A Self-learning Nurse Call System

Femke Ongenae[a,*], Maxim Claeys[a], Wannes Kerckhove[a], Thomas Dupont[a], Piet Verhoeve[b], Filip De Turck[a]

[a]*Department of Information Technology (INTEC), Ghent University - iMinds, Gaston Crommenlaan 8 bus 201, B-9050 Ghent, Belgium*
[b]*iMinds VZW, Gaston Crommenlaan 8 bus 102, B-9050 Ghent, Belgium*

Abstract

The complexity of continuous care settings has increased due to an ageing population, a dwindling number of caregivers and increasing costs. Electronic healthcare (eHealth) solutions are often introduced to deal with these issues. This technological equipment further increases the complexity of healthcare as the caregivers are responsible for integrating and configuring these solutions to their needs. Small differences in user requirements often occur between various environments where the services are deployed. It is difficult to capture these nuances at development time. Consequently, the services are not tuned towards the users' needs.

This paper describes our experiences with extending an eHealth application with self-learning components such that it can automatically adjust its parameters at run-time to the users' needs and preferences. These components gather information about the usage of the application. This collected

*Corresponding author: Tel.: +32 9 331 49 38, Fax: +32 9 331 48 99

Email addresses: `Femke.Ongenae@intec.UGent.be` (Femke Ongenae), `Maxim.Claeys@intec.UGent.be` (Maxim Claeys), `Wannes.Kerckhove@intec.UGent.be` (Wannes Kerckhove), `Thomas.Dupont@intec.UGent.be` (Thomas Dupont), `Piet.Verhoeve@iminds.be` (Piet Verhoeve), `Filip.DeTurck@intec.UGent.be` (Filip De Turck)

information is processed by data mining techniques to learn the parameter values for the application. Each discovered parameter is associated with a probability, which expresses its reliability. Unreliable values are filtered. The remaining parameters and their reliability are integrated into the application.

The eHealth application used is the ontology-based Nurse Call System (oNCS), which assesses the priority of a call based on the current context and assigns the most appropriate caregiver to a call. Decision trees and Bayesian networks are used to learn and adjust the parameters of the oNCS. For a realistic dataset of 1,050 instances, correct parameter values are discovered very efficiently as the components require at most 100 milliseconds execution time and 20 megabyte memory.

Keywords:
Self-learning, Adaptive, Ontology, eHealth, Nurse call system

1. Introduction

Due to a longer life expectancy and dwindling fertility rates, the percentage of people over 60 is growing more rapidly than any other age group [1]. Because of health problems, a lot of the elderly are no longer able to live independently and require some form of institutionalized long-term care, e.g., residential care or long stays in the hospital [2]. These developments are accompanied by emerging staff shortages in the formal care sector. In 2006, the World Health Organization (WHO) reported an estimated shortage of almost 4.3 million doctors, midwives, nurses and support workers worldwide [3]. Moreover, people are increasingly living longer with one or more chronic diseases, which increases the complexity of diagnosis and treatment and requires more personalized healthcare and specialized staff. Consequently, the healthcare costs have also been on the rise. Spending on healthcare almost consistently grows faster than the Gross Domestic Product (GDP) [4].

To achieve a more optimized use of resources and rostering of staff and to reduce the healthcare costs, Information Technology (IT) and technological equipment, e.g., monitoring equipment and Electronic Patient Records (EPR), are often introduced in institutionalized healthcare settings [5]. Electronic Healthcare (eHealth) software and services can then be built that take advantage of all the collected information to ideally support caregivers in their daily work practices. The benefits of eHealth, such as improved operational efficiency, higher quality of care, and positive return on investments, have been well documented in the literature [6]. However, the increased introduction of eHealth also increases the complexity of healthcare as the caregivers are responsible for tweaking and configuring the eHealth solutions to suit their needs. The various healthcare environments where the services are deployed, e.g., different nursing units or hospital departments, have slightly different requirements pertaining to how the collected information about the patients, caregivers and environment is taken into account. It is difficult to capture these small nuances at development time as domain experts often find it difficult to assess these parameters. Consequently, the resulting services are not really personalized towards the needs and preferences of the caregivers and they have to significantly alter their workflow patterns to accommodate the technology instead of the other way around [7]. This hinders the adoption of these services [8].

An important way to coordinate work, communicate and provide continuous care is by making use of a nurse call system. In previous research, we have developed an ontology-based Nurse Call System (oNCS) [9], which finds the most appropriate caregiver to handle a call based on profile and environment information captured in an ontology, e.g., the risk factors of the patient, the locations of the staff and patient, the priority of the call and the current tasks of the staff. Simulations showed that the workload distribution amongst nurses and the arrival times of caregivers at calls are positively influenced by using the oNCS [9]. However, user tests performed with the prototype also showed that small nuances were often required in how the profile information was taken into account within a specific healthcare setting. Domain experts also found it difficult to specify the parameters of the oNCS, i.e., which context should be taken into account and how, at development time. However, little previous research has been done on how discovered trends and patterns can be used to automatically optimize the nurse call assignment. To resolve this issue, this paper presents an extension of the oNCS that allows automatically adjusting its parameters at run-time. More technical details about the self-learning, probabilistic, ontology-based framework, which was developed to realize this extension, can be found in Ongenae et al. [10].

The remainder of this paper is structured as follows. Section 2 gives an overview of the oNCS and the associated priority assessment and nurse call algorithm. Section 3 details the extension of the oNCS with components, which enable the autonomous adjustment of its parameters. The implementation of these components is discussed in Section 4, while Section 5 highlights how the correctness and performance of the extension was evaluated. Finally, Section 6 discusses the results and Section 7 summarizes the conclusions.

2. Ontology-based Nurse Call System

The main functionality of the oNCS is to provide an efficient support for wireless nurse call buttons and to employ a sophisticated nurse call algorithm that takes the profiles of the staff members and patients into account. A detailed description can be found in Ongenae et al. [9]. To realize the latter, a continuous care ontology [11] is used of which the most important classes pertaining to the dynamic algorithm are visualized in Figure 1. An ontology [12] formally models all the concepts and their relationships and properties within a domain. The ontology models people and associates them with their roles, location, profile, the hospital department they work or lie on, risk factors, and current tasks. Additionally, the ontology models the various types of nurse calls. Patients can launch three types of calls, i.e., service calls for "caring" requests, sanitary calls originating from sanitary spaces and normal calls for mostly medical requests. All the other calls, i.e., urgency, medical, technical and (sanitary) assistance calls, are launched by nurses. Each call is associated with a status and a priority. It is also indicated who made the call and which staff members are assigned to it.

When a new call is launched, the information captured in the ontology is used to assign the most appropriate staff member to the call. First, the priority of the call is determined, using the algorithm visualized in Figure 2. The ontology specifies for each risk factor a probability, which indicates the likelihood that a patient with this risk factor is classified as a high, medium or low risk patient. Patients can of course exhibit several risk factors. In

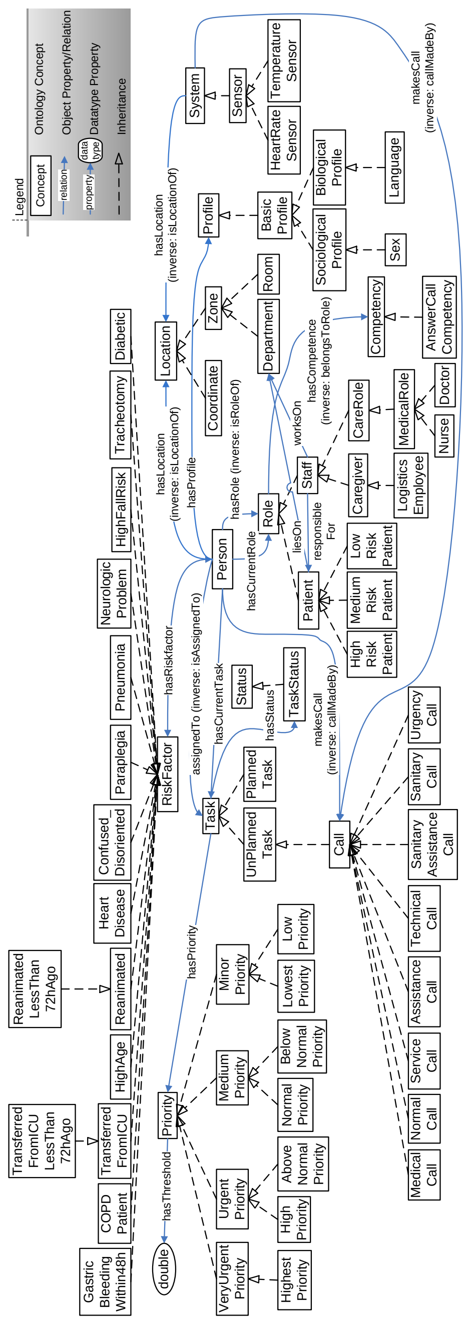


Figure 1: Prevalent concepts of the continuous care ontology

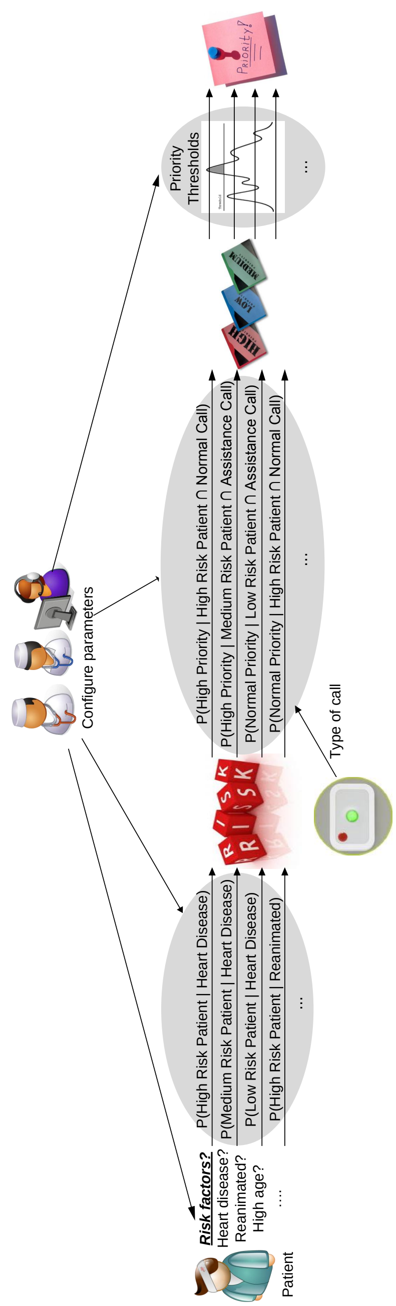


Figure 2: Probabilistic priority algorithm

this case, probabilistic reasoning on the specified probabilities is used to determine for each risk group the combined likelihood that a particular patient belongs to it. As shown in Figure 1, there are seven priority levels. Probabilities are indicated in the ontology, which specify the likelihood that a call of a particular type made for a patient associated with a particular risk group has a certain priority. As example, Table 1 shows the probabilities for the types of calls, which can be launched by patients. For each of the seven priority classes, probabilistic reasoning is thus used to combine these probabilities with the probabilistic assignment of patient to risk groups in order to determine the likelihood that a call of a certain type has this priority. To determine the suitable priority for this call based on these probabilistic values, a threshold algorithm is used. Thresholds are specified in the ontology for each priority class. If the probabilistic value for the highest priority is higher than or equal to the threshold for this priority, the call is associated with the highest priority. If not, the same condition is checked for the other priority classes in the following order: high, above normal, below normal, normal, low and lowest.

The priority of the call is then combined with the other context information in the ontology to find the most appropriate staff member to handle the call, e.g., the distance between the caregivers and the patient, the current tasks of the available staff and the capability of the caregivers to handle the call based on their roles and competencies. For calls with a higher priority, more weight is given to finding a caregiver who is able to quickly rush to the patient and assess the situation. In contrast, other context information is given more weight for calls with a lower priority such as the profile and

Risk group	Type of call	Highest	High	Above normal	Normal	Below Normal	Low	Lowest
High	Normal		0.2	0.6	0.2			
	Sanitary		0.3	0.6	0.1			
	Service			0.2	0.2	0.6		
Medium	Normal			0.3	0.6	0.1		
	Sanitary			0.4	0.5	0.1		
	Service				0.2	0.4	0.4	
Low	Normal				0.6	0.3	0.1	
	Sanitary				0.7	0.2	0.1	
	Service					0.4	0.4	0.2

Table 1: Probabilistic assignment of priorities to calls based on the risk group of the patient and the type of call.

competencies of the staff. The assigned caregiver receives the call on a smartphone, which runs the mobile nurse call application. This application allows staff to receive, assess, accept and redirect calls. They are also able to change the priority of the call or indicate its reason. The information provided by the caregivers using the application is also captured in the ontology.

It can be noted that the adequate assessment of the priority of a call and thus the suitable assignment of caregivers to calls largely depend on the correctness of the specified probabilities and thresholds. The probabilities were determined by consulting various domain experts, i.e., nurses, doctors and developers of nurse call systems. The thresholds were determined by running simulations of calls and calculating the probabilistic priority assignment for these calls using the probabilities defined by the experts. Thresholds were then chosen such that the distribution of the simulated calls across the different priority classes deviates the least from the ideal distributions as determined by the experts, namely 5% - 10% - 25% - 35% - 25% - 0% - 0%, ordered from the highest to the lowest priority.

However, it was found that domain experts struggled upon defining these probabilities and ideal distribution of calls amongst priority categories. It was also difficult to extract these probabilities out of logging data as the current installed nurse call systems do not allow nurses to indicate or change the priority of a call. Furthermore, these parameters also slightly differ between hospital departments depending on the medical profile of the patients and the gravity of the treated pathologies. Therefore it was chosen to initialize the oNCS with the educated guesses of the domain experts and employ a self-learning framework. This framework allows automatically adjusting the probabilities and thresholds to the specific needs of the department where the oNCS is deployed.

3. Self-learning extension of the oNCS

The self-learning extension of the oNCS is visualized in Figure 3. The oNCS was built as an extension of the *Context-Aware Service Platform (CASP)* [13], which consists of a collection of *OSGi* [14] bundles to handle context information. The *Context Framework Layer* contains the *Con-*

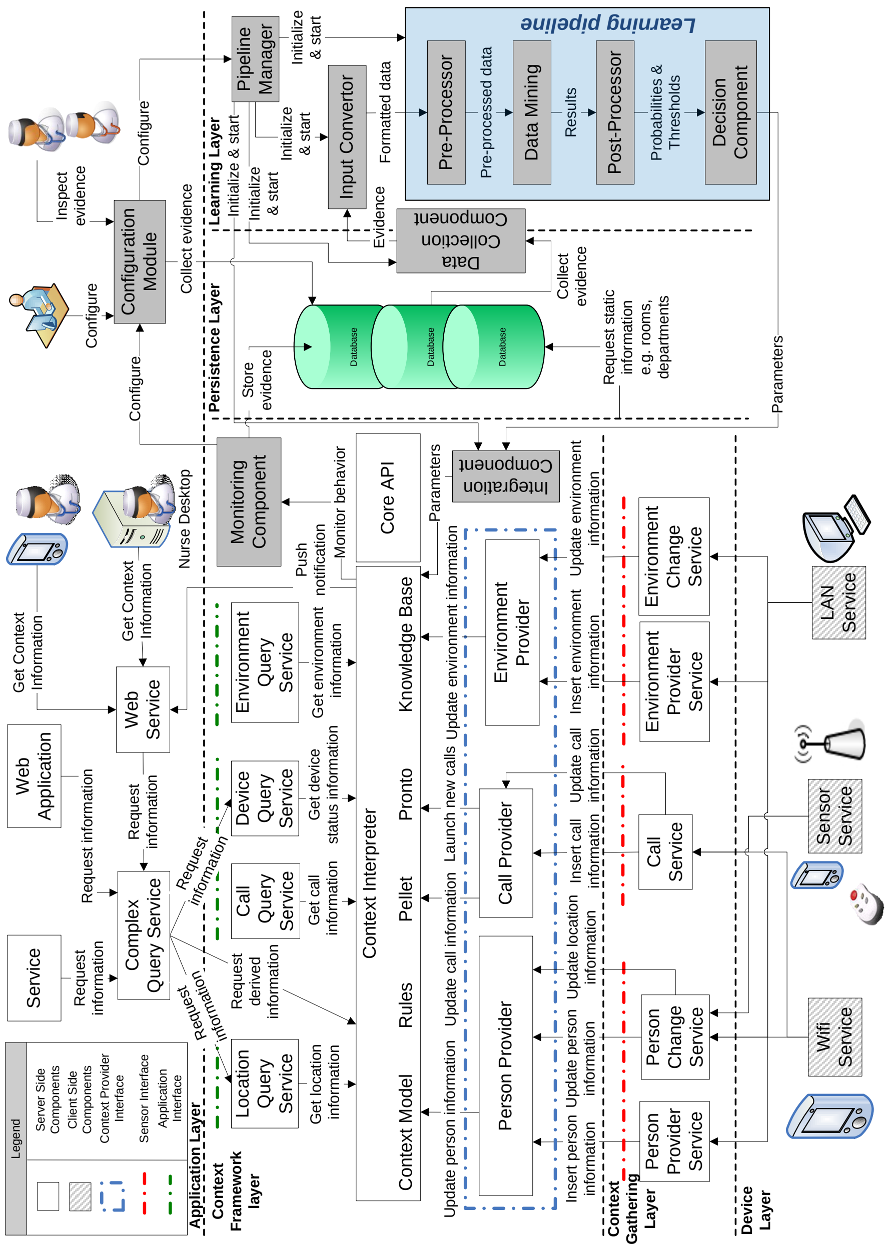


Figure 3: The oNCS extended with self-learning components

text Interpreter, which uses the continuous care ontology implemented in OWL [15] to model all the context information gathered about the environment, tasks, calls, patients and staff members. Pronto [16] is used to reason on the probabilistic information in the ontology, while Jena Rules [17] implement the threshold and nurse call algorithm. The *Context Providers* allow inserting new information into the *Knowledge Base*, e.g., a new nurse call or location of the patient. This new information can come from a database (*Persistence Layer*) or directly from a device (*Device Layer* and *Context Gathering Layer*). In contrast, the *Query Services* are used to extract derived knowledge from the *Knowledge Base*, such that it can be processed by the applications and services in the *Application Layer*. To improve the scalability and robustness of the system, context information can be stored in the *Persistence Layer*. This historical context information can then be exploited by the new self-learning components to adjust the parameters of the oNCS to the behavior of the users. These new components are indicated in grey.

The *Monitoring Component* constantly monitors the ontology to pick up trends and patterns in the way the priorities are assigned to calls by the caregivers. This component stores the evidence in the *Persistence Layer*. This evidence can be inspected by the domain experts by using the *Configuration Module*. When enough evidence has been collected, the *Learning Pipeline* can be initiated by the *Configuration Module*. The *Configuration Module* is notified of which data should be collected for the *Learning Pipeline*, either by the *Monitoring Component* or by the domain experts and administrator. The latter allows to initiate the *Learning Pipeline* with external data provided by the stakeholders. The *Configuration Module* configures the Pipeline

Manager to use the *Data Collection Component*, *Input Convertor* and *Integration Component* that suits this type of evidence. It also passes the correct parameters to the *Pipeline Manager*, which are needed to retrieve the data from the *Persistence Layer* using the *Data Collection Component.*

The *Learning Pipeline* is implemented using the Pipes-and-Filters architectural design pattern [18]. A pipeline consists of a set of filters, implementing small processing steps, which are connected by pipes. All the filters implement the same interface such that they can easily be rearranged, omitted or added. In this way, an extensible and flexible architecture is achieved.

The *Pipeline Manager* initiates the *Data Collection Component* to collect the necessary evidence. To achieve a flexible *Learning Pipeline*, a generic internal data format is used, which allows expressing both the information which is used as evidence and the probabilities and thresholds that are obtained as output. The format is largely based on the Attribute-Relation File Format (ARFF), which is the text file format used by the Waikato Environment for Knowledge Analysis (WEKA) [19]. The *Input Convertor* is responsible for converting the collected data to this format.

Next, the *Pipeline Manager* creates and starts the *Learning Pipeline. Pre-Processor* components can be used to clean the data, e.g., remove outliers or scale the data. This cleaned data is then processed by a *Data Mining* component to build a model, e.g, a Bayesian network or decision tree, that conveys the relation between the properties of the call, e.g., its type and the patient group, and it priority. This learned model is then processed by a *Post-Processor* component to extract the probabilities or thresholds for the oNCS.

Finally, to assess the correctness of the learned probabilities and thresholds, the *Decision Component* associates each discovered parameter with a probabilistic value expressing its reliability. When the calculated probabilistic value is too low, the discovered parameter is discarded and not adjusted in the oNCS.

The *Integration Component* is responsible for adjusting the parameters of the oNCS according to the probabilities and thresholds discovered by the *Learning Pipeline*. The associated probability, which was calculated by the *Decision Component*, is also added to the ontology to convey the reliability of the parameter values to the domain experts. If the parameter value in the ontology is the same as the learned value, the associated probability is updated to reflect its increased reliability, namely by using the average of the old and new probability.

4. Implementation details

Two scenarios can be identified, namely adjusting the probabilities and the thresholds. For the first scenario, this paper focuses on adjusting the probabilities, which indicate that a call has a particular priority based on its type and the risk group of the patient, who made the call. We will concentrate on learning the probabilities for calls launched by patients, i.e., normal, service and sanitary calls. Adjusting the probabilities that indicate the likelihood that patients belong to particular risk groups and for other types of calls, is analogous. The pipelines for these scenarios are visualized in Figures 4 and 5.

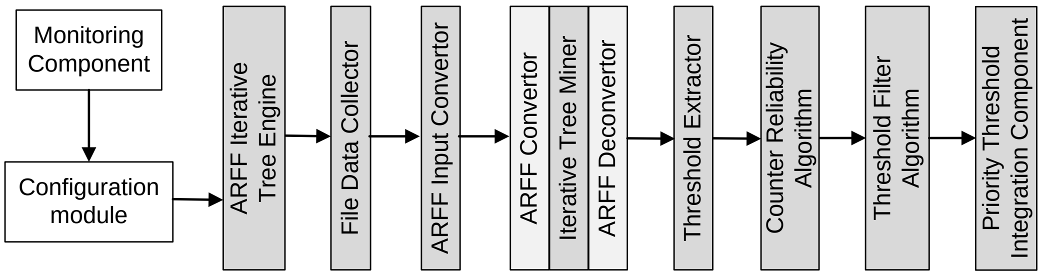


Figure 4: The *Learning Pipeline* used to learn and adjust the threshold parameters of the oNCS

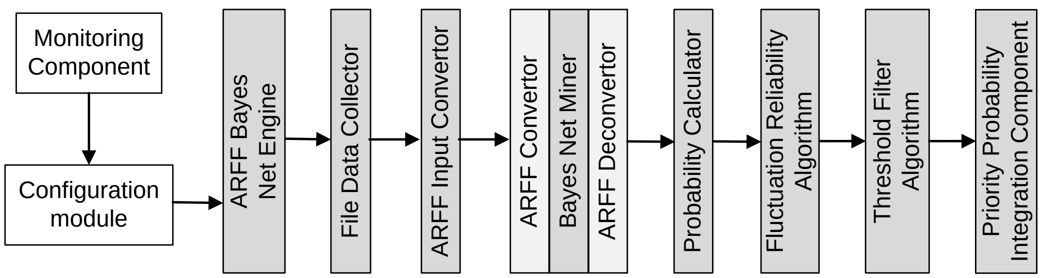


Figure 5: The *Learning Pipeline* used to learn and adjust the probabilistic parameters of the oNCS

4.1. Data collection and input conversion

The *Monitoring Component* monitors the ontology for new calls that receive the status `Finished`, indicating that the call has been completely handled and processed by the caregiver. The component collects the type and priority of the call using SPARQL [20] queries. The priority can be the one assigned by the oNCS, but it is also possible that the caregiver changed it using the mobile nurse call application. The *Monitoring Component* also retrieves the probabilistic assignment of the call to the seven priority classes based on its type and the probabilistic assignment of the patient to the three risk groups using the probabilistic reasoner Pronto. Finally, the probabilistic

Higest priority	High priority	Above Normal priority	Normal priority	Below Normal priority	Low priority	Lowest priority	Assigned priority
0.13	0.29	0.25	0.07	0.03	0.81	0.27	Above normal
0.18	0.96	0.46	0.45	0.06	0.66	0.01	High
0.12	0.18	0.20	0.00	0.00	0.00	0.70	*Below normal*
0.07	0.05	0.88	0.27	0.18	0.12	0.12	Above normal
0.06	0.02	0.15	0.11	0.02	0.56	0.59	Normal
0.44	0.11	0.53	0.27	0.21	0.51	0.31	Highest
0.20	0.09	0.12	0.01	0.04	0.54	0.03	*Above normal*

Table 2: Some example instances of the dataset to learn the threshold parameters

assignment of this patient to the three risk groups is requested. Based on this collected data, two datasets are created. Each instance in the dataset represents one call. The first is used to learn the threshold parameters and contains for each call the calculated probabilistic value for each priority class and the priority that was assigned it. Some example instances of this dataset are illustrated in Table 2. The second dataset is used to learn the probabilistic assignment of calls to priority classes based on their type and the risk group of the patient associated with the call. It indicates for each call the risk group of the patient, the type of the call and the assigned priority. Only calls with type normal, service or sanitary are retained. The risk group for the patient is chosen based on the calculated probabilistic assignment of this patient to the risk groups. For example, a patient with a heart disease has at least 50% chance of being a high risk patient. Some example instances of this dataset

Risk group	**Type of call**	**Assigned priority**
High	Normal	Above normal
Low	Sanitary	Low
Medium	Normal	Normal
High	Service	*High*

Table 3: Some example instances of the dataset to learn the probability parameters of the assignment of calls to priority classes

are listed in Table 3. To be able to demonstrate the *Input Convertor*, the datasets are saved in the ARFF format in the *Persistence Layer*.

The *Monitoring Component* keeps track of how many instances have been collected for each dataset. When a representative amount has been gathered, the *Configuration Module* is invoked to initiate the *Learning Engine*. Different *Learning Pipelines* are used to process each of the scenarios. These are implemented by different *Pipeline Managers*, e.g., *ARFFBayesNetEngine* or *ARFFIterativeTreeEngine*. The *Monitoring Component* also indicates to the *Configuration Module* the location of the data, its format and which *Pipeline Manager* should be used.

The *Configuration Module* configures the *Pipeline Manager* to use the appropriate *Data Collection Component* and *Input Convertor*, which suit the format of the data. A *File Data Collector* was implemented, which is able to read the data from a file at a specified location. The result is a `String`, which is provided to the *ARFF Input Convertor*. This *Input Convertor* is able to translate this `ARFF-String` to the internal format used by the *Learning*

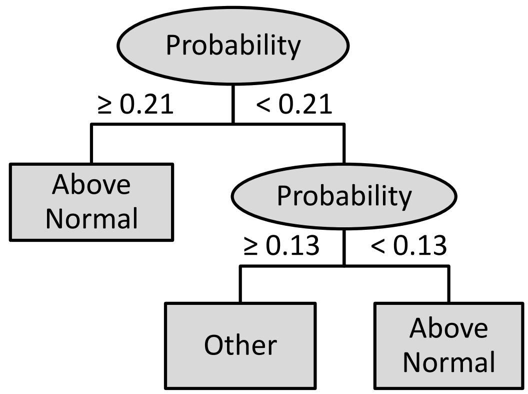


Figure 6: Example of a decision tree that encodes the learned knowledge about the threshold for the Normal priority class

Pipeline. A *Pre-Processor* is not needed for these scenarios as no anomalies can occur in the data.

4.2. Data mining and post-processing

Both scenarios use the WEKA data mining toolbox to learn the thresholds and probabilities of the oNCS. The first uses decision trees [21], while the latter uses a Bayesian network [22]. The following subsections detail how these models are built and how the parameters of the oNCS are derived from them. As previously mentioned, WEKA uses the ARFF data format to represent data. Therefore, (de)convertors were implemented that are able to translate the internal data format of the *Learning Pipeline* to and from the ARFF data format.

4.2.1. Discovering the thresholds using a C4.5 decision tree

The *Data Mining* filter needs to find relations in the threshold dataset between the probabilistic assignment of the calls to the priority classes and the priority that was eventually assigned to the calls. The former are considered input attributes, while the latter is called the label. Supervised [19] classification techniques [23] are used to discover these relations between the input attributes and the label. Decision trees are a well-known and easy to use classification technique. A decision tree consists of leaves, which each represent a possible value of the label, and internal nodes and branches, which represent the attributes on which the decision is based and the conditions that they must fulfill. An example is visualized in Figure 6. For this research, the J4.8 Java implementation of the C4.5 algorithm [24] in the WEKA data mining tool was used to build the decision trees.

The following knowledge of the threshold algorithm can be exploited to optimize the data mining. First, a call is assigned a priority x based on the probabilistic value for this priority class. Second, the probabilistic values for the priority classes are checked in a particular order, as discussed in Section 2. The probabilistic values for the priority classes, which occur later in the sequence than the assigned priority, are not taken into account for this call. Consequently, the decision was made to implement an *Iterative Decision Tree* algorithm, which builds a separate decision tree for each priority class. The decision trees are built in the same order as the priority classes are checked by the threshold algorithm. The dataset for each iteration consists only of one input attribute, i.e., the priority class under scrutiny. The label can also only assume two values, namely the considered priority and "Other". The latter is used to replace all other possible priority classes. Finally, all the instances that were assigned a priority class, which is checked earlier than the priority class for which the decision tree is being built, are removed

Above Normal priority	Assigned priority
0.25	Above normal
0.20	Other
0.88	Above normal
0.15	Other
0.12	Above normal

Table 4: Some example instances of the dataset to learn the threshold parameter for the Normal priority class

from the dataset. In this way, a dataset is built, which can be used by a decision tree to learn when the probabilistic value of a priority class is high enough to receive this priority as label. As an example, Table 4 visualizes some instances of the dataset for the Above Normal priority class, which were derived from the original dataset visualized in Table 2. It can be noted that all the instances were removed, which were assigned the Highest and High priority, as these are checked earlier by the threshold algorithm.

The *Iterative Decision Tree* algorithm builds the decision tree for each priority class. The J4.8 algorithm outputs a textual representation of the decision tree. For example, the tree visualized in Figure 6 is represented as follows:

N0 [label=“Probability”]

N0 $\rightarrow$ N1 [label=“ $>=$ 0.21”]

N1 [label=“Above Normal”]

N0 $\rightarrow$ N2 [label=“ $<$ 0.21”]

N2 [label=“Probability”]

N2 $\rightarrow$ N3 [label=“ $>=$ 0.13”]

N3 [label=“Other”]

N3 $\rightarrow$ N4 [label=“ $<$ 0.13”]

N4 [label=“Above Normal”]

The nodes and branches are identified and translated to the internal data format such that the results can be forwarded to the *Post-Processor*.

The *Threshold Extractor Post-Processor* was implemented, which extracts the discovered thresholds out of the textual representation of each decision tree. For each decision tree, all the branches are considered that result in a leaf with the priority class label, associated with this decision tree. The branches, which result in a leaf with the label “Other”, are ignored. All the considered branches are followed from the leaf up to the root and the conditions are checked. The condition that represents the highest lower bound is chosen as threshold for this priority class, i.e., a condition of the type $\geq x$ where x is the highest value for a condition of this type in this tree. The discovered thresholds are represented in the internal data format and forwarded to the *Decision Component*.

4.2.2. Discovering the probabilities using a Bayesian network

In this scenario, the *Data Mining* filter needs to find probabilistic relations between two input attributes, i.e., the type of the calls and the risk group of the patients, and the priority labels that were eventually assigned to the calls. Bayesian networks can ideally be used to discover these probabilistic relations. Bayesian networks are graphical models that represent the conditional dependencies between a set of variables as a directed acyclic graph. Each node is associated with a probability function. This function is able to calculate the probability of the variable represented by this node based on a particular set of values for the variables, which are represented by nodes that are parents of this node. Different techniques can be used to build Bayesian networks. Naive Bayesian networks assume that all the input attributes are conditionally independent. Consequently, a network is obtained in which the label is connected to each input attribute, but the input attributes are not connected to each other. As the risk group of the patient is independent of the types of calls this patient makes, Naive Bayesian networks are used for this research. The BayesNet implementation of WEKA was used to construct the network. The probabilities obtained by building the network are retrieved from WEKA and represented in the internal data format.

The *Probability Calculator Post-Processor* was implemented to calculate the needed probability parameters for the oNCS. To explain this calculation, the following notation is introduced:

- The risk group input attribute is represented by A and has $n1$ possible values $a_1, ..., a_{n1}$.
- The type of call input attribute is depicted by B and has $n2$ possible

values $b_1, ..., b_{n2}$.

- X represents the label, i.e., the priority class, and has m possible values $x_1, ..., x_m$.

The output of the BayesNet algorithm contains the following probabilities:

- $P(X = x_i)$, $\forall\ i \in [1, m]$.
- $P(A = a_i|X = x_j)$, $\forall\ i \in [1, n1]$ and $\forall\ j \in [1, m]$.
- $P(B = b_i|X = x_j)$, $\forall\ i \in [1, n2]$ and $\forall\ j \in [1, m]$.

Bayes' rule can be used to calculate the probability parameters for the oNCS:

$$P(X = x_i|A = a_j \cap B = b_k) = \frac{P(A = a_j \cap B = b_k|X = x_i)P(X = x_i)}{P(A = a_j \cap B = b_k)}$$
$$\text{where } i \in [1, m], j \in [1, n1] \text{ and } k \in [1, n2] \quad (1)$$

Only the probabilities $P(X = x_i)$ can be directly derived from the Bayesian network. As attributes A and B are conditionally independent, the other term of the numerator can be calculated as follows:

$$P(A = a_j \cap B = b_k|X = x_i) = P(A = a_j|X = x_i)P(B = b_k|X = x_i)$$
$$\text{where } i \in [1, m], j \in [1, n1] \text{ and } k \in [1, n2] \quad (2)$$

The probabilities on the right hand side of this equation can also be derived from the Bayesian network. These calculated probabilities can be used to derive the denominator using the law of total probability as follows:

$$P(A = a_j \cap B = b_k) = \sum_{i=1}^{m} P(A = a_j \cap B = b_k | X = x_i) P(X = x_i)$$
$$\text{where } j \in [1, n1] \text{ and } k \in [1, n2] \quad (3)$$

By inputting the results of Equations 2 and 3 in Equation 1, the needed probability parameters can be calculated. These parameters are represented in the internal data format and forwarded to the *Decision Component*.

4.3. Filtering the results and expressing their reliability

As mentioned in Section 3, the *Decision Component* attaches probabilities to the discovered parameters to express their reliability to the users.

To assess the reliability of the thresholds, the *Counter Reliability Algorithm* is used. This algorithm applies the new thresholds to the original dataset. For all the calls of a particular priority, it then calculates the percentage that received this priority correctly by the new threshold algorithm. For example, suppose that 0.44 - 0.35 - 0.21 - 0.07 - 0.2 - 0 - 0 are discovered as thresholds, ordered from the Highest to the Lowest priority. If these thresholds are applied to the dataset visualized in Table 2, the threshold for the Above Normal priority achieves 67% reliability, as the first and fourth calls are correctly assigned the Above Normal priority, while the last call incorrectly receives the Low priority.

The *Fluctuation Reliability Algorithm* computes the reliability of the discovered probability parameters. It first calculates the difference x between the new and old parameter value. When the Learning Pipeline is used for the first time to learn the probability parameters, the probability parameters

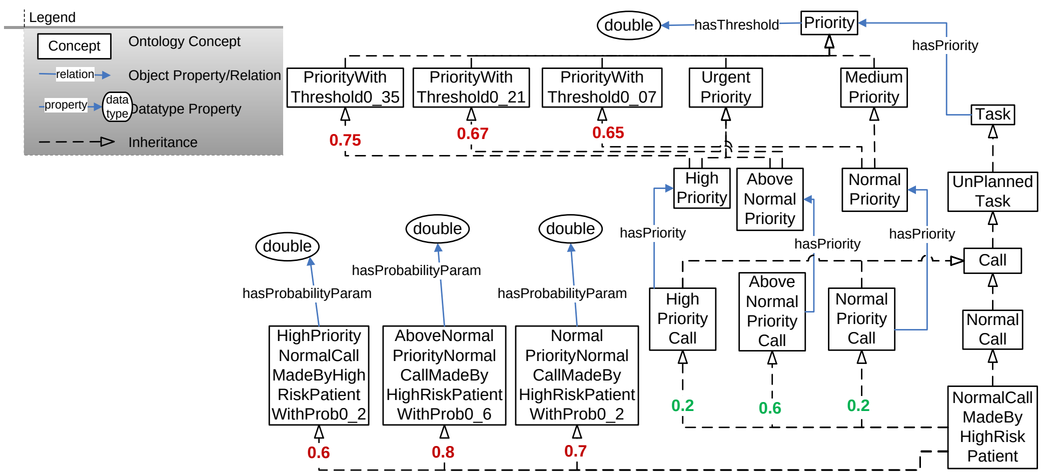


Figure 7: Integrating the learned parameters of the oNCS into the ontology with an associated probability to express their reliability

in the ontology are used as the old parameter values. In later runs of the pipeline, the parameter values discovered in the previous run are used as old parameter values. The reliability of the new parameter is then set to $1 - x$. Consequently, if the *Learning Pipeline* consecutively discovers very similar parameter values, the reliability increases. The reliability thus increases if the value of the parameter converges.

A simple filter algorithm, namely the *Threshold Filter Algorithm*, was implemented, which filters the parameters for which the reliability is lower than a specified threshold, e.g., 50%. These parameters are not adjusted in the oNCS. However, these discovered parameters are stored such that they can be used by subsequent runs of the *Learning Pipeline*, e.g., as old parameter values in the *Fluctuation Reliability Algorithm*.

4.4. Integrating the parameters in the oNCS

4.4.1. Integrating the thresholds in the oNCS

The *Priority Threshold Integration Component* is responsible for integrating the discovered thresholds into the oNCS with their associated probability. To integrate a discovered threshold for a particular priority class, this component first checks whether this priority was already associated with this threshold, i.e., the parameter value has not changed. If this is the case, only the reliability is changed, as explained further. To integrate a new threshold, a subclass of the `Priority` class is introduced in the ontology, as shown in Figure 7. For example, to integrate the threshold of 0.21 for the Above Normal priority, the `PriorityWithThreshold0_21` class is created. This class is defined as follows:

Priority AND (hasThreshold VALUE 0.21ˆˆdouble)

If this class already exists in the ontology, it is re-used. The priority class associated with this threshold is then defined as a subclass of this class, e.g., `AboveNormalPriority` becomes a subclass of `PriorityWithThreshold0_21`. The priority also inherits the definition and is thus effectively associated with the correct threshold. The subclass relationship with the previous threshold is removed.

Next, the associated reliability is expressed in the ontology. Pronto is used to represent and reason on the probabilistic information in the ontology. To express probabilistic knowledge, Pronto uses Generic Conditional Constraints (GCCs) [25]. A GCC is of the form (D—C)[l,u] where D and C are the classes in the ontology and [l,u] is a closed subinterval of [0,1].

To represent these GCCs in the ontology, Pronto employs subsumption axiom annotations. For example, to express that the 0.21 threshold for the normal priority class only has a reliability of 67%, the subclass relationship between the `AboveNormalPriority` and `PriorityWithThreshold0_21` concepts is annotated as follows:

< owl11:Axiom >

< rdf:subject rdf:resource="#AboveNormalPriority" >

< rdf:predicate rdf:resource="&rdfs;subClassOf" >

< rdf:object rdf:resource="#PriorityWithThreshold0_21" >

< pronto:certainty > 0.67;0.67 < /pronto:certainty >

< owl11:Axiom >

Pronto uses probability intervals to express probabilistic knowledge. However, as illustrated in the previous example, strict probabilities can easily be expressed by defining an interval with an equal upper and lower limit. When a new threshold is associated with a priority, the reliability calculated by the *Decision Component* is used. If the priority was already connected to this threshold, the reliability is changed to the average of the old and the new reliability.

4.4.2. Integrating the probabilities in the oNCS

The probability parameters, which express the the likelihood that a call of a particular type made by a patient belonging to a specific risk group has a particular priority, are represented in the ontology by annotated subsumption axioms between `Call` classes, as illustrated in Figure 7. For example, the following annotated subsumption axiom expresses that a normal call made by a high risk patient has 0.2 probability of having a normal priority:

< owl11:Axiom >

< rdf:subject rdf:resource="#NormalCallMadeByHighRiskPatient" >

< rdf:predicate rdf:resource="&rdfs;subClassOf" >

< rdf:object rdf:resource="#NormalPriorityCall" >

< pronto:certainty > 0.2;0.2 < /pronto:certainty >

< owl11:Axiom >

These two classes are defined as follows:

NormalCallMadeByHighRiskPatient:

NormalCall AND (callMadeBy SOME (hasRole SOME HighRiskPatient))

NormalPriorityCall:

Call AND (hasPriority SOME NormalPriority)

To integrate the discovered probability parameters in the oNCS, the *Priority Probability Integration Component* just changes the probabilistic value in the annotated subsumption axiom.

Next, the *Priority Probability Integration Component* associates the reliability with this discovered parameter. To realize this, a new class is created in the ontology that represents the annotated subsumption axiom. For example, to represent the previous subsumption axiom, the class `NormalPriorityNormalCallMadeByHighRiskPatientWithProb0_2` was created with the following definition:

hasProbabilityParam VALUE 0.2˜double

An annotated subsumption axiom is then created, which associates the input attributes, i.e., a call of a particular type made by a patient belonging to a specific risk group, with this new class and annotates this subclass relationship with the reliability. For example, the following annotated subsumption axiom is created for the running example to express that this parameter value has a reliability of 70%:

< owl11:Axiom >

< rdf:subject rdf:resource="#NormalCallMadeByHighRiskPatient" >

< rdf:predicate rdf:resource="&rdfs;subClassOf" >

< rdf:object rdf:resource="#NormalPriorityNormalCallMadeBy
HighRiskPatientWithProb0_2" >

< pronto:certainty > 0.7;0.7 < /pronto:certainty >

< owl11:Axiom >

Note that if the parameter value has not changed, the reliability is updated to 100%, as this reliability expresses how much the parameter value deviates from the previous value.

5. Evaluation set-up

To adequately evaluate the correctness and performance of the self-learning components, generated datasets are used for both scenarios. In this way, trends can be introduced into the datasets, which should be discovered by the *Learning Pipeline*. To achieve realistic datasets, noise is introduced. The following subsections detail how these datasets were generated and noise was added. The datasets were generated in the ARFF format and stored in the *Persistence Layer* so that they can be retrieved by the *File Data Collector* and translated to the internal format by the *ARFF Input Convertor*.

To evaluate the applicability of the framework, it is important to assess the correctness of the derived parameters. The correctness of the data mining techniques is influenced by the size of the dataset and the amount of noise. To assess the influence of the latter, the *Learning Pipeline* was consecutively applied to datasets of the same size, but with an increasing amount of noise. The amount of noise is varied from 0% to 50% in steps of 1%. It is unnecessary to increase the noise percentage beyond 50% as a random label is assigned at this point and the dataset becomes meaningless. The amount of noise needs to be increased in a dataset of realistic size. Each instance in the dataset corresponds to one made by or for a patient. Out of logging data of the nurse call system installed at Ghent University Hospital [26], it was derived that one average five calls are made per 24 hours by or for a specific patient. Consequently, for a nursing unit containing on average 30 patients, 1,050 calls are launched per week on average. Therefore, to assess the influence of noise, datasets were generated containing 1,050 instances.

The influence of the size of the dataset on the correctness is evaluated by consecutively applying the *Learning Pipeline* to datasets of increasing size. The dataset sizes range from 100 to 2,000 instances in steps of 100 instances. This range also contains the realistic dataset size for each of the scenarios.

It is also important to evaluate the performance, i.e., execution time and memory usage, of the developed *Learning Engine*. Although, the learning process will mostly run in the background, it is important to assess the amount of resource usage. Most healthcare environments have a limited amount of resources and delegating the processing to the cloud is often difficult because of privacy issues. To evaluate the influence of noise on the performance, the same datasets were used as for the correctness tests. However, to assess the influence of the size of the dataset, datasets were generated with sizes ranging from 1,000 to 30,000 in steps of 1,000 instances. Bigger datasets were used as it is important to explore the limits of the proposed self-learning components.

To achieve reliable results, each test was repeated 35 times, of which the first three and the last two were omitted during processing. For each run, a new dataset was generated. Finally, the averages across the 30 remaining runs are calculated and visualized in the form of graphs. The tests were performed on a computer with the following specifications: 4096 megabyte (MB) (2 x 2048 MB) 1067 megahertz (MHz) Double Data Rate Type Three Synchronous Dynamic Random Access Memory (DDR3 SDRAM) and an Intel Core i5-430 Central Processing Unit (CPU) (2 cores, 4 threads, 2.26 gigahertz (GHz), 3 MB cache).

5.1. Generating the dataset to discover thresholds

As indicated in Section 4.1, the dataset consists of seven input attributes, i.e., the probabilistic assignment of a call to the priority classes. As label, the assigned priority of the call is used. The dataset is generated in such a way that discovered thresholds should be the ones that are currently being used by the oNCS, i.e., 0.21 - 0.3 - 0.24 - 0 - 0.05 - 0 - 0, ordered from the highest to the lowest priority.

To generate a new instance of the dataset, a priority label is first chosen. The label is chosen such that the distribution of the generated calls amongst the different priority classes reflects the following realistic distribution determined by domain experts: 5% - 10% - 25% - 35% - 25% - 0% - 0%, ordered from the highest to the lowest priority. Based on this label, the probabilistic values for the input attributes are generated. For all the priority classes that are checked earlier by the threshold algorithm than the assigned priority, a probabilistic value is randomly generated that is smaller than the threshold for this priority. For example, if a call with a High priority is being created, then the probabilistic value for the Highest priority will be lower than 0.21. For the assigned priority, a random probabilistic value is generated, which is higher than its threshold. Finally, for the remaining priority classes, a random probabilistic value is generated. The thresholds for these priorities are thus not taken into account.

To introduce noise in the generated datasets, the priority labels of some generated instances are changed. This means that they receive a different label from the one which would be assigned by the threshold algorithm and which was used to generate these instances. For a noise percentage of x, each generated instance has $x\%$ chance of being assigned a priority label that is one level higher or one level lower than the correct priority label. Some generated instances are shown in Table 2. The labels indicated in italics represent noise.

5.2. Generating the dataset to discover the probabilities for the priorities

The dataset generated for this scenario contains two input attributes, i.e., the type of the call and the risk group of the patient who made it, and the assigned priority as label. To create a new instance, a risk group is randomly assigned based on the following distribution: 20%, 50% and 30% chance of being a High, Medium or Low Risk patient respectively. Moreover, the instance has 60%, 30% and 10% chance of being a Normal, Sanitary and Service call respectively. These distributions were determined based on input from domain experts. Using the parameters already defined in the oNCs and visualized in Table 1, the probabilistic assignment of this generated call to the various priority categories is determined. For example, if an instance is generated with the input attributes Normal type of call and High Risk patient, then it has 20%, 60% and 20% chance of receiving the High, Above Normal and Normal priorities respectively. Based on this distribution, a priority is randomly chosen as label.

Similar to that in the previous scenario, noise is introduced by changing the label of an instance to a priority that is one lever higher or lower than the assigned one. Some generated instances are shown in Table 3. The labels indicated in italics represent noise.

6. Results and discussion

6.1. Correctness of the discovered thresholds

To assess the correctness, the relative error of the discovered thresholds is calculated. The relative error expresses how much the learned threshold deviates from the threshold on which the dataset generation was based. For

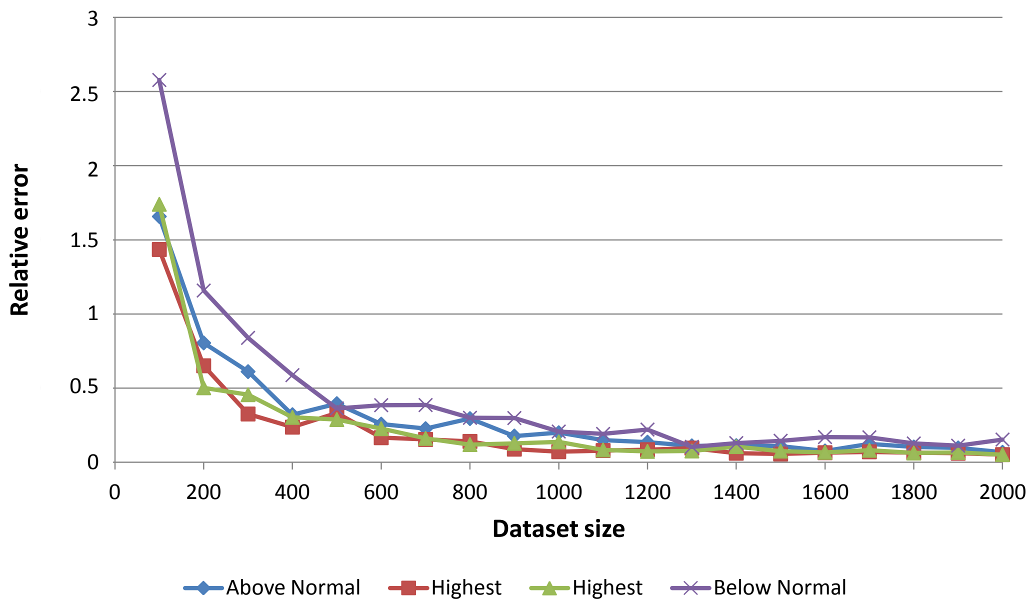


Figure 8: The relative errors (%) of the thresholds discovered for the different priority categories as a function of the size of the dataset

example, a relative error of 5% for the threshold of the Above Normal priority indicates that the discovered threshold deviates at most 5% from 0.24. The oNCS employs a threshold of 0 for the Normal, Low and Lowest priority categories to ensure that the default priority assigned to calls is the Normal priority. The Low and Lowest priorities are generally reserved for particular types of calls, e.g., technical assistance calls. Because of the way the dataset generation algorithm takes these zero thresholds into account to generate the instances, these thresholds are always discovered. Therefore, only the other, non-zero, thresholds are discussed.

Figure 8 depicts the relative error of the discovered thresholds as a function of the dataset size. It can be derived that very accurate thresholds are obtained, even when datasets with a small amount of instances are used. When the dataset contains at least 500 instances, the relative error stays smaller than 0.5% for all the thresholds. As mentioned previously, on average five calls are launched per patient in a department with on average 30 patients. Consequently, four days after deployment of the oNCS enough data would be collected to accurately adjust the thresholds to the behavior of the caregivers. Note that for small datasets, more accurate results are obtained for the thresholds of higher priority classes. A separate decision tree is built for each priority class, based on a subset of the total dataset. In these subsets the instances are removed, which received as label a higher priority class than the one that the decision tree is currently being built for. Consequently, the decision trees for lower priorities are trained on less data than the decision trees for higher priorities. As a result, these lower priorities exhibit a higher relative error for small datasets.

Figure 9 visualizes the relative errors for the discovered thresholds as a function of the amount of noise in a realistically sized dataset of 1,050 instances. It is clear that the *Learning Pipeline* is insensitive to a noise rate of less than 20%, as they result in relative errors for the thresholds of less than 5%. If the amount of noise increases beyond this point, the relative errors quickly rise to 10% and higher. The relative error of the threshold of the Below Normal priority is higher than the ones of the Normal and High priority because it is trained on smaller datasets, as explained in the previous paragraph. The relative error of the threshold of the Highest priority is much higher than the others. This is the first threshold that needs to be determined. Consequently, it is trained on a dataset with a very high

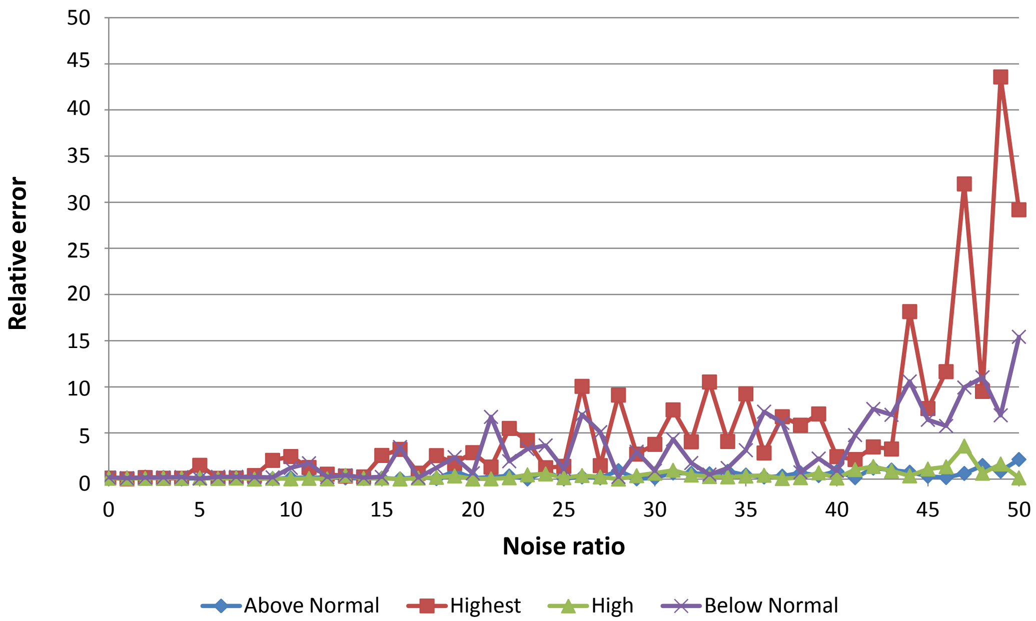


Figure 9: The relative errors (%) of the thresholds discovered for the different priority categories as a function of the amount of noise in the dataset

amount of instances labeled as "Other". This skewed dataset, containing more negative than positive examples, results in a higher relative error for this priority.

6.2. Correctness of the discovered probabilities

The dataset for this scenario consists of two input attributes, namely the risk group of the patient and the type of the call, each of which can have three possible values. The priority label can have seven possible values. Consequently the Bayesian network needs to determine 63 probability parameters. It is difficult to give a clear overview of all the calculated parameter values for all the different dataset sizes and noise ratios. Therefore, Table 5 visual-

Risk group	**Type of call**	**Relative error**						
		Highest	**High**	**Above normal**	**Normal**	**Below normal**	**Low**	**Lowest**
High	Normal		1	3	1			
	Sanitary		6	2	5			
	Service			4	4	*16*		
Medium	Normal			0	4	2		
	Sanitary			4	4	1		
	Service				2	5	*14*	
Low	Normal				6	3	3	
	Sanitary				1	2	2	
	Service					3	2	*12*

Table 5: Relative error (%) for the discovered probability parameters for a dataset with 1,050 instances

izes only the relative errors for the discovered probabilities for a dataset of realistic size, i.e., 1,050 instances, without noise. Despite the large number of parameter values that need to be deduced from a relatively small dataset, the relative errors are quite small. Three discovered probabilities have a relative error bigger than 10%. These errors are indicated in italics in Table 5. However, all the other derived parameter values deviate only on average 3% and maximum 6% from the correct value. It can also be noted that higher relative errors correspond to situations that do not occur often in reality. As the dataset is generated based on realistic distributions, these situations are represented by less instances in the dataset. This makes it more difficult for the Bayesian network to obtain a correct parameter value for these situations. For example, as explained in Section 5.2, an instance only has 10% chance to receive the type Service and 20% chance of being launched by a High Risk patient. Consequently, there's only 2% chance that an instance is generated that fulfills both of these criteria. As a result, the relative error for this probabilistic value is 0.16%.

6.3. Execution time of the threshold Learning Pipeline

The execution time as a function of the size of the dataset is depicted in Figure 10. The execution times of the *Threshold Extractor*, *Counter Reliability Algorithm* and *Threshold Filter Algorithm* are negligible compared to the execution times of the visualized components. The execution time of the *Priority Threshold Integration Component* depends heavily on the complexity and the amount of data in the ontology as this component checks the consistency of the ontology after the parameters are adjusted. As the ontology was not initialized with a realistic data set, e.g., representing a realistic

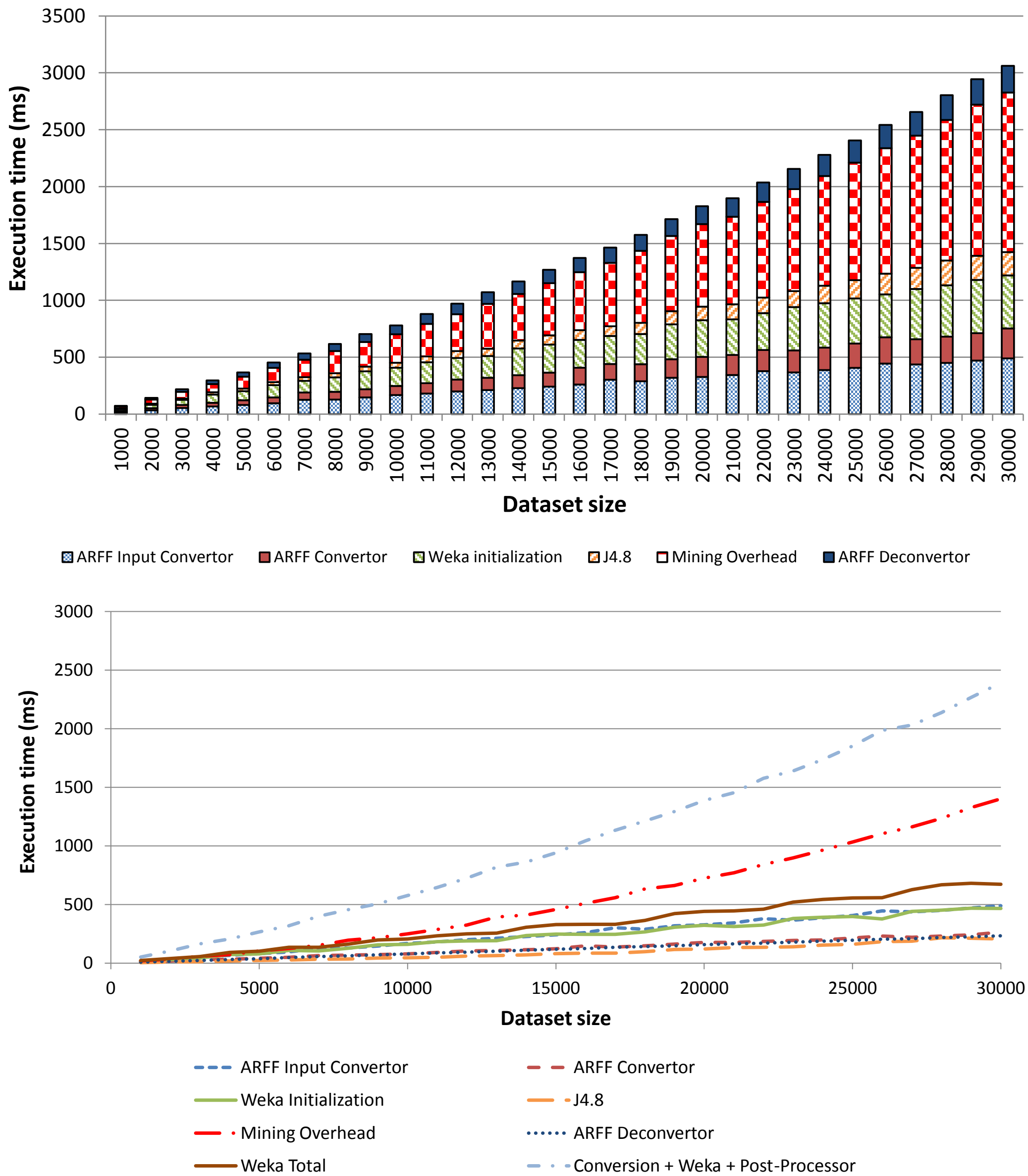


Figure 10: Execution time as a function of the dataset size for the different components of the threshold *Learning Pipeline*

amount of staff members and patients, the execution time of this module is not shown. The processing of the data by the *Iterative Tree Miner* can be split up into three parts. The *Mining Overhead* denotes the time needed to pre-process the dataset such that the different decision trees can be built as explained in Section 4.2.1. The *Weka Initialization* step consists of transforming the ARFF format to Java Objects, while *J4.8* algorithm builds the actual decision tree using WEKA. The execution times of these three steps are visualized separately.

It can be derived from Figure 10a that the execution time is exponential as a function of the size of the dataset. Figure 10b shows that this is caused by the exponentially increasing execution time of the *Mining Overhead*. The execution times of the other components are linear as a function of the amount of instances. The complexity of the J4.8 algorithm is $O(m * n^2)$ for a dataset with m instances and n attributes [27]. The number of attributes is constant in this scenario, i.e., one input attribute and one label per decision tree built for a particular priority. Consequently, the complexity reduces to $O(m)$ and thus becomes linear in the number of instances. The `ARFF Input Convertor`, `ARFF Convertor` and `ARFF Deconvertor` are also linear in the size of the dataset, as they need to (de)convert all the instances one by one. It can also be noted that the *ARFF Input Convertor* consumes more time than the *ARFF Convertor*. The first translates a `String`-based representation of the dataset, while the second receives the instances expressed in the internal data format as input. This second, structured representation can be processed more easily.

Figure 11 analyzes the execution time of the *Mining Overhead* in more

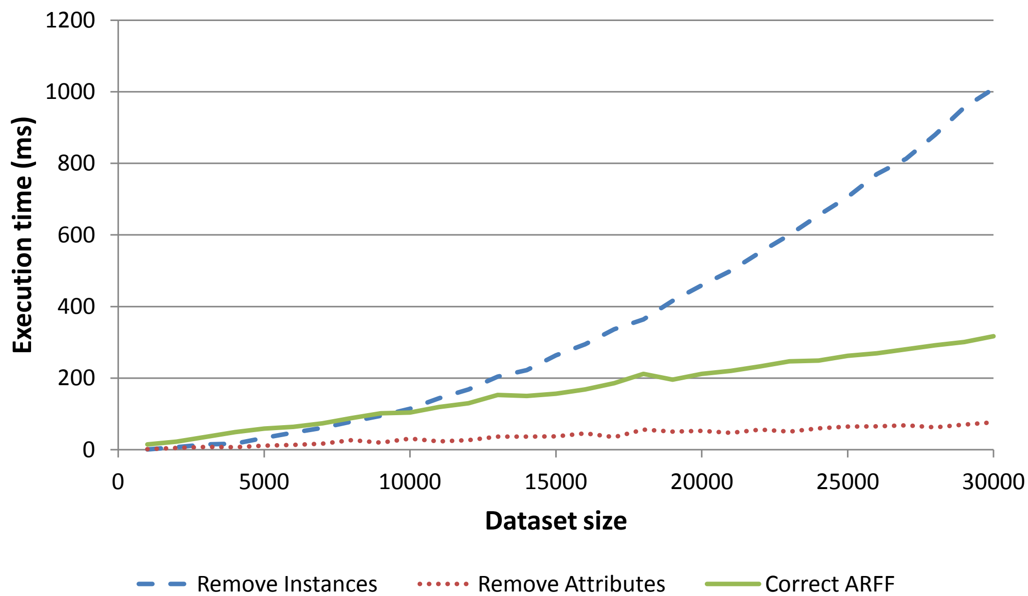


Figure 11: Execution time as a function of the dataset size for the different steps of the *Mining Overhead*

detail. As explained in Section 4.2.1, a dataset is constructed for each priority by removing the input attributes related to the other priority classes, removing all the instances labeled with a higher priority and renaming all the lower priority labels as "Other". Figure 11 indicates that most of the execution time is consumed by removing the instances. A possible solution is removing the instances before the dataset is translated to the ARFF format. The complexity of removing instances from the dataset, represented in the internal data format, is linear in the size of the dataset. However, this solution also requires that each separate dataset is translated by the *ARFF Convertor*. This also increases the execution time as there is significant overlap between the datasets and thus more instances need to be converted. Figure 12 com-

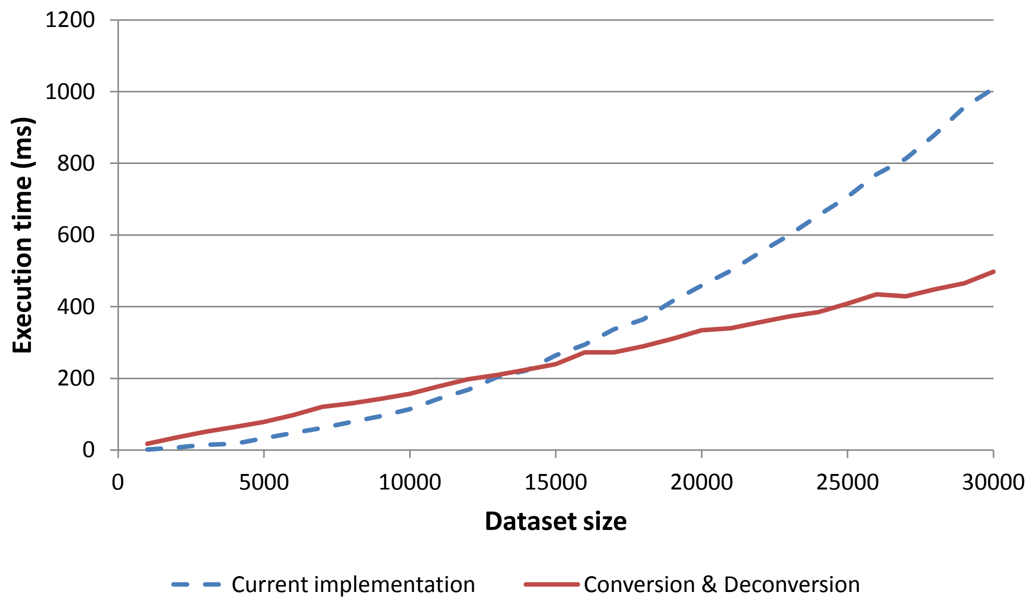
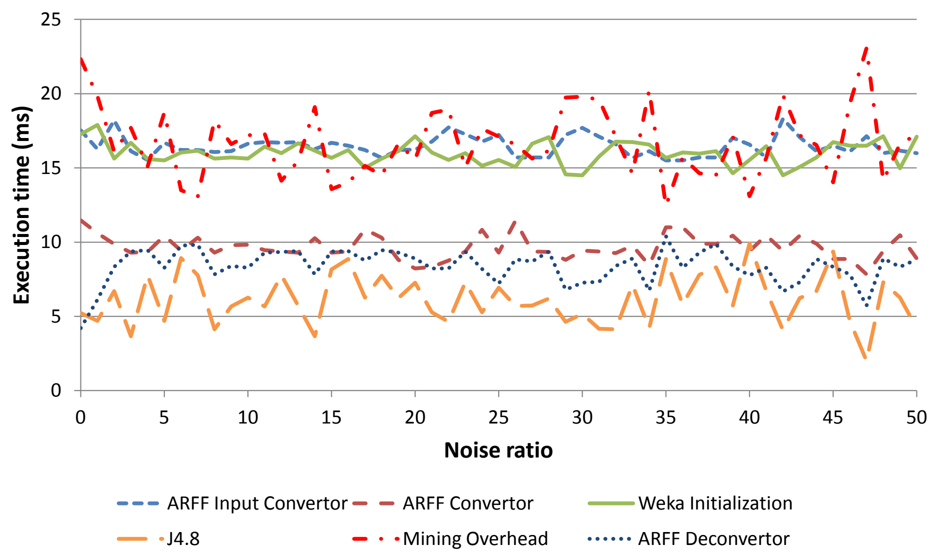


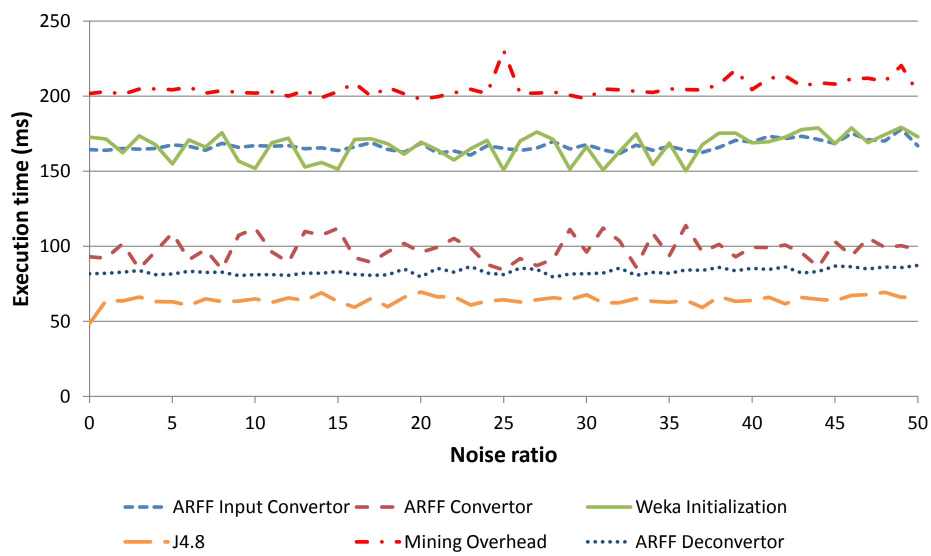
Figure 12: Compares the execution times of removing instances from the dataset as a function of the dataset size for the current and alternative implementation

pares the execution time of the current implementation for removing the instances with the additional execution time, which is needed to (de)convert the separate datasets for the alternative solution. The additional execution time of the alternative implementation is linear in the amount of instances. However, it only achieves a better performance for bigger datasets with at least 15,000 instances. As 1,050 instances were deemed to be a realistic size of the dataset, the current implementation is preferred.

Figure 13a depicts the execution time as a function of the amount of noise for the realistic dataset containing 1,050 instances. As the measured execution times are quite small, i.e., lower than 25 ms, the graphs are quite



(a) Dataset of 1,050 instances



(b) Dataset of 5,000 instances

Figure 13: Execution time as a function of the amount of noise in the dataset for the different components of the threshold *Learning Pipeline*

erratic and unpredictable. To get a clear view on the underlying trends, the performance tests were repeated for a dataset consisting of 5,000 instances. The resulting graph is visualized in Figure 13b. It can be derived that the influence of the amount of noise on the execution time is negligible. The dataset for each decision tree consists of only one input attribute and a label, which can only assume two values. Consequently, increasing the amount of noise will not have a large impact on the complexity of the constructed decision tree.

It can be concluded that a dataset with a realistic size of 1,050 instances can be processed in less than 100 ms, irrespective of the amount of noise.

6.4. Execution time of the probabilities Learning Pipeline

The execution time as a function of the size of the dataset is depicted in Figure 14. The execution times of the *Probability Calculator*, *Fluctuation Reliability Algorithm*, *Threshold Filter Algorithms* and *Priority Probability Integration Component* are not shown for the same reasons as in the previous section. The *Bayes Net Miner* consists of only two steps, namely initializing Weka and building the model using the *BayesNet* algorithm of Weka. The execution times for these two steps are visualized separately. It can be noted that the execution time is linear as a function of the size of the dataset. Figure 14b illustrates that the execution time of each of the individual components is also linear as a function of the size of the dataset. The execution times are also very small. The input conversion and initialization of Weka consume most of the execution time. Building the Bayesian network only requires a small amount of time, namely at most 20 ms for a dataset of 30,000 instances. The complexity of the Bayesian network is the

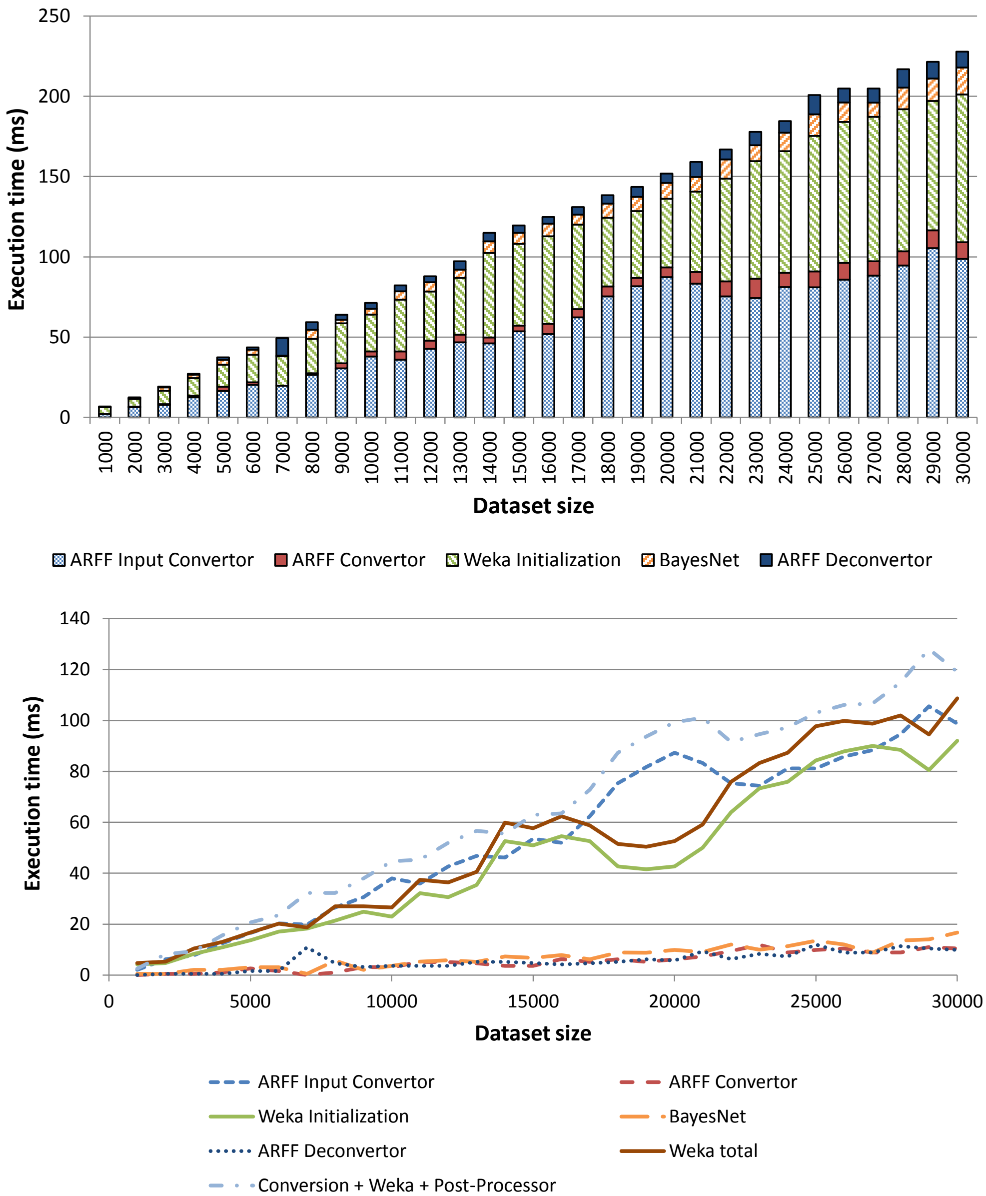


Figure 14: Execution time as a function of the dataset size for the different components of the probabilities *Learning Pipeline*

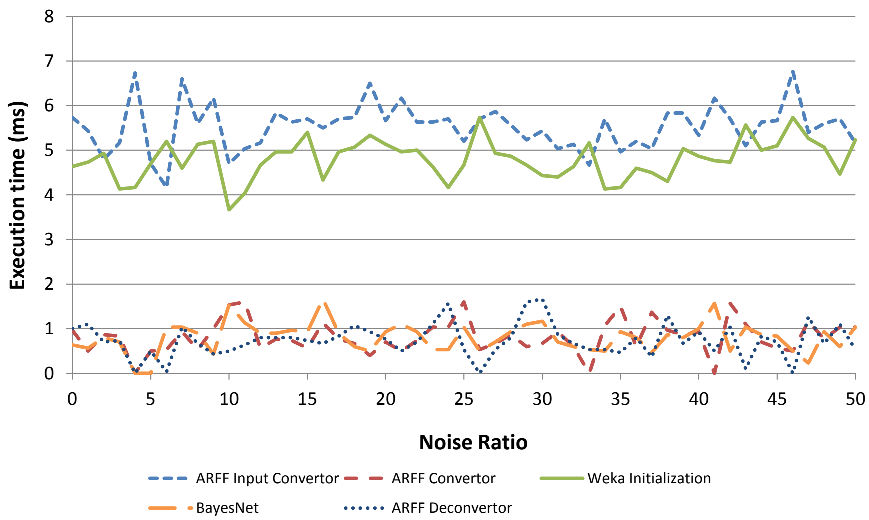
same as the J4.8 algorithm, namely $O(m * n^2)$ for a dataset with m instances and n attributes [28]. As the amount of attributes does not change in this scenario, this complexity also reduces to $O(m)$ and thus becomes linear in the number of instances. The difference in execution time between the ARFF Input Convertor and ARFF convertor was already explained in the previous section.

Figure 15a depicts the execution time as a function of the amount of noise for the realistic dataset containing 1,050 instances. Again, these execution times are too small, i.e., lower than 7 ms, to perceive a clear trend and the tests were repeated for a dataset of 5,000 instances, as shown in Figure 15b. Similar to the previous section, it can be concluded that the influence of the amount of noise on the execution time is negligible.

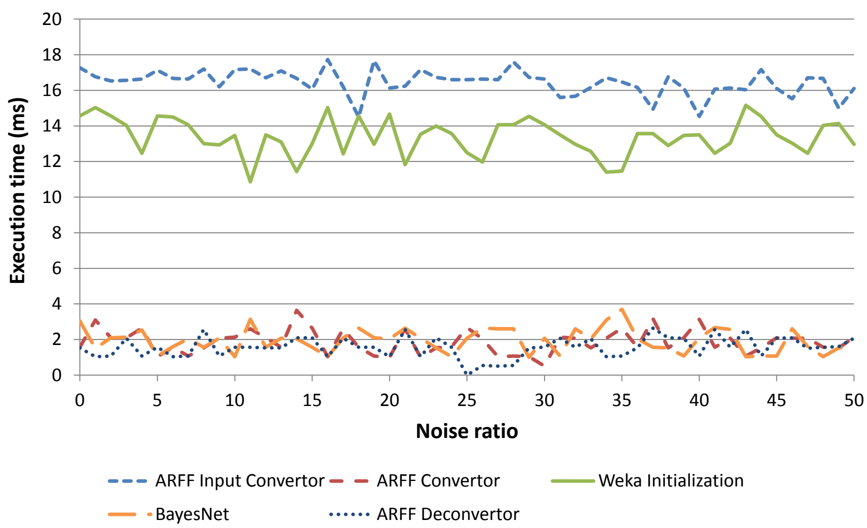
For this scenario, it can also be concluded that, irrespective of the amount of noise, the execution time is very good and negligible for datasets of a realistic size of 1,050 instances, i.e., less than 20 ms.

6.5. Memory usage

Figure 16 illustrates the memory usage of the Learning Pipeline for both scenarios as a function of the size of the dataset. The fluctuating pattern of the graphs can be explained by the memory that is consumed by the *Garbage Collector* in Java. However, trend lines can clearly be discerned. It can be noted that the memory usage is linear as a function of the amount of instances. Moreover, the total amount of consumed memory stays quite low, i.e., at most about 120 MB for the threshold *Learning Pipeline* and 25 MB for the probabilities scenario. For the realistic dataset of 1,050 instances, the memory usage is negligible for both scenarios, namely lower than 5 MB for

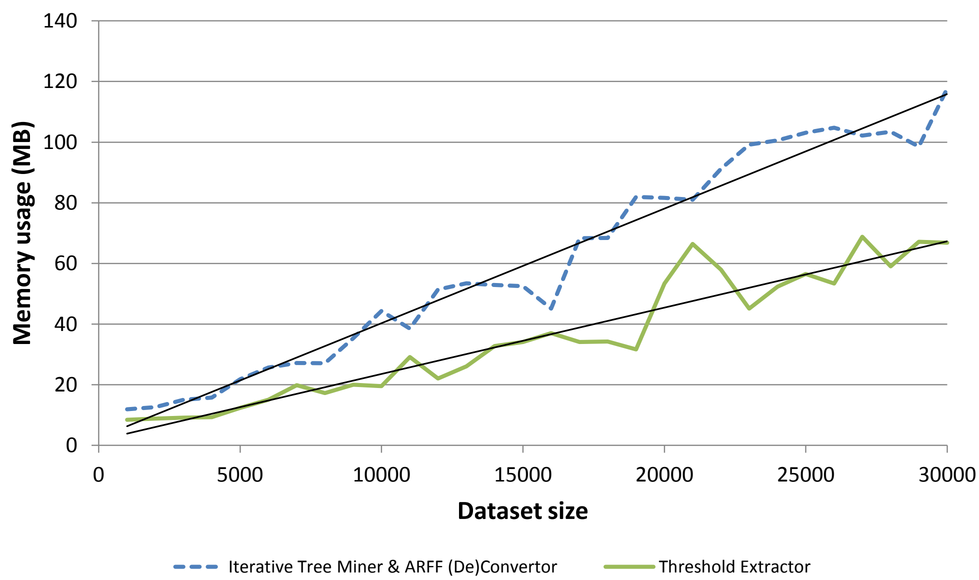


(a) Dataset of 1,050 instances



(b) Dataset of 5,000 instances

Figure 15: Execution time as a function of the amount of noise in the dataset for the different components of the probabilities *Learning Pipeline*



(a) Threshold *Learning Pipeline*

(b) Probabilities *Learning Pipeline*

Figure 16: The memory usage as a function of the size of the dataset

the probabilities *Learning Pipeline* and 20 MB for the threshold scenario. The memory usage for the threshold scenario is significantly higher. This can be explained by the different datasets that need to be created and stored to build the decision trees for each of the priorities.

7. Conclusion

This paper describes our experiences with extending the oNCS with self-learning components such that it can automatically adjust its parameters. This ensures that the application is tuned towards the needs and requirements of the caregivers and increases its adoption. Moreover, caregivers are no longer burdened with trying to define accurate parameter values for the application at development time or tweak its configuration at run-time.

The self-learning extension consists of the following steps. First, *Monitoring Algorithms* are used to monitor how the application is used with a certain context. These algorithms gather and store data. When enough data has been collected the *Data Collection Component* and *Input Convertor* retrieve the data and transform it to the internal data format used by the self-learning components. Second, the *Pre-Processor* cleans the data. *Data Mining* techniques and a *Post-Processor* are used to discover the new parameter values. The *Decision Component* associates probabilities with these learned parameter values to express their reliability. Values with a too low probability are filtered. Finally, the *Integration Component* integrates the new parameter values and their associated reliability in the oNCS.

The oNCS contains two types of parameters, namely thresholds and probabilities. An extensive evaluation was performed to assess the applicability, correctness and performance of the self-learning components for both scenarios. For the thresholds, it was shown that correct results with a relative error of less than 5% are obtained when the dataset contains at least 500 instances, i.e., calls, and the noise ratio is less than 20%. For the probabilities, it was deduced that for a realistic dataset of 1,050 instances correct results were obtained. Both the threshold and probability parameters are learned very efficiently as the components require at most 100 ms execution time and 20 MB memory for a realistic dataset of 1,050 instances, irrespective of the amount of noise in this dataset.

Future work will mainly focus on evaluating a prototype of the self-learning oNCS in a real-life setting.

Acknowledgment

F. Ongenae and M. Claeys would like to thank the IWT for financial support through their Ph.D. grant.

[1] World Health Organization (WHO), Health topics: Ageing, `http://www.who.int/topics/ageing/en/` (2013).

[2] I. Meyer, S. Müller, L. Kubitschke1, A. Dobrev, R. Hammerschmidt, W. B. Korte1, T. Hüsing, T. van Kleef, S. Otto, J. Heywood, M. Wrede, eCare as a way of coping with an ageing population today and tomorrow. The eCare benchmarking study, Tech. rep., European Commission, Directorate General Information Society and Media, Brussels, `http://ec.europa.eu/information_society/newsroom/cf/itemdetail.cfm?item_id=10182` (April 12 2013).

[3] World Health Organization (WHO), The world health report 2006 - working together for health, http://www.who.int/whr/2006/en/ (2006).

[4] E. Percy, Healthcare challenges and trends, Tech. rep., Logica (2012).

[5] C. Orwat, A. Graefe, T. Faulwasser, Towards pervasive computing in health care - a literature review, BMC Medical Informatics and Decsion Making 8 (26) (2008) 18.

[6] J. Li, A. Talaei-Khoei, H. Seale, P. Ray, C. R. MacIntyre, Health care provider adoption of ehealth: Systematic literature review, Interactive Journal of Medical Research 2 (1) (2013) e7.

[7] J. H. Jahnke, Y. Bychkov, D. Dahlem, L. Kawasme, Context-aware information services for health care, in: Proc. of the Workshop on Modeling and Retrieval of Context, 2004, pp. 73–84.

[8] J. Criel, L. Claeys, A transdisciplinary study design on context-aware applications and environments. A critical view on user participation within calm computing, Observatorio 2 (2) (2008) 57–77.

[9] F. Ongenae, D. Myny, T. Dhaene, T. Defloor, D. Van Goubergen, P. Verhoeve, J. Decruyenaere, F. De Turck, An ontology-based nurse call management system (oNCS) with probabilistic priority assessment, BMC Health Services Research 11 (2011) 26.

[10] F. Ongenae, M. Claeys, T. Dupont, W. Kerckhove, P. Verhoeve, T. Dhaene, F. De Turck, A probabilistic ontology-based platform for self-

learning context-aware healthcare applications, Expert Systems with Applications 40 (18) (2013) 76297646.

[11] F. Ongenae, L. Bleumers, N. Sulmon, M. Verstraete, A. Jacobs, M. Van Gils, A. Ackaert, S. De Zutter, P. Verhoeve, F. De Turck, Participatory design of a continuous care ontology: Towards a user-driven ontology engineering methodology, in: J. Filipe, J. L. G. Dietz (Eds.), Proceedings of the International Conference on Knowledge Engineering and Ontology Development (KEOD), ScitePress Digital Library;, Paris, France, 2011, pp. 81–90.

[12] T. Gruber, A translation approach to portable ontology specifications, Knowledge Acquisition 5 (2) (1993) 199–220.

[13] M. Strobbe, O. V. Laere, F. Ongenae, S. Dauwe, B. Dhoedt, F. D. Turck, P. Demeester, K. Luyten, Novel applications integrate location and context information, IEEE PERVASIVE COMPUTING 11 (2) (2012) 64–73.

[14] S. Haiges, A step by step introduction to OSGi programming based on the open source Knopflerfish OSGi framework, Tech. rep. (October 2004).

[15] D. L. McGuinness, F. v. Harmelen, OWL Web Ontology Language overview, Tech. Rep. REC-owl-features-20040210, World Wide Web Consortium, `http://www.w3.org/TR/owl-features/` (February 10 2004).

[16] P. Klinov, Pronto: A non-monotonic probabilistic description logic reasoner, in: Proceedings of the 5th European Semantic Web Conference, Tenerife, Spain, 2008, pp. 822–826.

[17] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, Jena: implementing the semantic web recommendations, in: Proceedings of the 13th international conference on World Wide Web, Alternate track papers & posters, New York, NY, USA, 2004, pp. 74–83.

[18] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 2nd Edition, Addison-Wesley Professional, 2003.

[19] I. H. Witten, E. Frank, M. Hall, Data Mining: Practical Machine Learning Tools and Techniques, 3rd Edition, Morgan-Kaufmann, 2011.

[20] E. Prud'hommeaux, A. Seaborne, SPARQL Query Language for RDF, W3C Recommendation REC-rdf-sparql-query-20080115, `http://www.w3.org/TR/rdf-sparql-query/` (January 15 2008).

[21] S. B. Kotsiantis, Decision trees: a recent overview, Artificial Intelligence Review 39 (4) (2013) 261–283.

[22] R. E. Neapolitan, Learning Bayesian Networks, Prentice-Hall, San Francisco, CA, USA, 2003.

[23] S. B. Kotsiantis, Supervised machine learning: A review of classification techniques, Informatica 31 (3) (2007) 249–268.

[24] J. R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann, San Francisco, CA, USA, 1993.

[25] T. Lukasiewicz, Probabilistic description logics for the Semantic Web, Tech. rep., Technical University of Wien, Institute for Information Systems, Wien, Austria (2007).

[26] Ghent university hospital, `http://www.healthcarebelgium.com/index.php?id=uzgent` (2013).

[27] J. Su, H. Zhang, A fast decision tree learning algorithm, in: Proceedings of the 21st National Conference on Artificial Intelligence, Boston, MA, USA, 2006, pp. 500–505.

[28] J. Su, H. Zhang, Full Bayesian network classifiers, in: Proceedings of the 23rd International Conference on Machine Learning (ICML), Pittsburgh, PA, USA, 2006, pp. 897–904.