likelihood probability value generated by the PFAs of the model. The major computational limitation in our fully observable state space is the amount of memory required by the trained automaton. A possible solution is to employ only a few amount of non-observable internal states. Another possible approach is the use of some notion of context learning in order to reduce the number of possible states.

We have proposed several models for behavior cloning and two evaluation metrics. First, we used ML classifiers to predict the action of the agent in a given state (a deterministic model). Then, we trained a PFA to estimate the probabilities of the agent’s actions in a given state (a stochastic model). In both cases, we measured the predictive accuracy on new unseen data: deterministic approaches turned out to give slightly better results, but the difference was negligible. Nevertheless, when comparing the trace of the learned strategy with the trace of the original agent on a randomly generated map, we could experimentally check that PFAs do a better job than any of the classifiers under study.

On the other hand, this project has given me the opportunity to deepen my knowledge in machine learning and data mining.

Finally, at a personal level, this work lead to my debuting in scientific writing by contributing to two conference articles, accepted for publication at CAEPIA 2015 [11] and at UCAmI 2015 [22] as LNCS proceedings.

We plan to extend this methodology to a more realistic type of robot, which can perform rotations of different angles and whose position is represented by real values on the map. Moreover, the set of agent’s strategies should be extended to include non-stationary type behaviors and other more sophisticated behaviors, such as the outward-moving spiral of the Roomba robot. Future work also contemplates the usage of probabilistic transducers, to take into account the input-output (state-action) nature of the observations composing the learning traces in the observation scenario. Finally, another future work direction is getting our models to deal with a greater fixed amount of memory, with the obvious computational limitation induced by our fully observable state space, which is directly proportional with the amount of memory required by the trained automaton.
References


