POLITECNICO
MILANO 1863

Department of Industrial and Information Engineering
Master of Science in Computer Science and Engineering

Automated Learning in Complex Event Processing with Decision Tree Classifiers

Supervisors:
Prof. Gianpaolo Cugola
Prof. Alessandro Margara
Prof. Matteo Matteucci

Thesis Dissertation of:
Samuele Di Leva
824219

Academic Year 2017-2018

Sommario

In ambito Complex Event Processing, regole che descrivono pattern di eventi primitivi permettono di individuare situazioni di interesse più complesse, come pericoli o possibili opportunità, a partire da uno stream di eventi in real-time.

Un approccio alternativo alla definizione manuale di regole per sistemi Complex Event Processing è l'apprendimento automatico dei pattern che conducono a eventi complessi. Nel presente lavoro, utilizzeremo tecniche general-purpose di Machine Learning – ed in particolare l'algoritmo C5.0, un tool off-the-shelf basato su alberi di decisione – per affrontare le sfide poste dal contesto Complex Event Processing.

Nel corso di questa tesi, presenteremo il sistema sviluppato, che permette a C5.0 di individuare occorrenze di eventi complessi all'interno di uno stream di dati, testando la sua accuratezza e valutando il suo grado di applicabilità in una varietà di situazioni distinte e caratterizzate da diversi parametri.

Abstract

In Complex Event Processing, rules describing patterns of primitive events allow to detect complex situations of interest, such as threats or opportunities, from a stream of data and in real-time.

An alternative approach to the manual definition of Complex Event Processing rules is the automated learning of the patterns leading to composite events. In the present work, we employ general-purpose Machine Learning techniques — and in particular C5.0, a supervised learning off-the-shelf tool based on decision trees — to deal with the challenging scenarios proposed by Complex Event Processing.

In the course of this thesis, we present the system developed that enables C5.0 to detect occurrences of composite events within streams of event data, and we test its accuracy and applicability in a variety of situations.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Complex Event Processing (CEP) is an Event-Driven Architecture that analyzes and reacts on-the-fly to streams of data coming from multiple sources. The objective of a CEP system is to detect situations of interest called *complex events*, which are expressed as patterns of low-level *primitive events*. Such patterns, capturing the causality link between primitive and complex events, are called *rules* and play a central role in any CEP architecture.

Traditionally, rules are manually formulated and provided to the system thanks to the knowledge and experience of *domain experts*, but a different and challenging approach consists in trying to *learn* them from the data stream itself or, in other words, based on the primitive events observed in the past. However, to this moment the research conducted on the topic is still very limited and no commercial system supporting automated CEP rule learning features is currently available.

In the present work, we aim at finding out if it is possible to employ traditional and well-known Machine Learning (ML) techniques to automatically predict future occurrences of complex events without prior knowledge of the underlying patterns. In order to answer this question, we concentrated our efforts in the direction of *decision tree-based* learning algorithms and we eventually managed to adapt C5.0 - the successor of the popular C4.5 algorithm - to work inside a CEP context, allowing it to build classifiers used to label *traces* of events, discriminating the ones which lead to composite events from the ones which do not.

In the following chapters, we present the solution developed, describing it in detail and pointing out its strengths as well as its limitations.

In particular, in [Chapter 1] we revise some background theory on both CEP and ML, introducing all the concepts required to understand the rest of the work. In [Chapter 2] the thesis problem is formally defined, whereas an high-level overview of our approach at solving it can be found in [Chapter 3]. The implementation of the solution is detailed in [Chapter 4] where we explain the structure and the function of each component of the architecture developed. Finally, in [Chapter 5] we evaluate the accuracy of the learning environment implemented, commenting the results collected and drawing some conclusions on the performance of the system as a function of different parameters, whereas [Chapter 6] revises some related work in the field of automated CEP rule learning and pinpoints many open issues which emerged at the end of the work.

Chapter 1

Background Theory

In this chapter we explain all the background concepts required to understand the thesis problem. [Section 1.1] presents a general overview on CEP, [Section 1.2] introduces *CEP languages* and Complex Event Recognition with reference to the state of the art in the field, whereas the event model adopted is illustrated in [Section 1.3]. Lastly, [Section 1.4] revises some Machine Learning background.

1.1 Overview on Complex Event Processing

During the last two decades, the interest for CEP technologies has been constantly increasing and the paradigm attracted the attention of both academia and industry. In fact, with the Big Data explosion, the increasing diffusion of distributed applications, along with greater network capabilities and the availability of more computing power at lower costs, today the information is continuously flowing and coming from a potentially unlimited number of sources in massive quantities.

Therefore, a company business is more often than not influenced by factors (the *events*) that originate from outside its own environment, such as from the Internet of Things – real-time sensor networks that can measure and report on a multitude of situations – and traditional computer science approaches, like DBMSs (in which data is mainly static and slowly changing over time) as well as point-to-point, batch-oriented, request-reply based solutions (e.g., client-server) are no more suited for *reactive*

applications dealing with large amounts of data that has to be processed in real time (a concept known as *real time situational awarness*).

Despite being a novel technology, CEP already has a broad variety of applicative scenarios. Some examples are given by credit card fraud detection systems, that analyze a series of transactions with the aim of detecting frauds, software for automated financial trading (in which a stream of stock trade data is collected and analyzed in order to spot trends and opportunities), traffic flow supervision with variable toll charging and speed control, environmental monitoring systems, applications in the banking field providing support for loan and mortages, Intrusion Detection Systems (in which patterns of network traffic data can be used to prevent threats – for example, DDoS attacks), software for the automated distribution of products, applications in the automotive field and many more.

An Event-Driven Architecture, like CEP, is virtually useful in any scenario requiring real-time processing of data and the production of fast reactions to the occurrences of events; in this way, it can help to avoid extraordinary conditions that represent a threat (like in fraud detection systems or environmental monitoring scenarios), but it can also benefit industries by automating some business processes thus cutting expenses, helping to avoid worst-case scenarios or even supporting the management in decision making. CEP architectures are intrinsically better equipped to handle this sort of situations because they are specifically designed to process information as a flow, according to some processing rules; in other words, they are *context-aware*, i.e. "smart" enough to sense and react to the environment on-the-fly in order to accomplish some task.

For sure, Event-Driven contexts are very challenging because they require both high throughput (1.000-100.000 events/s) and low latency (ms-seconds); in addition, the control flow of the program does not depend on the values of system state variables (as it was the case with traditional technologies) but on externally generated *events*, which are substantially different from a simple user input because not only they are asynchronous but their timing falls outside the control of the program. However, the concept of "events" is precisely what makes CEP systems capable of facing such challenges, enabling them to work with difficult scenarios that need "just in time" reactions.

In a very informal way, an *event* can be defined as an action or a change of state: for example, a door changing its state from "open" to "closed",

the temperature going from "low" to "high" or a simple mouse click in a web application. An event is also required to be either something observable by an IT component (e.g., the humidity or pressure read by a sensor), or triggered by it (e.g. the result of a computation, a timer going off, ...).

CEP systems collect and process a stream of event data, i.e. a time-ordered sequence of events, with the aim of detecting higher-level *complex*, or *composite events*, which can be viewed as a pattern of low-level raw data (*atomic*, or *primitive events*). Such patterns are captured by *CEP rules*, which express the occurrence of a complex event as a combination of primitive events subject to a number of *constraints* (or *operators*).

From an architectural point of view, it is possible to identify three basic components of CEP systems:

- the *sources* (also called *Event Producers*) which observe primitive events, send event notifications and transmit elementary data,
- the *sinks* (or *Event Consumers*), that are in charge of reacting to the occurrence of a complex event, and
- the *CEP Engine*, which collects data from the sources and – if it finds a known pattern within the data (i.e., there is a match of a rule) – it notifies all the sinks that were waiting on that complex event to happen.

The overall, high-level architecture of a CEP system is showed in [Figure 1.1]. It is possible to observe how the CEP Engine collects primitive events, and then – by accessing a proper *Rule Base* – it notifies possible occurrences of complex events to sinks.

In order to further clarify the basic concepts expressed so far, we provide a simple example of events and rules. For instance, a primitive event could be the presence of smoke inside a room detected by a sensor, or a temperature reading. A complex event is a combination of primitive events: for example, the complex event `Fire` can be expressed as a conjunction of the atomic events `Smoke` and `HighTemperature`. It is also useful to include a time-frame (*window*) within which the atomic events need to happen in order to trigger the complex event. The following rule, written in almost natural language, summarizes what was said above and catches the occurrences of the event `Fire` :

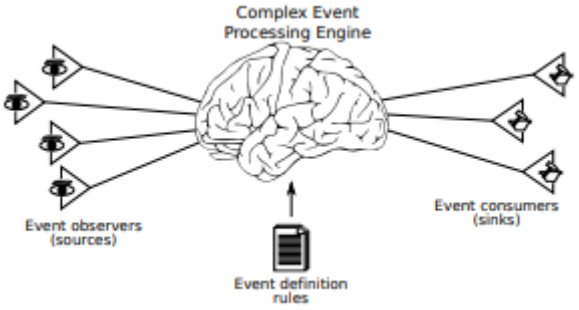

Figure 1.1: The high-level view of a CEP system. Image taken from [14], p.1

```
within 5 minutes: if (Smoke and HighTemperature) then Fire.
```

In this case the rule is very simple and can be formulated just by common sense; however, in most real-world scenarios the problem of correctly identifying the relationships that link primitive events to complex ones is actually much more challenging. Furthermore, not only it is difficult to spot all the patterns hidden in the data, but the patterns themselves may be extremely complicated and definitely hard to express in a human-friendly form.

About this, the usual approach adopted in the CEP field relies on the work of professionals called *domain experts*, who are in charge of suggesting proper CEP rules based on their knowledge of the applicative domain of interest. However, this traditional method works only for relatively simple and well-known domains; in some cases, it becomes very difficult to catch all the hidden patterns that lead to the occurrence of a composite event and even domain experts cannot accurately predict the vast majority of the existing patterns.

Another, more advanced and potentially much more efficient way to detect complex events is to *automatically derive* the needed rules starting from some observed event data. This is the point at which Machine Learning instruments come at hand and could be useful to enhance current CEP technologies. In particular, the question that naturally arises – and that is central in the present work – is whether or not it could be feasible to employ existing general-purpose ML techniques to learn event patterns, instead of relying on domain experts or having to develop ad-hoc learning algorithms to do the job.

Before proceeding to illustrate our tries in employing such ML techniques, it is useful to clear some more semantics and terminology with regard to the CEP background. In the next section, events, operators and rules are going to be formally defined in order to give the reader a better overview on CEP, removing any ambiguity surrounding such fundamental concepts.

1.2 CEP Languages and Models for Complex Event Recognition

The problem of choosing a proper model for events, as well as a formal *rule definition language*, is crucial for the development of CEP engines, since it influences both the resulting processing algorithms and the overall system architecture.

Complex event recognition (CER) refers to the problem of detecting events in big data streams; during the last two decades, many models were proposed to represent events and patterns, as well as different strategies and methods of detection. Traditionally, as pinpointed in [10], the main techniques employed in CER can be split into two broad categories, based on the processing strategies and event specification languages adopted: *automata-based* techniques and *logic-based* techniques.

Automata-based are the most widely employed solutions for pattern matching, being a stream of events conceptually similar to a string of characters to be recognized either by a Deterministic Finite Automaton (DFA) or Non-Deterministic Finite Automaton (NFA). Such solutions aim at working under different degrees of uncertainty, assuming a probabilistic input stream due to noise and missing information and providing some kind of (approximate) inference. Plenty of pattern languages, translating an input event stream into regular expressions recognized by an automaton, have been proposed in the course of the years, starting from early approaches like the Lahar system [31] or [33]. Noticeable are also SASE [17] and its extension SASE+ [9], which exploits a buffered NFA (NFA^b model) to detect queries; such systems have been optimized to deal with expensive queries very common in real-world scenarios [37]. Other similar proposals include CEP2U [16] and the PADUA system [26].

On the other hand, *logic-based* strategies express complex events in terms of logic formulas and rely on logical inference to perform CER. Some examples are given by Markov Logic Networks (MLN), which are

undirected probabilistic graphic models combining Markov Networks with first-order logic [35, 18, 32, 25]. Other examples are given by ILED [34], which adopts a run-time version of the Event Calculus formalism (RTEC) [20], and [35] exploiting Allen's Interval Algebra [11] built-in predicates to represent temporal relations between events.

1.3 An Abstract Event Specification Language

In addition to automata-based and logic-based approaches, a totally different CER technique stems from traditional relational databases and it employs SQL-like languages, with the addition of new ad-hoc operators that allow to process input data streams on-the-fly to produce output streams. These models are known as Data Stream Management Systems (DSMSs) and are opposed to traditional DBMSs, in which data is static and indexed. However, as emphasized in [14, 23], such languages usually fail to capture ordering relationships and show severe limitations in recognizing complex patterns among events – or they do it in an unnatural and difficult to support manner (e.g. Esper [3], Oracle CEP [7]).

Finally, many languages have been *explicitly designed* to deal with event notifications and not generic streams of data like in the case of DSMSs. An example is given by T-Rex [14], a CEP middleware offering a language, TESLA, which is built around events and deals with them in an easy way while still providing an efficient processing of rules.

In the present work we refer to the latter kind of models because it naturally simplifies the task of expressing complex events as patterns of primitive ones and it is better suited for the task of adapting ML techniques to CEP. In particular, we adopt an *abstract model* close to the one presented in [23], purposely avoiding any specific CEP language in order to preserve generality. Such model will be detailed in the next subsections.

1.3.1 Events

Throughout the course of this work, we assume that each event is characterized by a type, a set of attributes and a timestamp.

- The *timestamp* records the exact time in which the event took place; in fact, we assume events to happen instantaneously at some point in time (although this is not necessarily the case in every model).

- The *type* is a meaningful unique string describing the event in exam (e.g, "Fire", "Smoke", "ClickOnBanner", "StockPriceUp", ...).
- *Attributes* can just be viewed as variables: they have a *name* and a *type* (not to be confused with *event types*) that can be a character, string, integer, long, float, double and so on. Like variables, attributes may assume different values taken from a given *domain*, which depends on their type. The role of event attributes is to provide important additional information about the event of interest (for example, if the event notification is a temperature reading, we are also interested in knowing the actual temperature value that has been recorded, because the notification alone is not useful).

  The name, class, number and ordering of the attributes entirely depends upon the event *type*; therefore, different event types lead to different sets of attributes (and, on the other hand, two or more events of the same type are characterized by the same attribute structure).

Taking the fire scenario previously introduced, an example of event – expressed according to our model – could be the following:

```
Temp@10(area="A1", value=24.5)
```

Here, `Temp` is the event type, `10` is the timestamp and `area` and `value` are two attributes: a string which identifies the area of reading and the actual temperature value (a float).

Some events may be characterized by no attributes at all; for example, the event `Smoke@17()` just records the occourence of something (i.e., the presence of smoke) at some point in time, and does not need to include any more information. However, the presence of a timestamp associated with an event is always required.

The model described above is valid both for primitive and composite events; however, because of their nature – and since we interpret them as a pattern of primitive events – composite events are usually devoid of attributes, since they just denote something of interest happening at some point in time (e.g., `Fire()`). Throughout the course of this work, we will thus consider composite events as a "special" kind of events without attributes (but this is just an assumption and not necessarily the case every time).

1.3.2 Operators

CEP operators (or *CEP constraints*) are the basic building blocks that allow to construct complex patterns composed by primitive events.

According to literature, as observed in [15], it is possible to recognize at least 5 fundamental CEP operators: selection, conjunction, sequence, window, and negation.

- The *selection* operator filters events according to the value of their attributes (e.g., `Temp(value>50)` selects events of type `Temp` if and only if their attribute `value` is greater than 50).
- *Sequences* regard the order of arrival of events and they are meaningful whenever events are required to take place in a certain order, according to their timestamps. For instance, `Temp -> Smoke` is a sequence constraint implying that the event `Temp` must always precede the event `Smoke`.
- *Conjunction* is a logical operator analogous to the logical AND; a conjunction of events is satisfied if and only if all the events in the conjunction have been detected. Here, no ordering relationship among the events is considered – what matters is only whether they are all detected or not (in any order). A simple example of conjunction is given by the constraint `Smoke() AND Temp(value>50)`, which also includes a selection operator on the event `Temp`.
- *Windows* determine which portions of the input flow have to be considered while evaluating other operators, i.e., they define the maximum timeframe of a given pattern. *Sliding windows* are the most common type of windows; they are characterized by a fixed size and their lower and upper bounds advance whenever new events enter the system. In other words, the timestamp advances by one unit at every iteration, thus selecting one new event every time while removing a previously included one.

  Windows can also be defined in terms of timestamps (e.g., a window of 10 seconds contains all the events recorded within a time interval of 10 seconds)
- Lastly, the *negation* operator selects all the events that are have *not* been detected; in this case what matters is not the presence, but the absence of such events.

- Other minor but worth mentioning operators are *parameterization* and *aggregation*; the first one involves constraints on the attributes of different event types (e.g., `Temp.area = Smoke.area`), whereas the second one filters events according to some aggregated function (typically, minimum, maximum, average, count and sum).

To understand how it is possible to exploit the operators presented above in order to build complex patterns defining composite events, just consider the following example:

```
within 5m.  Smoke() and Temp(value>50) and not Rain(mm>2)
where       Temp -> Smoke
```

The pattern is composed by a negation (`not Rain`), a conjunction (`Smoke and Temp and not Rain`), two selections (`Temp(value>50)`, `Rain(mm>2)`) and a sequence constraint (`Temp` $\rightarrow$ `Smoke`).

It is possible to express the same pattern with a rule written in any specific CEP language (e.g., TESLA) having an equivalent degree of *expressiveness*. The expressiveness of different languages can be evaluated with regard to the type and number of supported operators: some specific languages may employ variants of the operators described above or include additional operators; in this work, we choose to employ once again the same simple and intuitive syntax showed in [23], including almost all the original operators.

Lastly, it is not always necessary to exploit a *rule* in order to detect composite events: for example, in our case we employed *decision trees* instead of rules. To better understand the different method of detection proposed, it is first required to study the capabilities offered by the Machine Leaning discipline; therefore, we present some ML background in the next section, which is necessary to understand why the ML tools ultimately chosen constitute a feasible option for CEP learning.

1.4 Machine Learning Background

1.4.1 Introduction to Machine Learning

Machine Learning is the computer science subfield that aims at giving machines the ability to learn without being explicitly programmed. In other words, computers are not instructed about how to execute a specific task, but rather about how to learn the task themselves starting from some experience which is provided to them. As already known, we are concerned with ML algorithms to automatically find patterns in data; for this reason, we will focus on *supervised learning* methods, which aim at predicting the value of a target variable starting from a set of data points that have a known outcome.

There exist two kinds of supervised learning techniques, which in turn depend on the type of the outcome: if it is *continuous* (or numerical), we talk about *regression*, while if it is a *category* (i.e., a discrete value, or *label*) then we are faced with a *classification* problem. For the purposes of the current work we will explore the latter kind of supervised learning, in which a set of *labeled instances* is provided as input and the output is a model that will be used to predict the outcome of new unlabeled instances.

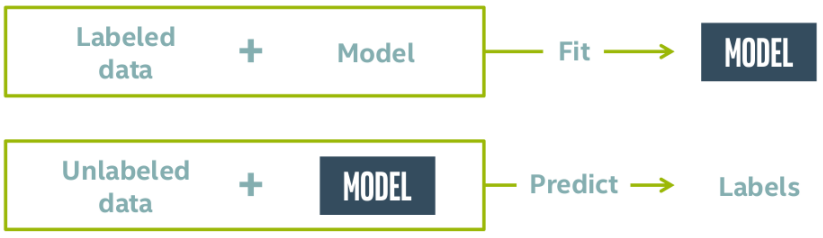

Figure 1.2: Supervised learning overview. Image source: [4].

Instances are individual, independent examples of a concept to be learned; different aspects of an instance are measured by *attributes*, which are a fixed predefined set of features characterized by a *type*. According to the analysis performed in [36], attributes can belong to four main categories: nominal, ordinal, interval and ratio.

- *Nominal* attributes assume values which are just distinct symbols taken from a finite set (e.g., `"A1", "A2", "A3"` for the attribute `Area`) without any ordering or distance relation implied among them.
- *Ordinal* attributes impose an order on values but no distance (e.g., `hot > mild > cool` for the attribute `Temperature`).
- *Interval* quantities are ordered and measured in fixed and equal units (e.g., `Temperature` expressed in degrees Fahrenheit): here, the difference of two values makes sense, but sum or product do not.
- Lastly, *ratio* quantities define a zero point for the measurement and hence they are treated as real numbers, supporting all mathematical operations.

However, a simpler categorization of attributes which is largely employed in many algorithms (and it is enough for the scope of the present work) simply divide them into two broader classes: *discrete* (i.e., nominal) and *numerical* or *continuous*, with the latter ones being just like ordinal attributes but also including mathematical continuity.

A value for an attribute of a given instance may be *missing* (because it may be unknown or unrecorded, e.g., due to malfunctioning equipment or a measurement not possible), in which case it must be treated in a special way – usually, by employing a separate attribute value indicated with the character `?`.

As already mentioned at the beginning of this section, in order to allow learning we need to gather some *experience* beforehand. Such experience usually comes in the form of a *training dataset*, which, in the case of supervised learning methods, is a list of already correctly labeled instances and includes all the values assumed by the attributes of the instances.

It follows, in [Table 1.1] below, an example of a dataset containing only nominal attributes. As it can be easily noticed, the classification attribute is "Recommended lenses" and every instance is labeled according to this attribute (credits for the public dataset go to [22]).

Generally speaking, the supervised learning process is composed by two fundamental steps: *learning* and *testing*. We have already seen that the first one aims at learning a model using training data (where the input scheme is provided with actual outcomes), whereas the second one tests the previously built model in order to assess its accuracy. In fact, a

Age	Spectacle prescription	Astigmatism	Tear production rate	Recommended lenses
Young	Myope	No	Reduced	None
Young	Myope	No	Normal	Soft
Young	Myope	Yes	Reduced	None
Young	Myope	Yes	Normal	Hard
Young	Hypermetrope	No	Reduced	None
Young	Hypermetrope	No	Normal	Soft
Young	Hypermetrope	Yes	Reduced	None
Young	Hypermetrope	Yes	Normal	Hard
Pre-presbyopic	Myope	No	Reduced	None
Pre-presbyopic	Myope	No	Normal	Soft
Pre-presbyopic	Myope	Yes	Reduced	None
Pre-presbyopic	Myope	No	Normal	Hard

Table 1.1: First 12 entries of the contact lenses dataset.

model not only summarizes what we know (i.e., the training dataset) but hopefully it will also be able to correctly classify new cases. Therefore, when building classification models one should have access both to training data (used to build the model) and *test data* (or *evaluation data*) to verify how well the model actually performs.

Such evaluation usually exploits two fundamental metrics: *precision* and *recall*, which are the reference point for all the experimental evaluations in the present work. In order to understand them, we should first define the concepts of *true positives* (TP), *true negatives* (TN), *false positives* (FP) and *false negatives* (FN).

All of them are relative to a particular class and it is possible to calculate the number of TP, FP, TN, FN for each existing class. In particular, TP are instances belonging to a class that are correctly identified with the label of that class, TN are instances correctly identified as not belonging to a given class, FP are instances not belonging to a class but incorrectly classified with that class label, and finally FN are instances belonging to the class of interest but incorrectly classified as they were not.

Given the definitions above, *precision* is expressed as:

$$P = \frac{TP}{TP + FP}$$

In other words, it specifies the fraction of relevant instances among the retrieved instances.

On the other hand, *recall* is defined as:

$$R = \frac{TP}{TP + FN}$$

Which is the fraction of relevant instances that have been retrieved over the total amount of relevant instances.

There exist other metrics measuring the accuracy of a test besides precision and recall; for example, we often employed the *F-score* or F_1-*measure*, which is the harmonic average of precision and recall:

$$F_1 = 2 * \frac{P * R}{P + R}$$

All the metrics mentioned above reach their best value at 1 (where we have a perfect precision/recall/F-score) and worst at 0; in order to obtain a perfect F-score, both precision and recall must be perfect as well.

In the next subsections, we describe many supervised learning strategies in greater detail.

1.4.2 Decision Tree Learning

Decision tree learning follows a "divide-and-conquer" approach and it adopts a decision tree structure as predictive model with the aim of deciding the value of a target variable by exploiting a number of input variables. In this work we refer to *classification trees* because the predicted outcome is a class (which is not necessarily the case with generic decision trees). In short, a classification tree is just a flowchart-like structure characterized by the following elements:

- *internal nodes* (also called *decision nodes*), which represent a test on a single attribute value (e.g. `Temperature>25`);
- *branches*, that follow from the outcomes of a test (namely, a corresponding subtree is associated with each branch);
- *leaves*, containing class labels.

When testing nodes, attribute values are usually compared to constants, but it is also possible to compare the values of two attributes or to use a function of one or more attributes. On the other hand, leaves assign a classification (or a probability distribution) on instances.

In order to classify an unknown instance, the instance itself is routed down the tree, checking all the conditions found in the path until a leaf is found and the instance is classified. If the test yielded in a node returns true, the case will proceed down the left branch, otherwise it will follow the right branch and the process is repeated until a leaf is found (assuming to employ *binary splits* and therefore to only have two branches).

With nominal attributes, the number or children of an internal node is usually equal to the the number of different values (hence the attribute will not get tested more than once); on the other hand, with numeric attributes the test is about whether the given value is greater or lesser than a constant (so the attribute is likely to get tested several times).

The following [Figure 1.3] shows a simple example of a classification tree, which refers to the dataset already introduced and presented in [Table 1.1], representing a possible solution for the problem of prescribing contact lenses to patients. The tree has been obtained by running the J48 algorithm of the Weka Data Mining Tool 3.9.1, which can be downloaded for free under GPL license at [8].

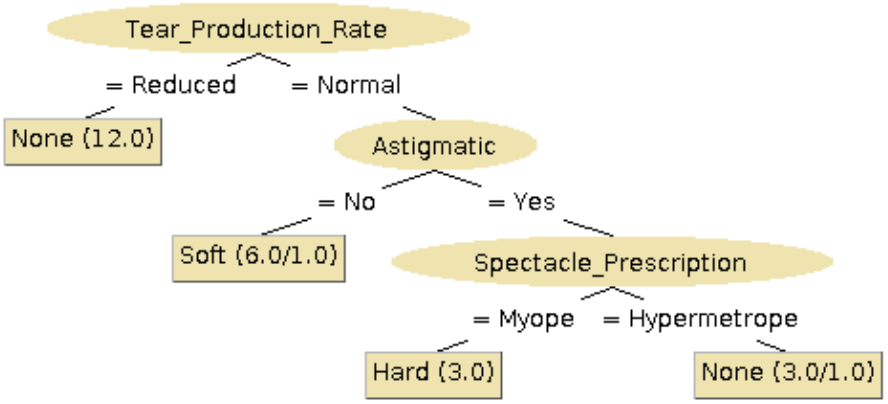

Figure 1.3: Decision tree for the contact lenses dataset

Here, the `Recommended_Lenses` are classified with three labels (`"None"`, `"Soft"` or `"Hard"`), whereas the internal nodes split on nominal attributes (`Tear_Production_Rate`, `Astigmatism`, `Spectacle_Prescription`).

The entire tree structure can be linearized into a series of *classification rules*, which are usually put in OR with each other. A rule is composed by an antecedent containing a series of tests (the ones of the internal nodes) logically ANDed together, and by a consequent which is a class label.

To convert a classification tree into a set of rules, we just extract one rule for each leaf; the antecedent will contain a condition for every node on the path from the root to the leaf, whereas the consequent simply consists of the class indicated by the leaf. For example, it is possible to convert the classification tree previously showed into the following rules:

```
1. if Tear_Production_Rate = Reduced then None
2. if (Tear_Production_Rate = Normal and Astigmatism = No) then Soft
3. if (Tear_Production_Rate = Normal and Astigmatism = Yes
   and Spectacle_Prescription = Myope) then Hard
4. if (Tear_Production_Rate = Normal and Astigmatism = Yes
    and Spectacle_Prescription = Hypermetrope) then None
```

The rules above are unambiguous (it does not matter in which order they are executed) and very simple. However this is not always the case, so whenever the resulting rules appear unnecessarily complex a *tree pruning* is usually performed in order to remove redundant tests/rules.

The "value" of a rule is measured thanks to two metrics, *support* and *confidence*; the former is simply the count of the number of correctly predicted instances, whereas the latter is the number of correct predictions, as a proportion of all the instances that the rule applies to. Poorly relevant rules can then be eliminated by pre-specifying thresholds for support and confidence.

The problem of building optimal binary decision trees is NP-complete, since it would require to generate all the possible trees and then to select the best one (which is unfeasible due to the fact that the number of possible decision trees is finite but very large) and hence the research has looked for efficient heuristics with the aim of building near-optimal decision trees. For this reason, every decision tree learning algorithm is based on heuristics, which on one hand always provides a solution in reasonable time, but it cannot guarantee that better trees have not been overlooked in the process.

1.4.3 C4.5 and C5.0

Many learning algorithms were designed with the aim of generating a classification model in the form of a decision tree starting from a set of examples. We decided to focus on the most widely-used one which is the popular C4.5, developed by Ross Quinlan as an extension of his earlier ID3 algorithm; C4.5 implementation is open source and written in C for UNIX environments and its Java implementation is known as J48 in the Weka Data Mining Package.

Unlike ID3, C4.5 supports continuous attributes, efficiently deals with missing or unavailable values, avoids overfitting the data by determining how deeply to grow a decision tree, and, among the other things, it supports post-pruning of trees and rules, provides better attribute selection measures while also improving the overall computational efficiency.

The input of C4.5 is simply a set of already classified data in which each instance consists of a n-dimensional vector containing the attribute values. One of these attributes represents the category of the record, and such category attribute contains the possible class values through which instances are cassified. Usually, the category attribute takes only the values "true", "false", or something equivalent: in the CEP case, the values are "positive", "negative" and they refer to traces of events.

The problem is to determine a decision tree that based on the outcomes of the tests on the non-category attributes correctly predicts the value of the category attribute. To accomplish this task, C4.5 employs binary splits (e.g., `Temperature < 24.5`) for numeric attributes and every attribute has many possible split points. The splitting criterion (i.e., deciding which attribute has to be selected for the split) is the normalized *information gain*, which is an information theory measure defined as the difference in entropy, useful to understand how *informative* is the presence of an attribute in the tree. In order to build a "good" decision tree, the most informative attributes should be selected first and less informative attributes are disregarded and not included in the tree, or included only if no better choice is available. The interested reader may find out more about entropy and information gain in [29] and [30].

An important feature included in C4.5 but absent in ID3 is the capability to successfully classify unknown, missing, or previously unseen instances. In fact, when the outcome of a test cannot be determined, C4.5 goes down the tree exploring all the possible subtrees and arithmetically

combines the encountered outcomes, thus being able to provide a classification even when dealing with missing values. The basic assumption behind this method is that unknown test outcomes are distributed probabilistically according to the relative frequency of known outcomes.

The direct successor of C4.5 is C5.0; its Unix single-threaded version (including source code) is released under GPL license and can be downloaded for free at [2]. Although commercial, multi-threaded and optimized implementations of C5.0 are available (along with the Windows release, See5), the author ensures that ultimately the classifiers built are exactly the same both for free and commercial releases [5].

Quinlan proved C5.0 to be much better than its predecessor while also offering new important functionalities. In fact, according to many tests conducted on different datasets [5], C5.0 heavily outperformed its predecessor in all ways, showing better accuracy (i.e. a reduced error rate on unseen instances), better speed and lower memory consumption (with one order of magnitude less of memory required). Also, C5.0 usually produces less rules without hindering the accuracy and the performance of the learned rules; and lastly, C5.0 decision trees turn out to be ultimately faster and smaller as well, and their considerable lower number of leaves leads to faster classifications.

But the crucial aspect that convinced us to choose C5.0 over C4.5 in the current research is represented by many new important functionalities offered by the algorithm. First of all, C5.0 introduces *variable misclassification costs* that allow to construct classifiers minimizing the expected missclassification costs rather than error rates; the reason behind the introduction of the feature is that, in many scenarios, some errors are more serious than others and hence they should not be treated equally like it was the case with C4.5. This new functionality assumes a very important role in our work, since we dedicated an entire component of the architecture developed just to provide a good estimation of the misclassification costs for C5.0.

Other worth mentioning new features offered by C5.0 (but not much exploited in the present work) are the provision for a *case weight attribute* that quantifies the importance of each case (useful when dealing with cases of unequal importance, aiming at minimizing the weighted predictive error rate), and new data types available (including dates, times, timestamps, ordered discrete attributes and case labels; attributes can be even defined as *functions* of other attributes). Finally, C5.0 supports

extended options like *sampling* and *cross-validation*, as well as *attribute winnowing*, which is useful when dealing with high-dimensionality applications that include hundreds or thousands of attributes.

Practical details on the C5.0 software can be found at [1], where a brief tutorial describes how to use the program.

Chapter 2

Problem Definition

In this chapter we define the thesis problem. First, in [Section 2.1] the research question is described in an informal way; then, in [Section 2.2] it follows a more rigorous definition, and lastly, in [Section 2.3] we clear and motivate some assumptions adopted which simplify the problem and make it easier to tackle.

2.1 Problem Statement

From a high-level point of view, our main problem is to enable general-purpose ML algorithms to work with streams of events in order to detect situations of interest (composite events).

As already known, any learning or data mining technique require a training dataset to work; however, in the case of CEP we can only observe a stream of *events*, which are complex entities characterized by particular properties and following a precise structure dictated by a model.

As a consequence, the stream of events must be first *translated* into a static dataset to be exploited for learning purposes. More precisely, we define the *training history* H_T as a time-ordered *set* of events; then, our problem becomes to *encode* all the relevant information contained in H_T as a series of lines (*instances*, or *samples*) containing comma-separated values of properly crafted *attributes*, thus obtaining a *dataset* D_T necessary to train the learning model.

In addition, after the learning phase is over, the learned model must be exploitable for Event Recognition purposes; in other words, it must be able to correctly detect composite events within new incoming streams.

2.2 Detailed Formulation

In order to provide a more formal definition of the problem, we introduce the concept of *traces* of events, which are just contiguous portions (subsets) of the training history characterized by the following properties:

- A trace requires both a starting and an ending timestamp (TS_{start} and TS_{end} respectively) and it contains all the events recorded within the given timeframe. The *size* W of the trace is defined as
$$W = TS_{end} - TS_{start}$$
This parameter is also called *window size* since it relates to the window operator presented in [Section 1.1], that determines how long the portion of relevant input must be before evaluating other operators.

- All the events contained in a trace must be primitive, with the only exception consisting of the *last* event of the trace, that can be a composite event.
If this is the case, we say that the given trace is *positive*, since its primitive events ultimately lead to an occurrence of the composite event. Otherwise, the trace is *negative*.

As a simple example, consider the following set of traces (where, for each event, the letter refers to the event type and the number to its timestamp):

```
T1: A@0, B@2, C@5, A@8, A@10, D@11, CE@15
T2: D@4, C@8, B@11, A@12, B@13, C@17, CE@19
T3: D@11, B@19, A@26
```

`T1` and `T2` are positive traces, because their last event is represented by the composite event `CE`, whereas `T3` is negative since it does not lead to `CE` (which in fact does not appear in it).

It is easy to notice that the following pattern, describing the occurrence of `CE` in terms of primitive events, emerges from the traces above:

```
within 15s. { A() and B() and C() }
where       { A → B }
```

The pattern (or *rule*) states that, in any positive trace, an event of type `A`, an event of type `B` and an event of type `C` must occur (i.e., the trace must include the conjunction of those events); also, the sequence operator `A → B` indicates that events of type `A` must always precede events of type `B`.

The pattern takes place in both the positive traces above (`T1` and `T2`), but – as expected – it cannot be found within the negative one (`T3`); therefore, the rule above makes it possible to *discriminate* between positive and negative traces, which in turn allows to correctly predict future occurrences of composite events.

At this point, we are finally able to give a more detailed formulation of the thesis problem.

Problem Formulation. *Given a starting set of traces θ_T obtained from the training history H_T, and a set of new, previously unseen and unlabeled traces θ_U, the problem is to correctly predict the occurrences of the complex event CE inside θ_U or in other words to assign a label (positive or negative) to each trace of θ_U.*

The problem is further summarized as follows (where $\theta_{U,p}$ is the set of positive traces and $\theta_{U,n}$ the set of negative traces of θ_U):

$$\boxed{\begin{array}{c}\textbf{Given } \theta_T \in 2^{H_T} \textbf{ , } \theta_U = \theta_{U,p} \cup \theta_{U,n} \textbf{ :} \\ \forall \epsilon \in \theta_U \textbf{, decide whether } \epsilon \in \theta_{U,p} \textbf{ or } \epsilon \in \theta_{U,n}\end{array}}$$

2.3 Assumptions

To better understand the formal description of the problem just presented, as well as the solution that we propose in the next chapters, it is worth remarking some points beforehand:

- The use of the power set 2^{H_T} is due to the fact that traces are contiguous portions of the training history and, therefore, θ_T is an *element* of the set of all the possible subsets of H_T;
- The traces in θ_U differ from the ones of θ_T in that they do not contain occurrences of complex events CE; this represents the fact

of *not knowing* whether a trace leads to CE or not (i.e., whether it is positive or negative). Therefore, a trace ϵ may belong to $\theta_{U,p}$ even if its last event is not CE.

- As a consequence, $\theta_{U,p}$ and $\theta_{U,n}$ are not known sets and our goal consists in building both the subsets in the most *accurate* way.

  In other words, our aim is not only to take a decision about ϵ, but also to take *a correct one*. In order to measure the accuracy of the predictions, we employ precision and recall (already mentioned in [Chapter 1.4]).

It should also be noted that the formulation above actually represents a *classification problem*, since the goal is to assign a *label* ("positive" or "negative") to each trace. Our approach both at formulating the problem and at solving it differs from ad-hoc solutions because a *decision tree* (and not a rule) is learned starting from H_T.

Last but not least, a fundamental assumption that we rely on throughout the course of this work regards the window size W, which is considered *fixed* and *already known* in advance and hence it is not learned by the system that we developed.

The reason for this choice is motivated by the fact that in most real-world scenarios windows do not usually represent a critical issue; if the applicative domain is well-known – as it often is the case – the order of magnitude of a proper window size is known as well. For example, in environmental monitoring systems the timeframe to consider typically consists of several minutes, whereas in a trading stock scenario it is required to be much smaller, e.g., in the order of seconds. Therefore, we consider the assumption above to be reasonable and acceptable at least from a practical viewpoint.

Moreover, learning window sizes would be a very computationally demanding and time-consuming operation, as it would require several executions of the machine learning tool employed in order to work. In fact, the window size parameter is the only one that we did not manage to learn with a *single pass* of the algorithm (which on the other hand is a noticeable advantage provided by our technique). Nonetheless, we would have faced significant issues even if we executed the program multiple times adopting different sizes and selecting the best one based on the precision, recall or F-score obtained after each execution, because it is

not trivial to decide about the *number* of possible candidate window sizes and about *which* particular values should be tested in the first place; if we are looking for an exact value – or a very good approximation thereof – the number of iterations needed is likely to quickly become prohibitive.

Chapter 3

Towards a Solution

After having tried to apply with little success different kinds of Machine Learning algorithms, such as Inductive Logic Progamming (FOIL,GOLEM) and Association Rule Mining techniques, we chose to focus our research on decision trees, and, in particular, on the already presented C5.0 algorithm, since we realized that the decision tree paradigm was better suited and easier to adapt to the CEP context than the other solutions. Moreover, the new functionalities and capabilities offered by C5.0, if compared to other algorithms and even its predecessor C4.5, turned out to be particularly useful in addressing many issues that we encountered.

Basically, our work was driven by a fundamental objective, which is to allow C5.0 to *learn* from traces of events. This requires the creation of a *dataset*, containing the properties of the events, thanks to which C5.0 can start building a decision tree oriented at discriminating between positive and negative traces. In a second step, we aim at employing such decision tree to classify *new* traces of events with the best accuracy possible.

Operatively, we managed to accomplish the goals above by taking several steps:

1. First, it is necessary to find a way to *extract* a set of traces, which constitute the basis for learning, from the training history.

2. Then, the traces so obtained are *encoded* to generate a *dataset* that can be read in input by C5.0.

   This encoding step is fundamental, since general-purpose, off-the-shelf tools are not designed to deal with events, and in particular

they lack a notion of time and sequence which on the other hand is fundamental in CEP.

3. The algorithm is finally executed on the dataset and a *decision tree* is built consequently. However, such decision tree cannot be directly employed to label event traces because of the previously adopted encoding scheme.

   Therefore, it was necessary to develop a custom *Classifier* to allow the decision tree to label new traces of events; this component also greatly helps to improve the overall accuracy of the predictions.

4. Lastly, a feature offered by the algorithm itself, which is the possibility to employ *misclassification costs*, is exploited to further lower the number of incorrect predictions made by C5.0.

   The misclassification costs estimator internally runs C5.0 and performs many cross-validations on the training dataset in order to produce a proper estimate.

The learning architecture just described is showed in [Figure 3.1]; it should be viewed as a black-box that receives two inputs: a *training history* and the *window size*, which is needed in order to extract traces from the training history since it specifies how long a trace should be. The output of the black-box is the learned decision tree, which is then exploited to label unseen traces of events working in conjunction with the *Classifier*.

The *Traces Extractor* is the simplest component and it is in charge of looping throughout the entire training history, providing in output the set of extracted traces (both positive and negative). To obtain the set T of extracted traces, we employ the following procedure:

Algorithm 1 EXTRACT_TRACES

Input: Training history H_T, window size W
Output: Set of training traces T

```
TS_start ← TS_min, T ← ∅, ϵ ← ∅
while TS_start + W ⩽ TS_max do
    ϵ ← GetAllEventsInWindow(H_T, TS_start, TS_start + W)
    T ← T ∪ {ϵ}
    TS_start ← TS_start + 1
return T
```

where TS_{min} and TS_{max} are the minimum and maximum timestamps of events as observed in the training history H_T and ϵ is the current trace being extracted. The subprocedure GetAllEventsInWindow simply returns a trace containing all the events of H_T that carry a timestamp included in the time-frame $[TS_{start}, TS_{start} + W]$.

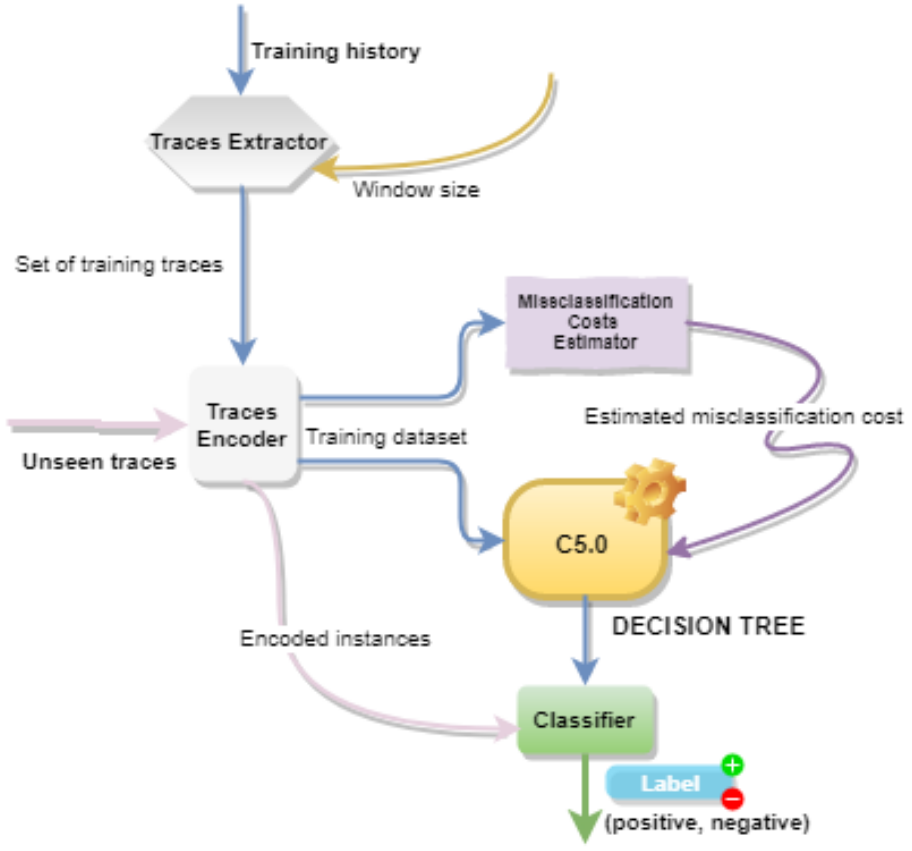

Figure 3.1: Overview of the learning architecture developed.

Chapter 4

System Architecture and Implementation

In the course of this chapter, we proceed to detail every component of the learning architecture developed and introduced in the previous chapter.

In particular, the *Traces Encoder* is described in [Section 4.1], the *Classifier* can be found in [Section 4.3] whereas the *Misclassification Costs Estimator* is presented in [Section 4.4].

About the remaining parts of the chapter, [Section 4.2] describes the problem of Event Type Repetitions, an important issue that emerged during the encoding and that was necessary to fix, whereas [Section 4.5] and [Section 4.6] extend the basic model developed to deal with more complex situations, respectively characterized by additional constraints (negations and aggregates), and multiple rules included within the same history.

4.1 Trace Encoding

In order to successfully employ C5.0, it is required to encode all the information about events as a *finite set of attributes* in a way that allows the algorithm to discriminate between positive and negative traces.

This is accomplished in two subsequent steps:

- first, all the significant properties of events and traces are precisely identified;

- then, an attribute (or a set of attributes) is created to represent each property.

With reference to the event model adopted, we need to encode at least the following properties:

- the *presence* of event types in a trace;
- the *values* assumed by event attributes;
- the *temporal relations* taking place between events.

The *presence* of event types is important because composite events may be triggered by some other kind of primitive event that happened at the same time or before (for instance, the event `Fire` triggered by the events `Smoke` and `HighTemperature`).

A type is *relevant* if its presence in a trace is required to make a composite event occur; therefore, positive traces must contain at least one occurrence of each relevant type.

If the universe U of primitive event types in the domain of interest is finite and fully known, it is rather easy to encode the presence of event types as it is sufficient to employ a discrete *boolean* attribute for each possible type, taking the value `"true"` if the type has been detected within the trace and `"false"` otherwise.

We express this class of attributes in the form $\texttt{Event}_x\texttt{Detected}$, with $1 \leq x \leq N$, $Event_x \in U$ and $N = |U|$; since an attribute is created *for each type*, we need N attributes to fully encode all the information.

In some cases, the presence of event types alone is not very useful. Take, for example, a scenario composed by sensors periodically emitting information, such as temperature readings: since events happen at regular intervals, some types appear in every collected trace and thus they no longer constitute a peculiar characteristic of positive traces.

This is why it becomes necessary to also encode the *values* of the attributes of an event and not just its simple occurrence; furthermore, some composite events only take place if the attributes of a relevant event assume certain values, according to a *selection* constraint (e.g., the complex event `Fire` triggered by the event `Temperature(Value > 28.5)`).

Therefore, a *different C5.0 attribute* is created for each existing *event attribute*. These attributes may be either continuous or discrete depending on the event type, but for simplicity we assume that they are all *continuous*, implying that event attributes are integers or real numbers. If a given event type has not been detected within a trace, all the relative attributes take the missing value, `?`.

Assuming that each event is characterized by A different attributes, we refer to this class of attributes with the formulation $\texttt{Event}_x\texttt{Attribute}_k$, where $1 \leq x \leq N$ and $1 \leq k \leq A$. If each trace contains at most one occurrence of a given type, we need to create $N*A$ attributes of this kind.

Sometimes, we must take into account the *temporal orderings* of primitive events in order to detect composite events; as a consequence, we tried to encode data in a way that allows C5.0 to implicitly spot *sequence constraints*. To do that, we created a number of additional discrete *boolean* C5.0 attributes, each one representing the mutual temporal relation intercurring between a couple of events. For example, the attribute $\texttt{Event}_x\texttt{PrecedesEvent}_y$ (where $x, y \in N$) takes value `"true"` if in a given trace $\texttt{Event}_x$ is detected before $\texttt{Event}_y$ (i.e. the timestamp of $\texttt{Event}_x$ is lower than the one of $\texttt{Event}_y$), and `"false"` otherwise.

If a trace only contains one out of the two events (or none of them) the attributes assume a missing value instead, because it does not exist any ordering relation between them.

The number of this kind of attributes depends once again on the number N of different event types in the domain of interest. Observing that, given $\texttt{Event}_x\texttt{PrecedesEvent}_y$, it is not needed to also encode $\texttt{Event}_y\texttt{PrecedesEvent}_x$ because one is the logical NOT of the other, the number of required attributes is

$$\frac{N * (N - 1)}{2}$$

Lastly, a *category attribute* allows C5.0 to assign a label to traces; we call such discrete attribute `Class`, and its possible values are `"positive"` and `"negative"`.

It follows, in [Table 4.1], a brief summary of the encoding model for C5.0 just described.

In conclusion of this section, I am going to provide an example showing how a trace of events can be encoded thanks to the scheme adopted.

C5.0 Attribute	Type	Possible values	Information encoded	Constraints captured	Number of attributes needed
Event$_x$ Detected	discrete	true, false	presence of event x	conjunction	N
Event$_x$ Attribute$_k$	continuous/ discrete	depending on the attribute domain	attribute k of event x	conjunction, selection	$N * A$
Event$_x$ Precedes Event$_y$	discrete	true,false,?	temporal ordering between events x and y	sequence	$\frac{N*(N-1)}{2}$
Class	discrete	positive, negative	label for traces	/	1

Table 4.1: Encoding model for C5.0.

Consider the following trace, in which $U = \{E1, E2, E3, E4, E5\}$, $R = \{E1, E2, E5\}$, $A = 2$, $W = 10$, $N = |U| = 5$ where $R \subset U$ is the set of relevant events. The occurrence of the complex event is denoted with CE whereas the usual syntax @ indicates the timestamp.

```
T: E1[A1=1,A2=2]@1, E2[A1=0,A2=5]@5, E5[A1=6,A2=6]@8, CE@10
```

First of all we look at which event types were observed within the trace and we set the corresponding attributes accordingly:

```
Event1Detected = true
Event2Detected = true
Event3Detected = false
Event4Detected = false
Event5Detected = true
```

Then, we proceed to encode event attributes and sequences (notice how missing values are assigned to events not detected within the trace):

```
Event1Attribute1 = 1
Event1Attribute2 = 2
Event2Attribute1 = 0
Event2Attribute2 = 5
```

```
Event3Attribute1 = ?
Event3Attribute2 = ?
...
Event1PrecedesEvent2 = true
Event1PrecedesEvent3 = ?
Event1PrecedesEvent4 = ?
Event1PrecedesEvent5 = true
...
```

Lastly, since the composite event CE is the last event of the trace, we can conclude that the trace is positive:

```
Class = positive
```

The result of the encoding procedure is the following *instance*:

```
t,t,f,f,f,1,2,0,5,?,?,?,?,6,6,t,?,?,t,?,?,?,?,?,?,positive
```

The pseudocode of the procedure ENCODE is showed in the next page; by printing the comma-separated values of the encoded instance I it is possible to obtain a line to be added to the training dataset for C5.0.

Instead of a single one, the procedure can be generalized to work with a *set* of traces (i.e., an *history*); we call such procedure ENCODESET.

Algorithm 2 ENCODE

Input: trace ϵ
Output: encoded instance I

```
Types ← GetEventTypesInTrace(ϵ).
for each t ∈ U do
    if t ∈ Types - {CE} then
        I.Event_x Detected ← true
    else
        I.Event_x Detected ← false
for each e ∈ ϵ do
    t ← e.Type
    if t ∈ Types - {CE} then
        Att_current ← GetAttributesOfEvent(e)
        for each a ∈ Att_current do
            I.Event_x Attribute_k ← a
    else
        for all 1 ≤ k ≤ A do
            I.Event_x Attribute_k ← ?
for each e1 ∈ ϵ do
    for each e2 ∈ ϵ do
        if x < y then
            t1 ← e1.Type, t2 ← e2.Type
            if t1, t2 ∈ Types - {CE} then
                if e1.Timestamp ≤ e2.Timestamp then
                    I.Event_x PrecedesEvent_y ← true
                else
                    I.Event_x PrecedesEvent_y ← false
            else
                I.Event_x PrecedesEvent_y ← ?
if CE ∈ Types then
    I.Class ← "positive"
else
    I.Class ← "negative"
return I
```

Variable/Procedure	Description
ϵ	Input trace (a set of time-ordered events)
U	Universe of primitive event types (not including CE)
A	Number of attributes of each event
$Types$	Set of event types present in trace ϵ (possibly including CE)
CE	A string denoting the complex event type
$1 \leq x, y \leq N$	Indexes of event types, assumed to be known at every step of the algorithm
$1 \leq k \leq A$	Indexes of event attributes
$Att_{current}$	A set containing the values of the attributes of the current event
I	Encoded instance returned by the procedure
GetEventTypesInTrace(ϵ)	Returns the set of event types detected in trace ϵ
GetAttributesOfEvent(e)	Returns the values of the attributes of event e

Table 4.2: Definitions for the Encode algorithm.

4.2 Event Type Repetitions

The encoding strategy explained works under the assumption that traces only contain *a single occurrence* of each event type.

However, it is fairly common for a trace to present multiple occurrences of events sharing the same type. Such events are generally not identical, because their attributes may as well assume different values (e.g., `Temperature(value=20)` and `Temperature(value=25)`).

Since it does not exist a concept of "occurrence" in the encoding scheme, we are currently unable to deal with traces containing event type repetitions.

Solutions for this issue are possible, but they come at the cost of including incorrect information in the dataset. To understand the concept of *incorrect information*, we should first define what we mean by *correct* information instead, distinguishing between positive and negative traces.

Positive traces are characterized by the following property: for each operator included in the pattern leading to the composite event, they must contain *at least one* event that *satisfies* the operator; if there are

different operators concerning the same event type, each of them should be satisfied by at least one event of the trace.

As an example, consider the universe $U = \{E1, E2, E3\}$ and the following CEP rule, according to which there are two relevant event types, $E1$ and $E2$, each subject to a selection equality constraint on the only attribute $A1$, which is discrete and takes values from the domain $D_{A1} = \{x, y, z\}$:

```
R:[E1[A1=x],E2[A1=y]]
```

Now consider the following trace, with the * character denoting an event occurrence satisfying a rule operator, i.e. in which $A1$ assumes the correct value (`'x'` or `'y'` depending on the type):

```
p: *E1[A1=x]@1, E1[A1=z]@2, *E2[A1=y]@3, E3[A1=z]@4, E2[A1=z]@5, CE
```

The trace is positive because it contains at least one event satisfying each constraint (`E1[A1=x]@1`, `E2[A1=y]@3`). It does not matter if other occurrences of the event types involved (e.g., `E1[A1=z]@2`, `E2[A1=z]@5`) violate the constraints, as long as there is at least one occurrence in the trace satisfying them.

On the other hand, *negative* traces require at least one rule operator to be *never* satisfied by the events of the trace (other operators may be satisfied, but not all of them at the same time).

For example, in the following trace the selection constraint on $E1$ is satisfied by the occurrence `E1[A1=x]@1`, but not the one on $E2$, which always appears with the wrong value z for the attribute $A1$ (`E2[A1=z]@3`, `E2[A1=z]@5`):

```
n: E1[A1=x]@1*, E1[A1=z]@2, E2[A1=z]@3, E3[A1=z]@4, E2[A1=z]@5
```

The considerations above lead to an important conclusion: *only positive traces* are affected by the problem of event type repetitions.

To see why this is the case, it should first be observed that while the encoding scheme do not work with traces containing repetitions, it can still be applied to all the *sub-traces* characterized by a single occurrence of each event type.

In order to generate a sub-trace we must pick only one occurrence for each repeated type, but if we select a *wrong* occurrence (i.e., not satisfying any constraint) from a *positive* trace the encoding would be

incorrect, because the instance gets labeled as "positive" even if it is not. By unknowingly removing events which are necessary to satisfy the rule, *false positives* are introduced into the training dataset before even having tried to run the algorithm.

For instance, with reference to the previous trace p, the sub-trace

```
p1: *E1[A1=x]@1, *E2[A1=y]@3, E3[A1=z]@4, CE
```

leads to a proper encoding, but the sub-trace

```
p2: E1[A1=z]@2, *E2[A1=y]@3, E3[A1=z]@4, CE
```

introduces a false positive into the dataset.

With *negative* traces we do not face the same problem, since in this case:

- either a relevant type is not present in the trace, or
- all its occurrences violate a rule operator.

Therefore, regardless of which occurrence we choose to encode into the dataset, a *false negative* can never happen (all the possible sub-traces are *true negatives*).

For example, the following sub-traces of n:

```
n1: *E1[A1=x]@1, E2[A1=z]@3, E3[A1=z]@4
```

```
n2: *E1[A1=x]@1, E3[A1=z]@4, E2[A1=z]@5
```

are both correctly labeled as negative, regardless of the choice made between the occurrences `E2[A1=z]@3` and `E2[A1=z]@5` (none of them satisfies the rule R).

As anticipated, we managed to find a solution for the problem of event type repetitions at the cost of introducing some false positives into the dataset; nonetheless, this solution exploits the original encoding scheme without needing to introduce additional attributes, which is a better option if compared to different alternatives that we tried to employ and that modify the encoding scheme.

In short, our approach proceeds as follows:

- first, traces with repetitions are *split* into *multiple* sub-traces containing a single occurrence of each event type;
- then, each sub-trace is encoded independently according to the usual scheme.
- In order to preserve all the original information, we need to generate every possible sub-trace of minimum length which can be extracted from the original trace; therefore, *all the possible combinations* of the occurrences of events having the same type must be computed.

Thanks to the method above, we are *guaranteed* that at least one of the sub-traces included in the dataset after the split is *correct*. However, as already explained, most of the sub-traces extracted from a positive trace are actually *false positives* and unluckily we cannot avoid to encode them into the dataset.

In fact, since we do not know the rule leading to composite events we cannot exploit it to correctly label sub-traces. We can only observe that a composite event happened at the end of the original trace, and all the sub-traces must be labeled as positive because we have no idea about *which* are the actual event occurrences in the trace that triggered the composite event.

The example below is useful to understand how the "splitting" method works in practice, as well as to see how false positive sub-traces are generated as a consequence.

Consider the following CEP rule, with $U = \{E1, E2, E3\}$, $D_{A1} = \{x, y, z\}$, $A = 1$:

```
[Rule:
EventConstraints: [E1[A1=x],E2[A1=y],E3[A1=z]
Sequence Constraints: [E1->E2]]
```

Now we take a positive trace containing event type repetitions (as usual, asterisks denote an event occurrence satisfying some operator of the rule):

```
*E1[A1=1]@1, *E2[A1=y]@2, E1[A1=z]@3, E1[A1=z]@4,
*E3[A1=z]@5, E2[A1=z]@6, CE@7
```

The trace is split into the following sub-traces:

```
*E1[A1=x]@1 *E2[A1=y]@2 *E3[A1=z]@5 CE %% True Positive
*E1[A1=x]@1  E2[A1=z]@6 *E3[A1=z]@5 CE %% False Positive
 E1[A1=z]@3 *E2[A1=y]@2 *E3[A1=z]@5 CE %% False Positive
 E1[A1=z]@3  E2[A1=z]@6 *E3[A1=z]@5 CE %% False Positive
 E1[A1=z]@4 *E2[A1=y]@2 *E3[A1=z]@5 CE %% False Positive
 E1[A1=z]@4  E2[A1=z]@6 *E3[A1=z]@5 CE %% False Positive
```

These sub-traces are then encoded according to the usual attributes:

```
E1A1: continuous
E2A1: continuous
E3A1: continuous
E1PrecedesE2 : true,false
E1PrecedesE3 : true,false
E2PrecedesE3 : true,false
Class: positive, negative
```

The result of the encoding are six different instances to be included in the dataset for C5.0:

```
*x ,*y,*z,*true,true,true,positive  %% True Positive
*x*, z,*z,*true,true,false,positive %% False Positive
 z ,*y,*z,false,true,true,positive  %% False Positive
 z , z,*z,*true,true,false,positive %% False Positive
 z ,*y,*z,false,true,true,positive  %% False Positive
 z , z,*z,*true,true,false,positive %% False Positive
```

It can be observed that the first line generated with the "splitting" strategy is a *true positive*, so the correct information is preserved. On the other hand, five (partially) *incorrect* instances are introduced into the dataset as well. Considering the correctness of the individual values encoded for each attribute, it can be noted that only 1/3 of the total information about $E1$, half the information about $E2$, and 2/3 of the information concerning the sequence constraint is correct.

The pseudocode of the procedure SPLIT_TRACE, employed to split traces with repetitions, is showed in the next page. Please note that, if the input trace is *positive*, the composite event type CE is treated like a regular type by the procedure and therefore it is included at the end of all the sub-traces generated.

4.3 Classifying Traces

By executing C5.0 on the dataset built according to the encoding scheme described, a decision tree classifier is generated by the algorithm and it can be employed to label *unseen traces*, allowing to detect new occurrences of composite events in real-time.

To classify a new trace, we follow the same procedure already adopted for the training traces, i.e. we split the trace (if it contains type repetitions) and then we proceed to encode all the sub-traces. At this point,

Algorithm 3 SPLIT_TRACE

Input: trace ϵ
Output: set of sub-traces E_{sub} obtained by splitting ϵ

```
Types ← GetEventTypesInTrace(ϵ)
O_all ← ∅
E_sub ← ∅
for each t ∈ Types do
    O_current ← GetAllEventOccurrencesOfType(t)
    O_all.add(O_current)
E_sub ← ComputeCombinations(O_all)
return E_sub
```

Variable/Procedure	Description
ϵ	Input trace, possibly containing event type repetitions
$Types$	Event types present in ϵ
$O_{current}$	Set containing all the event occurrences of the current type t
O_{all}	A set of sets of occurrences, containing every $O_{current}$ computed
GetEventTypesInTrace(ϵ)	Returns all the event types detected in trace ϵ
GetAllEventOccurrencesOfType(t)	Returns all the occurrences of events of type t in ϵ
ComputeCombinations(O)	It picks a single occurrence from each set of occurrences in O in every possible way, returning the set of all the possible sub-traces that can be built starting from ϵ

Table 4.3: Definitions for the SPLIT_TRACE algorithm.

C5.0 can assign a label to every encoded sub-trace thanks to the previously computed decision tree.

However, classifying *sub-traces* does not actually make sense per se, since it is the *original trace* (i.e., the one containing type repetitions) that ultimately has to be classified, even if this cannot be done directly because of the limits of our encoding model.

To overcome this problem, we assumed that if *at least one* of the sub-traces is classified as positive by C5.0, then the original trace is also

classified as positive. In other words, if we look at the "positive" and "negative" labels as boolean "true" and "false" values respectively, this method of detection is equivalent to a *logical OR* performed on the labels assigned to sub-traces.

The strategy brings two important consequences:

- it is easier to detect positive traces because this only requires a single sub-trace labeled as positive, and therefore it is harder to incur into false negative detections;
- it is harder to detect negative traces because in this case *all* the sub-traces must be given a negative label; C5.0 needs to be extremely accurate because a single incorrect classification of a sub-trace inevitably leads to a *false positive* detection.

It is possible to eliminate the drawback of the solution remembering that only *true negatives* can be obtained from the split of a negative trace. Therefore, it is not necessary to encode every negative sub-trace since they are equivalent to the initial trace in terms of correctness and validity, even if containing only a subset of the original information. In fact, the split of a positive trace is required only to be sure to include *at least one true positive* in the dataset (along with the false positives), but we do not have to worry about this issue for negative traces.

Hence, we opted to encode *only one* randomly selected negative sub-trace; in this way, we add to the dataset a valid instance but the chances of getting a false positive detection are significantly lowered. Of course, different datasets are built depending on the choice of the sub-trace but we argue that the statistical impact of the choice is negligible, given a large set of training traces and as confirmed by many experimental evaluations conducted.

As usual, it follows the pseudocode of the procedure CLASSIFY_TRACE, employed to classify traces.

4.4 Reducing False Positives

The "splitting" method employed for traces with repetitions actually leads to the construction of a *less* accurate decision tree, but this is true only on a *per sub-trace/dataset instance basis*. As we have seen,

Algorithm 4 CLASSIFY_TRACE

Input: trace ϵ, decision tree DT
Output: label L assigned to ϵ

```
L_sub ← ∅
E_sub ← SPLIT_TRACE(ε)
for each σ ∈ E_Sub do
    L_sub.add(l)
if L_sub.contains("positive") then
    L ← "positive"
else
    L ← "negative"
return L
```

Variable/Procedure	Description
ϵ	Input trace, possibly containing event type repetitions
DT:	Decision tree built by C5.0 during the learning phase
L	Label returned by the procedure and assigned to ϵ
E_{sub}	Set of all the possible sub-traces of ϵ
L_{sub}	Set of labels assigned to every trace in E_{sub}
ENCODE(σ)	Encodes the trace σ as an instance compatible with C5.0
CLASSIFY(DT,I)	Exploits the C5.0 built-in classifier to assign a label to the encoded instance I thanks to DT
SPLIT_TRACE(ϵ)	Returns all the possible sub-traces obtained by splitting ϵ

Table 4.4: Definitions for the CLASSIFY_TRACE algorithm.

the classification strategy employed afterwards improves the situation, bringing a much higher number of correct predictions for the original starting traces and a resulting overall lower *false positives* count.

To further reduce the number of false positives, we decided to rely on two more techniques: the use of *unbalanced training datasets* and the estimation of *misclassification costs*.

The basic idea behind the first technique stems from simple empirical findings; in fact, we observed that when the training dataset for C5.0 is composed by a greater number of *negative* instances than positive ones, the algorithm is usually able to assign negative labels more accurately

and, to an extent, without losing its capability to correctly predict positive instances.

It is empirically possible to find unbalancing proportions that work well in every tested scenario (like 1:10, for example); however, we are not able to show any proof that a given fixed unbalancing proportion is going to work in every practical case nor we can provide any theoretical way to estimate the optimal proportion to be employed, so this remains an open issue.

What we can state for sure, on the other hand, is that the concept of *biased dataset* should not raise any concern per se, even in practical applications. In fact, as long as the training dataset constitutes a *good approximation* of the true distribution over samples and labels, it is always legitimate to employ it; this is the case with CEP, because composite events usually represent some *extraordinary*, out of the norm condition and hence positive traces are *intrinsically much more rare* than negative ones. For example, the event `Fire` in an environmental monitoring scenario is not likely to be observed frequently, so it is not a difficult task (but the normality) to get a biased dataset in such an applicative context.

The other method employed to deal with false positives is the use of *misclassification costs*, which express the cost of an error in predicting a class label for an instance: if not specified, such errors are all treated equally by C5.0 but, in some cases, classifying an instance of a class with an incorrect label carries greater weight than a generic misclassification error.

For example, if we want to instruct the algorithm about the fact that misclassifying an actual negative trace as positive is k times more costly than the opposite we can just write the following line:

```
positive[Predicted Label], negative[Real Label]: k
```

By employing different costs, C5.0 is able to build decision trees oriented at minimizing the number of *more costly* errors. The direct consequence of this fact is that the algorithm becomes biased at accurately predicting the instances of some class, possibly at the expenses of some other class.

In terms of quality metrics, what happens in our case (considering that we only have two class labels) is that either precision or recall gets

boosted while the other metric is sometimes lowered. The total balance can be negative or positive, but the bottom line is that we are obviously interested into boosting the lowest metric without impacting too much on the other one.

Therefore, adopting misclassification costs becomes particularly convenient in situations characterized by a metric which happens to be very high (close to 1.0) whereas the other is mediocre (0.5-0.8), like in our case (high recall, poor precision).

At this point, two questions naturally arise from the considerations above.

- How to *estimate* the best possible misclassification costs for a dataset? (i.e., the ones providing the best accuracy possible in a particular scenario)
- How to decide whether a given misclassification cost is *better* than another one?

The second question is actually instrumental to the first one and, in general, the answer heavily depends on the applicative context. For example, it can happen that in a given CEP application a metric is deemed more important than the other (for example, in a medical scenario, where a false negative diagnosis is practically never tolerated but a false positive is more acceptable).

In such cases the answer is simple: a misclassification cost is better if it helps to improve the most relevant metric. Here, we *either* aim at reducing false positives *or* false negatives. Otherwise, if precision and recall are deemed of equal value, a different solution is needed.

It is easy to notice that the F-score metric, being defined as the harmonic average of precision and recall, is very well suited for this purpose. In fact, if a misclassification cost provides a gain for a metric which is greater than the loss of the other, the corresponding F-score is going to be higher; therefore, we may just decide to choose the costs maximizing the F-score. Even if in our experiments we mainly had to deal with false positives (and therefore the estimation could have been driven by *precision* alone), we still decided to rely on the F-score metric in order to preserve generality.

The estimation has been conducted by exploiting a *subset* of the set of training traces for learning whereas the rest of the set has been used

to *evaluate* C5.0. The entire process is repeated with different misclassification costs and the desired estimate is obtained by comparing the resulting F-scores provided by the validations conducted on the *testing* portions of the set.

To get a more accurate estimate, we decided to perform a *cross validation* on the set of traces, which is a useful technique widely employed in Machine Learning that aims at *predicting* how well a learned model will perform and generalize on new data without actually testing it on such data. The goal of cross validation is to define a dataset to "test" the model during the training phase, and, in our case, the outcomes of the predictions are then exploited to choose a proper misclassification cost.

Due to performance issues and computational limits, we opted for a *non-exhaustive* method of cross validation (i.e., not considering all the possible ways of splitting the original samples), which is *k-fold cross validation*.

With k-fold cross validation, the training dataset is split into k portions (or *folds*, or *sub-samples*) of equal size (i.e. each fold contains an equal number of instances); then, *k-1* folds are used to build a dataset to train the model and the remaining fold is used to validate the model. The process is repeated k times, and at each round a different fold is selected for the evaluation; at the end of the k-th round, every fold has been selected exactly once.

For instance, with a 3-fold cross validation (which we often employed in our tests), at each round roughly 2/3 of the instances are used for training and the remaining 1/3 for the evaluation.

It is important to underline the fact that we performed cross validation not directly on the training *dataset* of C5.0, but on the set of traces extracted from the *training history*. In other words, we proceeded to the encoding phase only after having generated k folds of both positive and negative *traces*. The reason for this is given by the fact that, if we encoded traces first and then divided the resulting dataset into k folds, it could happen that some instances obtained from the split of a certain trace (containing type repetitions) end up into a fold, while some other instances deriving from the same trace go into a *different* fold, which does not make sense even if the statistical impact would actually be quite small.

Now, it is interesting to finally find out how cross validation helps to provide a proper estimate of misclassification costs: *for each fold*, we test a number of different misclassification costs and then we select the one providing the best F-score for that fold. The final misclassification cost estimate is simply the *average* computed on all the best values (one for each fold).

The system turned out to provide good results, even if not always optimal. In fact, two factors prevent the optimality of the prediction.

- The *number* of folds employed: generally speaking, employing too many folds is impractical both because of the higher number of iterations of the algorithm required (often resulting in prohibitive running times) and because the datasets used for learning and testing could end up containing too few samples, which would lead to bad estimations; however, on the other hand, higher values of k are able to provide less bias towards overestimating the true expected error and a more accurate confidence interval on the estimation.

- The *choice* of the possible "candidate" misclassification costs to test and the *number* of such candidates; in fact, if N_{folds} is the chosen number of folds and M the number of misclassification costs to be tested, the number of C5.0 runs required to provide a final estimation is equal to $N_{folds} * M$. Luckily, even if misclassification costs could take any value, from a practical viewpoint the range of worth testing costs is very limited (usually, the optimal costs are included in the range [1,100] and not rarely even in the range [1,10], so in our experiments at most we run C5.0 $N_{folds} * 100$ times - or less, if we decide to test only a subset of the possible values). Nonetheless, in order to get a theoretically optimal prediction every natural number should be tested (until we get a perfect F-score, if possible) and this is obviously not feasible.

The pseudocode of the procedure ESTIMATE_MISCLASSIFICATION_COSTS is showed below.

Algorithm 5 ESTIMATE_MISCLASSIFICATION_COSTS

Input: F_P, F_N, N_{folds}, minimum cost c_{min}, maximum cost c_{max}
Output: misclassification cost estimate c

$k \leftarrow 1$
$C_{best} \leftarrow \emptyset$, $Dataset_T \leftarrow \emptyset$, $Dataset_E \leftarrow \emptyset$
do
  $Pos_T \leftarrow \bigcup_{j=1...N_{folds}, k \neq j} F_P$.get(j)
  $Neg_T \leftarrow \bigcup_{j=1...N_{folds}, k \neq j} F_N$.get(j),
  $Pos_E \leftarrow F_P$.get(k), $Neg_E \leftarrow F_N$.get(k)
  $Dataset_T$.add(ENCODESET(Pos_T))
  $Dataset_T$.add(ENCODESET(Neg_T))
  $c_{current} \leftarrow c_{min}$
  $c_{best} \leftarrow c_{min}$
  $F_{best} \leftarrow 0$
  do
    $DT \leftarrow$ EXECUTE_C5.0($Dataset_T$, $c_{current}$)
    $F1_{current} \leftarrow$ EVALUATE_F1($Pos_E \cup Neg_E$, DT)
    if $F1_{current} > F1_{best}$ **then**
      $F1_{best} \leftarrow F_{current}$
      $c_{best} \leftarrow c_{current}$
    $c_{current} \leftarrow c_{current} + 1$
  while $c_{current} \leq c_{max}$
  C_{best}.add(c_{best})
  $k \leftarrow k + 1$
while $k \leq N_{folds}$
$c \leftarrow$ AVERAGE(C_{best})
return c

Variable/Procedure	Description
N_{folds}	Number of folds employed for cross validation
F_P	Set of positive folds (it contains sets of positive traces)
F_N	Set of negative folds (it contains sets of negative traces)
c_{min}	Minimum misclassification cost to be tested
c_{max}	Maximum misclassification cost to be tested
c_{best}	Best misclassification cost for the current fold
$c_{current}$	Current misclassification cost being tested
C_{best}	Set containing the best costs estimated for each fold
c	Final misclassification cost estimated
Pos_T	Set of positive traces currently employed for training
Pos_E	Current fold used for the evaluation, containing positive traces
Neg_T	Set of negative traces currently employed for training
Neg_E	Current fold used for the evaluation, containing negative traces
$Dataset_T$	Set of instances obtained after encoding Pos_T and Neg_T
DT	Decision tree produced by running C5.0 on $Dataset_T$ with $c_{current}$
$F1_{current}$	F-score obtained by evaluating DT with $c_{current}$
$F1_{best}$	Best F-score obtained for a fold
ENCODESET(T)	Encodes the set of traces T producing a set of instances
EXECUTE_C5.0(D,c)	Runs C5.0 on the dataset D using cost c
EVALUATE_F1(H,DT)	Provides the F-score calculated after the evaluation of DT on history H (note that the union $Pos_E \cup Neg_E$ is considered a history since it is a set of traces)
AVERAGE(S)	Computes the average of the values of set S

Table 4.5: Definitions for the ESTIMATE_MISCLASSIFICATION_COSTS algorithm.

4.5 Negation and Aggregate Constraints

Until this point, we considered CEP rules characterized by only *selection*, *conjunction* and *sequence* constraints. After having observed that our system is able to excellently deal with them, we asked ourselves if it could also work in presence of additional rule operators.

The answer is affirmative and we did manage to adapt our system to deal with more complex patterns and rules which include *negation* and *aggregate* operators.

We now analyze the semantic of each new constraint introduced.

- *Negation* constraints require a certain event type to be *absent* from a positive trace, or to be present in a positive trace with attributes *not assuming* particular values. For example, according to the following rule containing a negation:

  ```
  [Rule:
  EventConstraints:[E1[A1=x]]
  NegationConstraints:[E2[A1=x]]]
  ```

  The following trace is labeled as "positive" since it does not contain events of type $E2$:

  ```
  E1[A1=x]@1, E1[A1=y]@2, CE1@3
  ```

  Now consider the following trace instead:

  ```
  E1[A1=x]@1, E1[A1=y]@2, E2[A1=y]@3, CE1@4
  ```

  Also this trace is positive because, even if it contains an event of type $E2$, its attribute $A1$ does not assume the value x. Therefore, every trace labeled as "negative" must contain the event `E2[A1=x]`; otherwise, it is positive (if it satisfies the other rule constraints).

  In order to deal with negations, it is not necessary to introduce additional attributes for the encoding, since the attribute $\texttt{Event}_x\texttt{Detected}$ is already capable of representing the *absence* of an event type in a trace, whereas attributes of the kind $\texttt{Event}_x\texttt{Attribute}_k$ can be exploited to detect the values *not* to be assumed by an event, working in the opposite way as they did with selection constraints.

- *Aggregate* constraints filter events according to some aggregated function. The functions supported in the present work are minimum, maximum, average and sum; therefore, they can only be applied when events are characterized by *numeric* attributes (integer or real numbers).

  In our model, a single aggregate constraint apply to a given *event attribute* of a given *event type* and takes into account all the occurrences of that type in a trace. The aggregated functions mentioned

above are self-explanatory; nonetheless, we summarize them in the following [Table 4.6]

Function	Description
Minimum	The minimum value of an event attribute as observed in a positive trace must be equal, greater than, or lesser than a specified threshold
Maximum	The maximum value of an event attribute as observed in a positive trace must be equal, greater than, or lesser than a specified threshold
Average	The average value of a given event attribute computed over all its occurrences inside a positive trace must be equal, greater than, or lesser than a specified threshold
Sum	The sum of the values of a given event attribute computed over all its occurrences inside a positive trace must be equal, greater than, or lesser than a specified threshold

Table 4.6: Possible functions included in aggregate constraints.

As an example, now consider the following rule:

```
[Rule:
EventConstraints:[E1[]]
AggregateConstraints:[Average[E1.A1 > 28]]
```

The rule requires that the *average* computed on the values assumed by the attribute $A1$ in *all* the occurrences of events of type $E1$ must be greater that 28 to make a composite event happen; otherwise, the trace is negative.

Here, the situation is different from the case of negations, since the attributes currently employed to encode traces (summarized in [Table 4.1]) are not sufficient to capture aggregate constraints.

Therefore, we added to the encoding scheme additional attributes that specifically represent aggregates, with the aim of enabling C5.0

to also consider this kind of operator when it comes to discriminate positive traces from negative ones.

In particular, we created a different class of attributes for each possible kind of aggregate operator:

- For each attribute of each event type, we introduced an attribute of the kind $\texttt{Event}_x\texttt{Attribute}_k\texttt{Minimum}$ and an attribute $\texttt{Event}_x\texttt{Attribute}_k\texttt{Maximum}$ containing respectively the minimum and maximum values of the k-th attribute of event x as observed in the current trace being encoded; both the classes require the creation of $N * A$ attributes.

  Even if they are somewhat redundant, since some of the information of $\texttt{Event}_x\texttt{Attribute}_k$ may be repeated, the new attributes showed to be able to greatly increase the performance of C5.0 while being conceptually more adequate to represent aggregates.
- Moreover, we created two more classes of attributes, $\texttt{Event}_x\texttt{Attribute}_k\texttt{Sum}$ and $\texttt{Event}_x\texttt{Attribute}_k\texttt{Average}$, respectively containing the sum and the average of *all* the values assumed by attribute k in every occurrence of event x within the current trace. Also these classes of attributes require the creation of $N * A$ attributes.

We show the updated encoding model in [Table 4.7]; this final model contains all the required attributes to deal both with the "old" operators (selections, conjunctions and sequences) as well as the new ones which have just been presented (i.e., negations and aggregates).

C5.0 Attribute	Type	Possible values	Information encoded	Constraints captured	Number of attributes needed
Event_x Detected	discrete	true, false	presence of event x	conjunction	N
Event_x Attribute_k	continuous/ discrete	depending on the attribute domain	attribute k of event x	conjunction, selection	$N * A$
Event_x Precedes Event_y	discrete	true,false,?	temporal ordering between events x and y	sequence	$\frac{N*(N-1)}{2}$
Event_x Attribute_k Minimum	continuous	depending on the attribute domain	minimum value of attribute k of event x	aggregate (min)	$N * A$
Event_x Attribute_k Maximum	continuous	depending on the attribute domain	maximum value of attribute k of event x	aggregate (max)	$N * A$
Event_x Attribute_k Average	continuous	depending on the attribute domain	average of the values of attribute k of event x	aggregate (avg)	$N * A$
Event_x Attribute_k Sum	continuous	depending on the attribute domain	sum of the values of attribute k of event x	aggregate (sum)	$N * A$
Class	discrete	positive, negative	label for traces	/	1

Table 4.7: Updated encoding scheme for the attributes of C5.0.

4.6 Support for Multiple Rules

So far, we assumed that *only one kind* of composite event can be triggered by a single rule at a time. However, in many real-world applications the number of composite event types of interest is likely to be greater than one; for example, in the usual environmental monitoring scenario we may be concerned about being able to detect both the event `Fire` and the event `Flood`.

As an expansion of our work with C5.0, we tried to enable the algorithm to deal with histories containing multiple types of composite events instead of only one.

Being C5.0 a classification algorithm, in order to achieve this goal it was basically sufficient to employ a *different label* for each different composite event appearing in the history. Therefore, given N_{rules} rules leading to as much composite events, the "positive" label for C5.0 is simply replaced with N_{rules} labels ("CE_1", "CE_2", ... "$CE_{N_{Rules}}$") and the objective is now to correctly classify traces of events leading to *every* possible composite event type (while labeling traces as "negative" only if they do not lead to any composite event).

In these scenarios, the main issue consists in the fact that more than one composite event may be recorded at the same time, at the end of the same trace (even if the triggering rules are different); if this is the case, then how should the trace be classified?

As an example, consider the following trace, with $U = \{E1, E2, E3\}$, $D_{A1} = \{x, y, z\}$, $A = 1$, $N_{rules} = 2$:

```
E1[A1=x]@1, E1[A1=y]@2, E2[A1=z]@3, E3[A1=z]@4, CE1@5, CE2@5
```

Where $CE1$ is triggered by the rule

```
[Rule1:
EventConstraints:[E1[A1=x]]]
```

and $CE2$ by:

```
[Rule2:
EventConstraints:[E1[A1=y]]]
```

Here, the composite events `CE1@5` and `CE2@5` are recorded at the end of the same trace and at the same timestamp; currently, we do not have a criterion to choose between $CE1$ and $CE2$ labels, so we are unable to encode the given trace. To model this kind of situations, in which there are multiple *concurrent* composite events, we introduce additional labels representing their simultaneous occurrence. This means that the previous trace is neither labeled as $CE1$ nor as $CE2$ but as a *third* kind of composite event, $CE1 + CE2$:

```
E1[A1=x]@1, E1[A1=y]@2, E2[A1=z]@3, E3[A1=z]@4, CE1+CE2@5
```

The new composite event can be viewed as an *independent* event triggered by the *composite rule*:

```
[Rule1+Rule2:
EventConstraints:[E1[A1=x], E1[A1=y]]]
```

It is important to underline the fact that $CE1 + CE2$ represents a situation which is distinct from both $CE1$ and $CE2$, since it expresses the *contemporary occurrence* of $CE1$ and $CE2$. For this reason, labeling a trace leading to $CE1 + CE2$ as simply $CE1$ or $CE2$ (indicating that only one out of the two events is taking place, in mutual exclusion) is conceptually wrong, since it would not fully represent the real situation taking place.

In order to generate a proper dataset for C5.0 in a context characterized by multiple concurrent rules, it was never needed to modify the encoding scheme already presented (except for class labels), nor the splitting procedure adopted with traces containing type repetitions. However, it was necessary to perform some changes in the *classification* procedure instead.

About that, it should be remembered that, when dealing with a single composite event, we chose to classify a trace with repetitions as "positive" if *at least one* sub-trace obtained from the split was classified as "positive" by C5.0. In the new scenario, there is not only a single "positive" label and hence the classification strategy has to be adjusted. In particular, when sub-traces get classified with labels of n different composite events, we classify the original trace with a label which is the *union* of all the labels observed for its sub-traces (i.e., $CE1 + CE2 + ... + CEn$). It is easy to understand the reason for this choice, since, with reference to the previous example, the classification of the following sub-traces is entirely *correct* (if only considering sub-traces and not the original trace):

```
E1[A1=x]@1, E2[A1=z]@3, E3[A1=z]@4, CE1@5
E1[A1=y]@2, E2[A1=z]@3, E3[A1=z]@4, CE2@5
```

However, the real label that has to be assigned to the original trace remains $CE1 + CE2$. A trace is therefore classified simply as $CE1$ or $CE2$ only if there is at least one sub-trace classified as $CE1$ ($CE2$) but none of the other class ($CE2$ or $CE1$, respectively). For instance, the following set of sub-traces:

```
E1[A1=x]@1, E2[A1=z]@3, E3[A1=z]@4, CE1@5
E1[A1=y]@2, E2[A1=z]@3, E3[A1=z]@4, negative
```

gets (wrongly) classified with the label $CE1$.

Lastly, if the label $CE1 + CE2$ directly appears in the set of classified sub-traces, that label is employed for the original trace; therefore, the following set of classified sub-traces lead to a correct prediction:

```
E1[A1=x]@1, E2[A1=z]@3, E3[A1=z]@4, CE1@5
E1[A1=y]@2, E2[A1=z]@3, E3[A1=z]@4, CE1+CE2@5
```

[Table 4.8] below summarizes and exemplifies the classification strategy just described.

Labels contained in sub-traces	**Classification**
CE_i	CE_i
CE_i, "negative"	CE_i
CE_i, CE_j, ..., CE_n	$CE_i + CE_j + ... + CE_n$
CE_i, CE_j, ..., CE_n, "negative"	$CE_i + CE_j + ... + CE_n$
CE_i, CE_j, ..., CE_n, $CE_i + CE_j + ... + CE_n$	$CE_i + CE_j + ... + CE_n$
CE_i, CE_j, ..., CE_n, $CE_i + CE_j + ... + CE_n$, "negative"	$CE_i + CE_j + ... + CE_n$
$CE_i + CE_j + ... + CE_n$	$CE_i + CE_j + ... + CE_n$
$CE_i + CE_j + ... + CE_n$, "negative"	$CE_i + CE_j + ... + CE_n$
"negative"	"negative"

Table 4.8: Classification with multiple concurrent composite events.

As in the case of a single composite event, a *negative* classification only happens if every sub-trace is classified as "negative". However, the possibility to get false negatives is higher with many composite events because, for example, $CE1$ can be incorrectly labeled as $CE2$ or $CE1 + CE2$, and both the predictions are false negatives w.r.t. the $CE1$ label.

Nonetheless, this kind of situations was rarely encountered in practice and C5.0 ultimately showed to be a strong tool to employ even in presence of multiple composite events.

As a conclusive note, estimating misclassification costs in a multi-label context becomes more complicated and computationally demanding; to

keep things simple while allowing to still benefit from the use of the technique, we opted to only estimate costs involving a composite event label and the negative label. In other words, for each composite event, we employed misclassification costs of the type:

```
CE1, negative: k1
CE2, negative: k2
CE1+CE2, negative: k12
```

while we avoided to estimate costs like:

```
CE1, CE2: h
CE1, CE1+CE2: q
```

Chapter 5

Experimental Results

In this chapter we test the architecture developed with the aim of finding out how much good it is at performing the learning tasks assigned to it (i.e., how much *accurate* it is at classifying traces of events).

For this purpose, we first analyze the overall system accuracy alone and, in a second step, we compare the results obtained with an ad-hoc learning algorithm, iCEP [23], which represents the state of the art in the field of automated CEP rule learning.

To accomplish this task, we require:

- a tool to generate histories of events (both for training C5.0 and for evaluating it) based on *controllable parameters*, since we want to find out how the algorithm performs under a broad variety of circumstances as well as to detect which are the most influential parameters affecting it;
- proper *metrics* that capture and precisely measure the accuracy of the system.

About the first point, we exploited a slightly modified version of the iCEP framework for synthetic benchmarking [23]as an artificial *history generator*: first, a training history H_T of primitive events is randomly generated, guided by an *oracle rule* R which is used to detect all the composite events in it; then, an *evaluation history* H_E, composed by events that are different from the ones in H_T, is also produced according to R.

At this point, its is possible to quantitatively compare the accuracy of the decision tree DT built by C5.0 against the oracle rule R as measured on H_E. To do that, we employ the well-known *precision*, *recall* and *F-score* metrics, already presented in [Section 1.4], and this satisfies the second point as well.

In our case, the *precision* of the algorithm can be seen as the fraction of composite events captured by DT which were also captured by R, whereas the *recall* is the fraction of composite events captured by R which were also captured by DT.

Operatively, C5.0 itself offers the possibility of conducting an evaluation of the decision tree, either on the training dataset itself or, if provided, on a different *evaluation dataset*. In order to do that, the algorithm selects every instance contained in the evaluation dataset and labels it according to the already computed decision tree; then, it simply compares the predicted label with the real one (i.e., the one observed in the evaluation dataset).

The result of this process is a *confusion matrix* that allows a visualization of the different error rates for each class and that makes it easy to calculate the needed quality metrics.

In the confusion matrix:

- each *row* represents the number of instances in a predicted class, while
- each *column* contains the number of instances in an actual class.

Since in our case there are only two possible class labels ("positive" and "negative"), the matrix is 2x2, and – with reference to the "positive" label - each cell contains the total number of true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN), respectively.

An example of a possible confusion matrix produced by C5.0 is showed below:

717(TP)	299(FP)
3769(FN)	6215(TN)

Table 5.1: Example of a C5.0 confusion matrix.

With the numbers above, it is possible to calculate precision, recall and F-score.

$$\text{Precision} = \frac{TP}{TP+FP} = 0.705709; \text{ Recall} = \frac{TP}{TP+FN} = 0.159831;$$

$$\text{F-score} = 2 * \frac{precision*recall}{precision+recall} = 0.260632$$

Our objective is to perform a similar task with *traces* of events extracted from the evaluation history. We cannot directly rely on the confusion matrix obtained by employing an evaluation dataset generated in the same way of the training dataset, because the method that we use to classify traces (described in [Section 4.3]) do not proceed on a per instance basis and we would lose all the advantages provided by the CLASSIFY_TRACE procedure thus getting a lot of misclassifications consisting of false positives. Therefore, we take the following steps in order to evaluate the decision tree generated by C5.0:

1. First, we extract all the available traces from H_E according to the window size W.

2. Then, for each trace we call the procedure CLASSIFY_TRACE to get the predicted label for the current trace, according to the decision tree DT;

3. At this point it is possible to acknowledge if the classification lead to a TP, FN, FP or TN by simply comparing the predicted label with the real label of the trace.

In practice, we modify the original architecture developed and showed in [Figure 3.1] in such a way that it receives an evaluation history instead of generic "unseen traces", and we add a component, called *Evaluator*, which is in charge of conducting the evaluation tasks just described, providing in output precision, recall and F-score for the current experiment.

The modified architecture employed for the evaluation is showed in [Figure 5.1] below.

Since we decided to compare the performance of C5.0 against iCEP, it is important to underline that in order to allow a fair confront both the algorithms were trained and evaluated on the *same histories*, containing composite events generated from the same *oracle rules*.

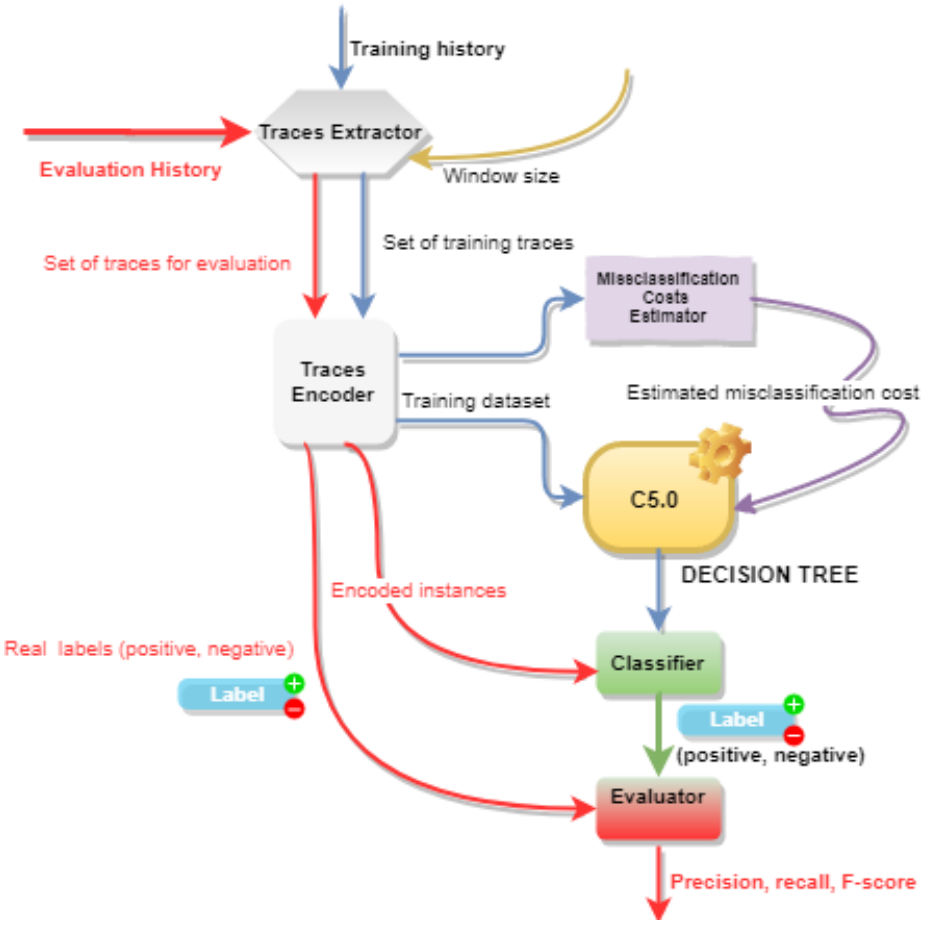

Figure 5.1: Overview of the architecture employed for the evaluation.

The remainder of this chapter is organized as follows: in [Section 5.1] we describe the main parameters involved in the experiments conducted that guided the generation of synthetic histories and of which we investigated the impact on the performance of C5.0, whereas in [Section 5.2] we present the actual results obtained for each of such parameters. Lastly, in [Section 5.3] we proceed to comment the results obtained.

5.1 Experimental Setup

In this section we present all the relevant parameters involved in the tests conducted.

Given the high number of different parameters, exploring the entire parameter space is clearly unfeasible; therefore, we were forced to fix some of them in order to reduce such large space. The following parameters were kept constant and constitute the basic environment of every experiment:

- *Distribution of event types*: probability distribution of event types in the history; a *uniform* distribution has been employed in every scenario, both for training and evaluation histories.
- *Number of positive traces*: it corresponds to the number of composite events included in the history. In our experiments, this parameter is fixed at 3000 positive traces for training histories and at 20000 for evaluation histories; typically, a higher number of positive traces in the training history brings little improvements in terms of accuracy whereas a lower number is likely to negatively impact on the learning capabilities of C5.0.
- *Number of negative traces*: the number of traces that do not contain a composite event. We employed 30000 negative traces for training histories and 100000 negative traces for evaluation histories. As a consequence, a fixed *unbalancing proportion* of 1:10 characterizes the training dataset.
- *Window size* W: constant and fixed at 10 time units.
- *Sequence probability* P_{seq}: the probability of a *relevant* event type included in rule R to be subject to a sequence constraint. In our experiments, it is fixed at 1.0, meaning that *every* relevant event in R is involved in at least one sequence constraint.
- *Number of sequence constraints* N_{seq}: depends on P_{seq} and it is equal to the number of relevant types N_{rel} [Section 5.1.2], since $P_{seq} = 1$. In other words, in all the tests conducted the rule R *always* contains sequence constraints.

- *Distribution of comparison operators in constraints*: for each constraint that involves a comparison with a threshold value (namely, selections, negations and aggregates), we assume that the percentage of constraints containing an equality comparison is fixed at 20%, whereas the percentage of constraints characterized by "$\geq$" and "$\leq$" comparisons is 40% in both the cases.
- *Number of folds* N_{folds} employed to estimate misclassification costs, fixed at 3.
- *Range* of possible misclassification costs, fixed at [1,20].

For a quick reference, the constant parameters are summarized in [Table 5.2].

Parameter	**Value**
Distribution of event types	Uniform
Positive traces in training history	3000
Negative traces in training history	30000
Unbalancing proportion in training history	1:10
Positive traces in evaluation history	20000
Negative traces in evaluation history	100000
Window size	10
Sequence probability	1.0
Number of sequence constraints	N_{rel}
Percentage of "=" constraints	20
Percentage of "$\geq$" constraints	40
Percentage of "$\leq$" constraints	40
Number of folds	3
Range for misclassification costs	[1,20]

Table 5.2: Fixed parameters adopted.

On the other hand, we argue that the following parameters, showed in [Table 5.3], are the most interesting to investigate both from a theoretical and an applicative point of view, since by manipulating them it is possible to simulate a broad variety of different scenarios. In fact, the values

employed for the tests were properly chosen and are meant to recreate the settings of many plausible real-world CEP applications.

Parameter	Symbol	Possible values
Number of event types	N_{types}	5, **15**, 30
Number of relevant types	N_{rel}	2, **3**, 5
Number of values	N_{val}	5, **50**, 500
Number of selection constraints	N_{sel}	0, **1**, 2, 3, 4
Number of attributes	N_{att}	0, **2**, 4, 6, 8
Number of negation constraints	N_{neg}	**0**, 2, 3, 5
Number of aggregate constraints	N_{agg}	**0**, 2, 3, 5
Number of rules (composite events)	N_{rules}	**1**, 2-10
Allow collisions of CEs	/	true, **false**
Frequency of noisy events	F_{noise}	3,9,**30**

Table 5.3: Parameters tested.

In the following sections, we focus on each of these parameters separately, describing them in greater detail.

5.1.1 Event Types

The *number of event types* N_{types} is the total number of different existing event types in the system; as already explained, we assume that the *universe* U of event types is finite and fully known; therefore, the number of event types $N_{types} = |U|$ is an important parameter to consider since it may significantly vary depending on the applicative context under exam.

5.1.2 Relevant Types

The *number of relevant types* N_{rel} is the number of types included in a *conjunction* constraint of R; the occurrence of relevant types is required in order to make a trace positive.

5.1.3 Attribute Values

The *number of values per attribute* N_{val} indicates *how many* possible values an event attribute may assume. In our testing scenario, event attributes can take values from finite domains and N_{val} express the *cardinality* of such domains. More precisely, attributes may take integer values included in the range $[0,N_{val}[$, so they are treated and can be encoded as *continuous* C5.0 attributes.

5.1.4 Selection Constraints and Number of Attributes

The *number of selection constraints* N_{sel} is the number of constraints on event attributes and it is the same *for each relevant type* included in rule R. This parameter is closely related to another parameter, N_{att}, that indicates the total number of attributes of each event type (not only those subject to constraints); this is because, in all our experiments, we assumed that $N_{att} = 2 * N_{sel}$. In other words, for each type included in R, the number of attributes of that type subject to a selection constraint is always half of the total number of the existing attributes, but this is just an assumption that simplifies the parameter space and not necessarily the case in general.

5.1.5 Negations

This parameter refers to the number of negation constraints N_{neg} contained in rule R.

5.1.6 Aggregates

The number of *aggregate* constraints in R is indicated with N_{agg}. In the tests performed, we aimed at including an equal total number of aggregate constraints of each kind (minimum, maximum, sum and average).

Only for aggregates, we opted to change the windows size, doubling it to 20. The reason behind this choice is to better amplify the impact of aggregate constraints in traces; in fact, with greater window sizes aggregate functions are calculated on a higher number of event notifications and therefore they likely to be more influential overall.

For this reason, we tested both the usual scenario with $W = 10$ and the new scenario with $W = 20$, which represents a more problematic working condition both for C5.0 and for iCEP.

5.1.7 Multiple Rules

To study the impact of including multiple kinds of composite events and rules, we distinguished between two different scenarios: in one case, we allowed *concurrent* composite events to happen at the same time, while in the other case we did not.

This means that we introduce an additional parameter, N_{rules} indicating the number of different rules in the history, and a boolean parameter that specifies if such rules may or may not overlap and be triggered concurrently.

5.1.8 Noise

One interesting thing about the iCEP history generator is that it allows to simulate *uncertainty*, which characterizes most real-world CEP scenarios and typically consists of *erroneous* or *incomplete* data streams (for example, this could happen because of inaccuracies in sensor readings, or a network failure). The introduction of a noticeable degree of *noise* adds more solidity to the results obtained, since it better recreates more realistic conditions that could be faced in practical applications.

The parameter F_{noise} denotes the *frequency between noisy events* and adds *uncertainty* to event traces, since it artificially introduces randomly generated events unrelated to the oracle rule R. By varying F_{noise}, we basically tell the history generator to introduce an erroneous event once every F_{noise} events.

Lower values of F_{noise} indicate a *higher* presence of noise within the history, and vice-versa; in our case, we decided to test three values for F_{noise}: 3, 9 and 30, representing different working conditions characterized by respectively high, medium and low noise. Being the window size W fixed at 10, with $F_{noise} = 3$, several noisy events are included within *the same* trace, for each trace; with $F_{noise} = 9$ every trace contains one noisy event on average, whereas with $F_{noise} = 30$ only one out of three traces contains a single noisy event.

5.2 Experimental Results

The actual results of the tests carried out on C5.0 are presented in the current section.

To perform the experiments, we adopted two different techniques:

- **Method 1.** The first strategy defines a *default scenario* and investigates the impact of each parameter separately, varying only one of them at a time (i.e., for each experiment); in this way, we do not explore much of the parameter space but it is easier to quantify the influence of a single parameter on the results obtained.
- **Method 2.** This method tests all the parameters selected in any possible combination, which allows to explore a significantly larger portion of space at the cost of testing single parameters less accurately.

It is important to underline that **Method 2** could not be applied to all the parameters of [Table 5.3], because some of the possible combinations of parameters do not make sense and could not be tested (this includes, for example, a scenario characterized by $N_{types} = 5$, $N_{rel} = 3$ and $N_{neg} = 3$). Therefore, the scope of this method has been limited to the first four parameters of [Table 5.3] (N_{types}, N_{rel}, N_{val}, N_{sel}), whereas it was possible to apply **Method 1** to all the other parameters as well, due to a proper choice of our *standard scenario*. The values assumed by the parameters in the standard scenario are the ones marked in boldface in [Table 5.3].

To allow a fair comparison between the two methods, we performed an equal number of experiments for the four parameters mentioned above. Based on the number of values tested for each parameter, with **Method 2** we needed to perform 5*3*3*3=135 tests in order to run one experiment for each different combination of the parameters; therefore, with **Method 1** we performed 45 tests for each value (27 in the case of selection constraints), that makes for a total of 135 runs as well.

Since **Method 2** was not sufficient to test every parameter, the results presented in this section have been obtained by employing **Method 1**. Nonetheless, all the results obtained with the alternative testing method can be found in [Appendix B].

To represent the statistical distribution of the results collected, we decided to employ *boxplots*. Thanks to this powerful descriptive statistics tool, we can study and draw conclusions on the results, and in particular:

- we can view the *median* as a "middle" value which is preferable to the mean since the distribution of the results may be skewed;
- the *Interquartile range* (IQR) measures the *variability*, or *statistical dispersion* of the data, which in turn allows us to evaluate the robustness of the results collected and to identify possible *outliers* that may represent interesting situations for us (i.e., particular settings that create troubles for C5.0). In our case, the IQR includes 50% of the values of the distribution in the upper and lower quartiles.

Informally speaking, the higher is the variability of the data, the harder is for us to draw meaningful conclusions. Luckily, given the reduced spread of the data observed in almost every tested scenario, we argue that the number of tests performed is indeed sufficient to provide statistically significant results in most of the cases.

As a last note, we provide a link [6] where the reader can download all the required .csv files used to obtain the plots included in this section, as well as the ones of the Appendices, in order to allow data reproducibility. We do not provide the source code of our architecture (which was developed in Java), but we do include all the logs produced by our application while running the tests.

In the following subsections, we will present and comment the results obtained with reference to each parameter took into account, in the same order as presented in [Section 5.1]. To improve readability, in the current section we only show plots containing F-score measurements; however, the full plots — also including precision and recall — are available in [Appendix A]. Lastly, [Appendix C] contains other miscellaneous but interesting plots showing the efficacy of the misclassification cost estimation strategy, the correlation of the parameters with the number of errors introduced and many more.

5.2.1 Event Types

About event types, C5.0 was able to provide high steady performance in presence of 15 or 30 different types, with an average F-score close to 1.0 and reduced spread; with 5 events, the dispersion is higher but the median remains close to 1.0. As further explained in [Appendix C], the reason for the higher variability of the results in presence of 5 types is due to the fact that with less types – while keeping fixed windows of 10 – the number of type repetitions and the consequent number of *errors* introduced in the training dataset is significantly higher. Even if the estimation of misclassification costs makes up for this issue, in some isolated cases the technique is not efficient enough, and this explains the wider IQR as well as the minimum, below 0.8, reached when dealing with 5 event types.

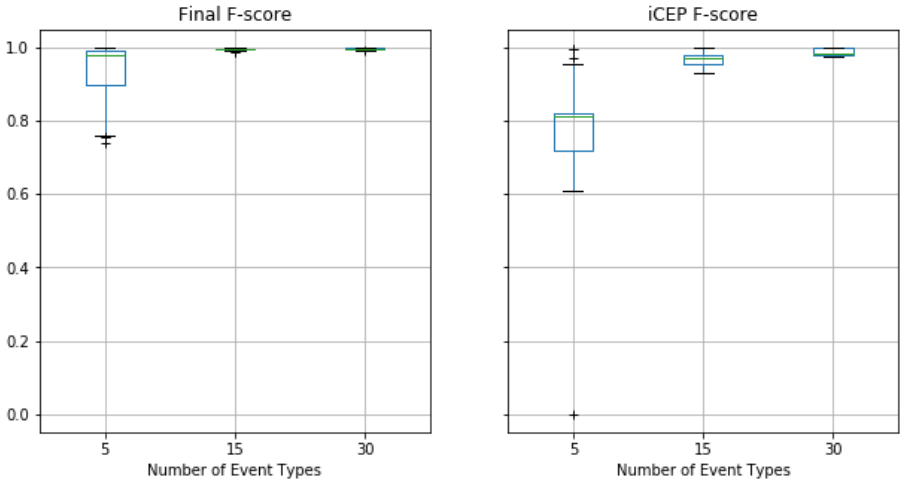

Figure 5.2: C5.0 and iCEP F-scores based on the number of event types.

On the other hand, iCEP performs as much as well with 15 and 30 types, but its F-score drops in the 0.8 area with 5 types, meaning that the algorithm encounters issues in this situation, not detecting a significant number of composite events. Apparently, the presence of a low number of event types compared to the size of the window is problematic also for iCEP, which suffers from type repetitions way more than C5.0.

5.2.2 Relevant Types

This parameter does not negatively influence the performance of C5.0, which is constantly excellent for every tested value. The same goes for iCEP, which provides average F-scores above the 0.95 area; however, despite being good, the algorithm is slightly worse than C5.0 since the results contain more variability and the medians are lower. Noticeable is the case of 2 relevant types, in which we can observe the lower quartile ending below 0.9, a minimum value below 0.8 and several outliers below 0.6, whereas for C5.0 the IQR is contained and even outliers are above 0.9, meaning that no situation encountered caused significant troubles to the algorithm, as it emerges in [Figure 5.3].

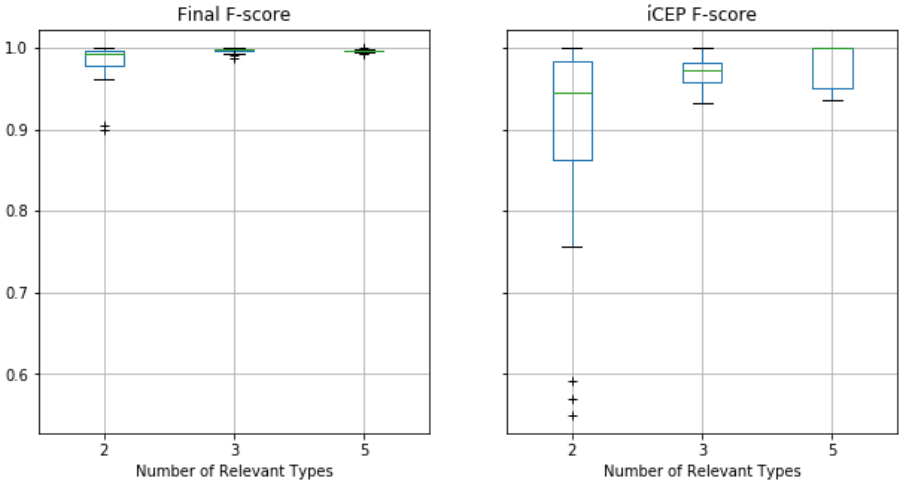

Figure 5.3: C5.0 and iCEP F-scores based on the number of relevant types.

The excellent results yielded by C5.0 demonstrate the efficiency of the attributes `Event`$_x$`Detected`, which in fact were created to capture relevant events. Such attributes are often exploited in decision nodes by C5.0, and we verified that the outcome of the tests in such nodes is almost always in accordance with the oracle rule. This means that C5.0 captures *conjunction* constraints with ease, detecting all the relevant types even when their number is as low as 2, which in general represents a difficult working condition, as also confirmed by the behavior of iCEP which shows a progressively improving performance as the number of relevant types increases.

5.2.3 Attribute Values

As with relevant types, also the number of attribute values do not to cripple C5.0 in any tested scenario. The performance is once again very satisfying with an average F-score always above 0.99. iCEP is once again only slightly worse, with a steady F-score around the 0.96 area and a bit higher variability as well as lower minimums.

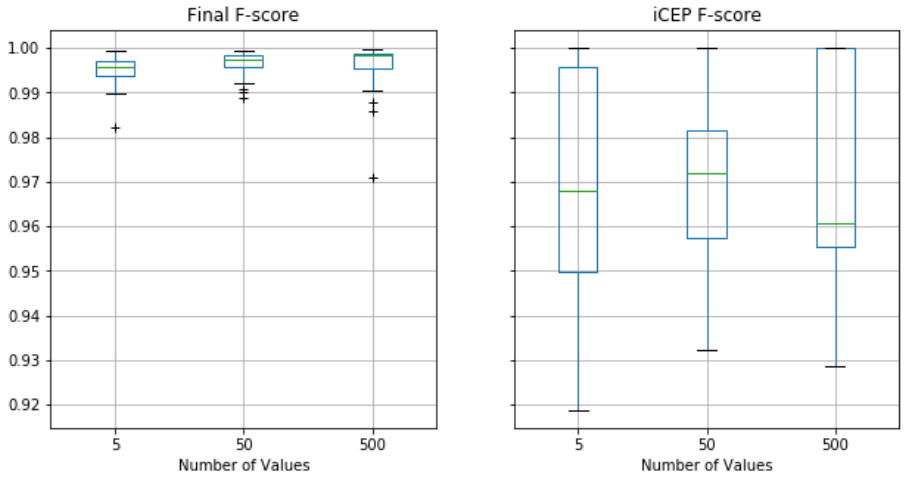

Figure 5.4: C5.0 and iCEP F-scores based on the number of attribute values.

About C5.0, the excellent results obtained should not be a surprise: as remarked many times, in our tests we assume *numeric* attributes (real or integer numbers) for the events and we encode them as *continuous* C5.0 attributes. Since the algorithm generally supports very well this type of attributes, it is easy to see why the number of possible values in their domain does not influence the performance in the slightest. In fact, the algorithm cannot tell the difference and manages the attributes in the same way regardless of the number of values that they may assume.

In addition, the context of our standard scenario is already a favorable situation for C5.0, so we can understand why the results yielded are so much good in this case.

5.2.4 Selection Constraints and Number of Attributes

The number of selection constraints, along with the number of attributes of each event type which is closely related to it since $N_{att} = 2 * N_{sel}$, seems not to affect neither C5.0 nor iCEP, which both provide comparable and almost perfect F-scores and the robustness of the results is confirmed by the very small IQR and no outliers in both the cases.

Since the number of attributes belonging to an event is tied to the number of selection constraints, in [Figure 5.5] we only show the plots relative to the latter ones, because obviously the plots for the other parameter would have an identical shape.

By inspecting the decision trees produced by C5.0, we could realize why the algorithm is so much accurate; in fact, the attributes $\texttt{Event}_x\texttt{Attribute}_k$, when employed in decision nodes, are always in accordance with the oracle rule, thus successfully capturing the required selections.

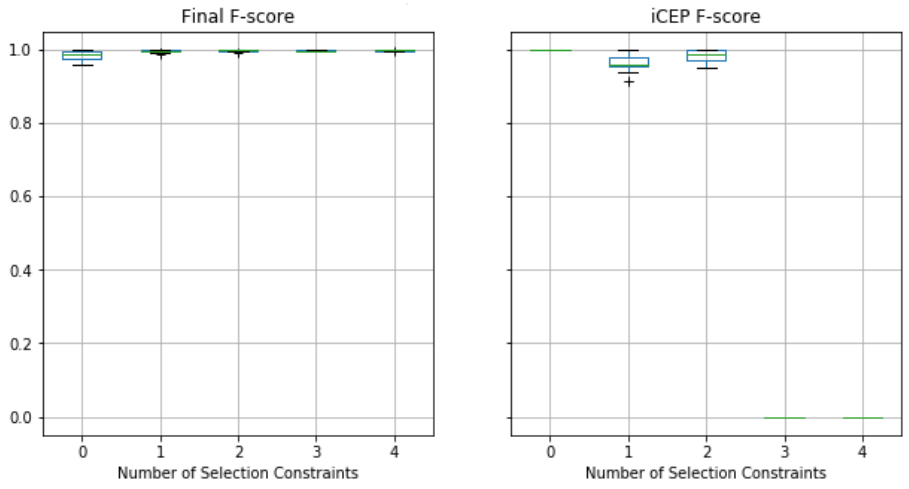

Figure 5.5: C5.0 and iCEP F-scores based on the number of selection constraints.

Unluckily, it was impossible to test iCEP in presence of any number of selection constraints higher than 2 because the algorithm runs out of memory and it is not able to finish the computation. On the other hand, the memory consumption and running time of C5.0 were not significantly influenced by the value chosen for this parameter. This means that, while

C5.0 supports any number of selection constraints in an efficient way, iCEP can only handle situations characterized by at most 2 selections because its computational complexity and memory requirements *prevent* us to even try to test the algorithm in these situations.

5.2.5 Negations

About negations, we observe that their presence makes it even *easier* for C5.0 to detect composite events; in fact, its F-score is almost perfect regardless of the number of negations introduced and the variability is practically non-existent.

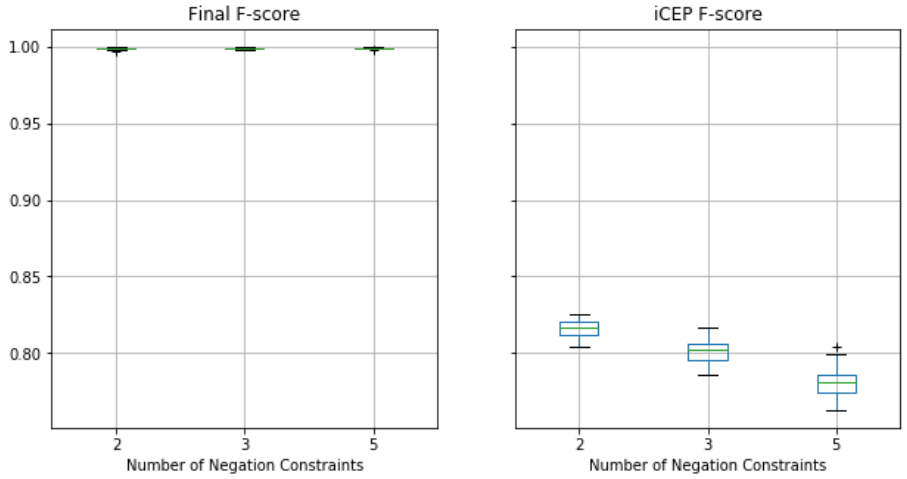

Figure 5.6: C5.0 and iCEP F-scores based on the number of negation constraints.

The explanation for the particularly satisfying performance of C5.0 can be found by looking at the decision trees built by the algorithm when dealing with negation constraints. In fact, we observe that the decision nodes often exploit attributes of the kind $\texttt{Event}_x\texttt{Detected}$, both to detect relevant types ("true" branches leading to "positive" labels, or the opposite) *and* negated types ("true" branches leading to "negative" labels, or the opposite). In fact, the history generator tends to provide traces in which negated types are required to be *absent* from the trace in order to trigger the composite event, so there is less focus on the *value* of their attributes while their *presence* becomes particularly discriminating. A possible prompt for future work about negations could be to extend the generator to produce a higher number of cases in which negations also involve event *attributes*, but we argue that the results yielded by C5.0 would not significantly change thanks to the splitting attributes of the kind $\texttt{Event}_x\texttt{Attribute}_k$.

On the other hand, it has to be underlined that iCEP constantly fails at dealing with this operator, missing all the occurrences of complex events in which negations are involved. As a consequence, the precision of the algorithm drops in the 0.65 area resulting in F-scores around 0.8, which is considered low for our standards and in any case much worse than C5.0. Furthermore, the situation gets progressively worse by increasing the number of negations included in the oracle rule, meaning that it is precisely this operator that is causing the performance to degrade.

5.2.6 Aggregates

As already anticipated, for aggregate constraints we tested two different working conditions: in one case we kept the original window size ($W = 10$) and in the other case we doubled it ($W = 20$).

If we choose not to modify the window size, the results remain good both for iCEP and for C5.0, with average F-scores above 0.95 even in presence of aggregates, as showed in [Figure 5.7].

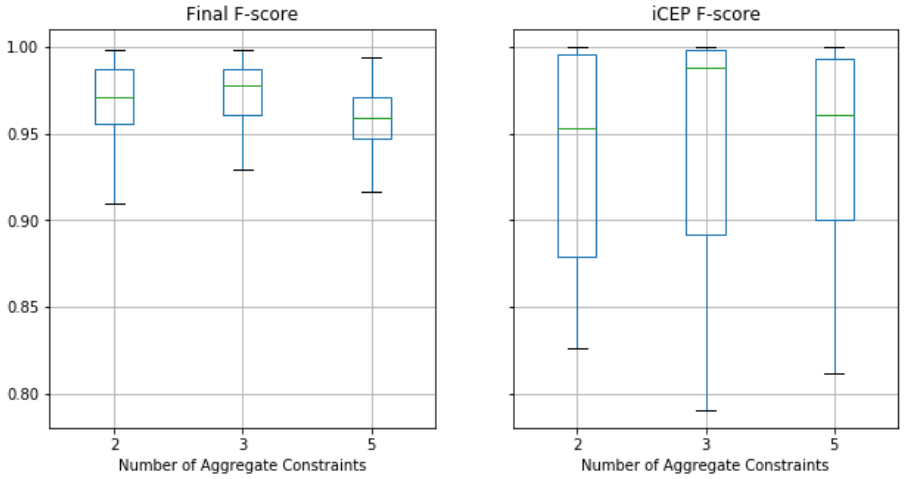

Figure 5.7: C5.0 and iCEP F-scores based on the number of aggregate constraints with $W = 10$.

We explain this fact considering that, by keeping $W = 10$ *in the standard scenario*, which is characterized by a universe of 15 event types, the number of repeated types – and therefore, of events involved in aggregate functions – is likely to be very low.

In particular, C5.0 works very well in the standard scenario adopted and it does not even need to employ misclassification costs in order to provide a more than satisfying performance. Nevertheless, it is possible to observe that aggregates do make the results significantly worse than usual, suggesting that they have an impact on the performance of C5.0.

Following this line of thought, we aimed at *amplifying* the effect of aggregate constraints by increasing the window size, and therefore, the number of (repeated) types in traces involved in some aggregate function.

The intuition turned out to be correct, as with $W = 20$ we observe a noticeable drop of the F-score provided by C5.0 (going in the 0.8-0.9 area), as showed in [Figure 5.8].

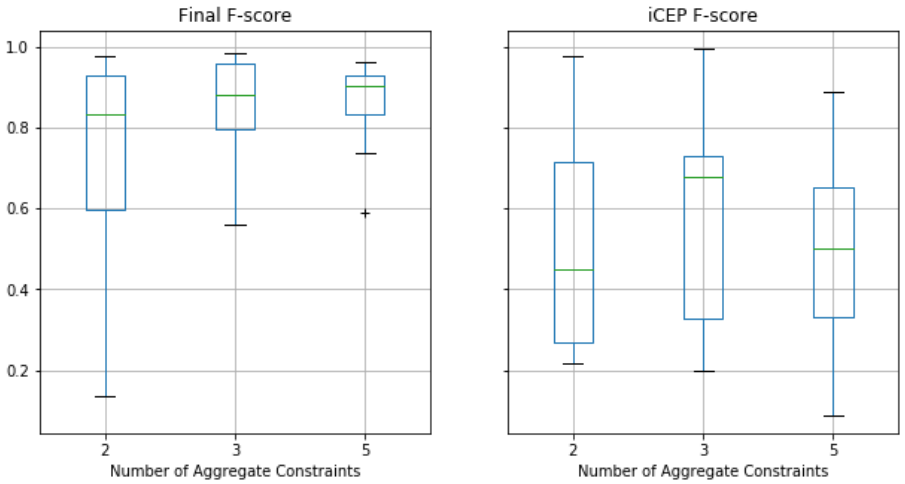

Figure 5.8: C5.0 and iCEP F-scores based on the number of aggregate constraints with $W = 20$.

If aggregate constraints, when more influential, constitute a problem for C5.0, on the other hand they completely cripple iCEP, which could not deal with them. In fact, the F-score yielded by the algorithm is terribly low, around 0.6 in the best case and below 0.4 in the worst case, meaning that aggregates are almost never captured by the algorithm.

C5.0 is able to maintain decent results thanks to the attributes that have been crafted *specifically* for aggregate constraints (namely, $\texttt{Event}_x\texttt{Attribute}_k\texttt{Minimum}$, $\texttt{Event}_x\texttt{Attribute}_k\texttt{Maximum}$, ...). This is confirmed by the fact that, if we look at the decision trees built by C5.0 in presence of aggregates, such specific attributes are usually also the ones more frequently selected for the splits (i.e., they often appear in decision nodes). However, they do not work perfectly as showed by the performance of C5.0, significantly lower than usual; in addition, it is possible to observe a trend according to which the algorithm gets progressively better as the number of aggregates increases. This reinforces the hypothesis that C5.0 is less accurate when it comes to precisely detect aggregates, but if the number of aggregates is high enough and the algorithm has

more attributes available to discriminate positive traces from negative ones, then the situation sensibly improves.

As a last note, we should not disregard the fact that increasing the window size also implies an higher number of false positives added to the training dataset because of the encoding model (as the number of type repetitions is greater). Therefore, the strategy of increasing W do not *fully* isolate aggregate constraints and also other dynamics are involved behind the reduced performance of C5.0.

5.2.7 Multiple Rules

Surprisingly, by increasing the number of rules included in the training/evaluation histories the performance of C5.0 only seems to be slightly negatively impacted, even in presence of multiple concurrent composite events. The fact that C5.0 is able to deal with multiple rules constitutes a significant advantage if compared to iCEP, which on the other hand was not designed to work in this kind of situations, thus not supporting and envisioning them.

In [Figure 5.9] it can be observed that even introducing 10 (non concurrent) rules in the standard scenario does not cause a significant drop of the F-score in terms of median. On the other hand, [Figure 5.10] shows that 2 or 3 rules with collisions are not a problem for C5.0 either (in this plot, 2 base rules lead to 3 composite rules and 3 base rules lead to 7 total composite rules).

However, in some isolated cases the performance does get slightly impaired because of the increased number of rules. For example, with non-concurrent rules the we can observe a trend of the F-score, which decreases as the number of rules increases, although still remaining very good. The same is not true in the case of concurrent rules, where the performance is almost identical regardless of the number of basic concurrent rules (but it has to be underlined that we did not test any number of basic rules higher than 3 and hence we cannot draw conclusions about more complicated scenarios).

As a side note, in [Figure 5.9] and [Figure 5.10] the F-scores presented refer to the overall F-score of the *system* of rules, which is defined as:

$$F_{1system} = \frac{Precision_{system} * Recall_{system}}{Precision_{system} + Recall_{system}}$$

Where

$$Precision_{system} = \frac{\sum_{n=1}^{N_{Rules}} TP_n}{\sum_{n=1}^{N_{Rules}} TP_n + \sum_{n=1}^{N_{Rules}} FP_n}$$

$$Recall_{system} = \frac{\sum_{n=1}^{N_{Rules}} TP_n}{\sum_{n=1}^{N_{Rules}} TP_n + \sum_{n=1}^{N_{Rules}} FN_n}$$

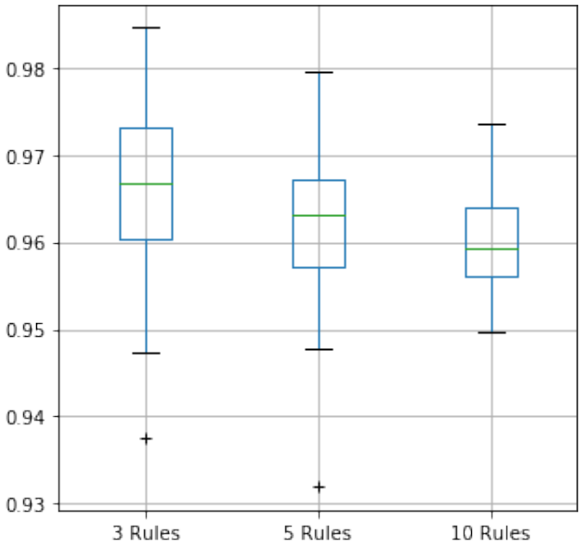

Figure 5.9: C5.0, system F-scores in presence of mutliple rules without collisions.

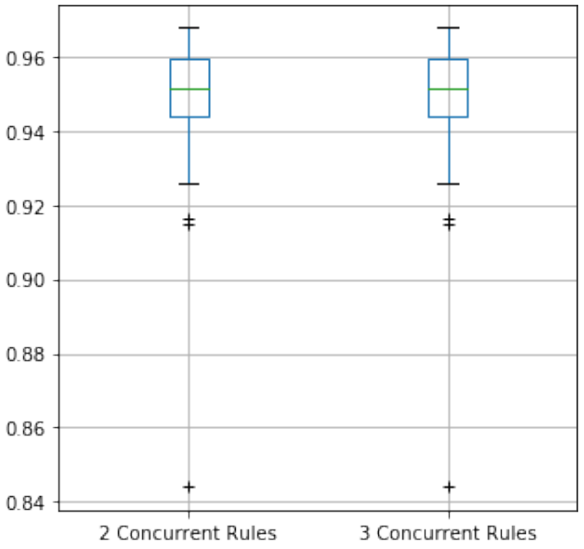

Figure 5.10: C5.0, system F-scores in presence of mutliple rules with collisions.

5.2.8 Noise

By varying the frequency of noisy events, we observe that, in general, both C5.0 and iCEP provide a steady high performance, with F-scores above the 0.95 threshold.

The performance of C5.0, however, seems to improve as the noise reduces: both in terms of medians and minimums, the trend is very clear in the plot below. On the other hand, iCEP do not correlate with this parameter; it even seems to be inversely correlated in terms of medians, which would not make sense per se. However, being the IQR large and given the fact that minimums do not follow a precise trend, we can conclude that iCEP is both *noise-tolerant* and *noise-independent*, whereas C5.0 is only noise-tolerant.

In any case, the important conclusion is that both the systems are able to deal with *uncertainty*, that is a crucial feature or real-world applicative scenarios in which missing or incorrect events are very common.

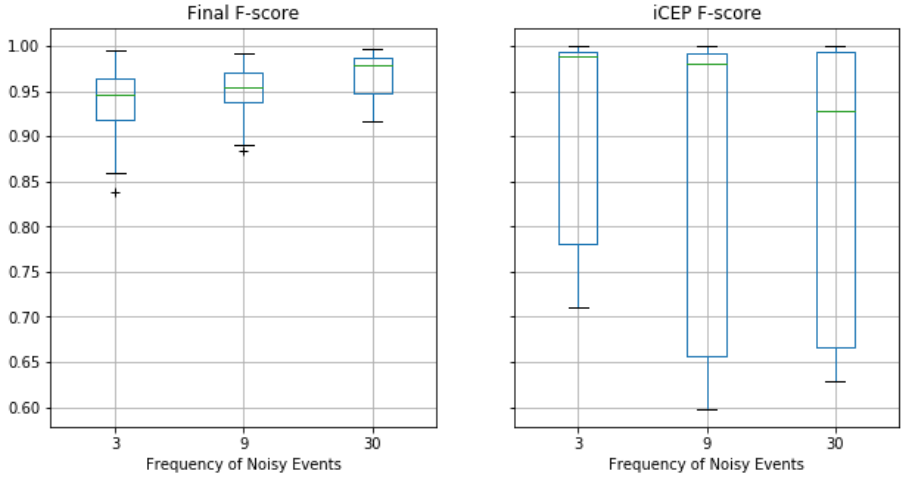

Figure 5.11: C5.0 and iCEP F-scores based on the frequency of noisy events.

5.3 Discussion

From the boxplots presented, it can be observed that – in terms of F-score – C5.0 is always slightly better than iCEP in every experiment conducted, so overall we can conclude that the two algorithms are at least quantitatively comparable. However, in some particular settings (such as when dealing with a low number of relevant types or with the presence of negation constraints) C5.0 is much better than iCEP and therefore, quantitatively speaking, superior.

It is worth remarking that the ability of C5.0 to efficiently detect composite events in presence of *negation* constraints constitutes a big step forward if compared to iCEP. In fact, negations were the most crippling issue for this ad-hoc learning system, as also emphasized many times in [23]. C5.0, on the other hand, handles negations with ease, despite the fact that they are for sure a "problematic" operator.

Even in the only situation in which C5.0 encountered issues, namely in presence of *aggregate* constraints (only with $W = 20$), it still somehow managed to work in a better way than iCEP. In fact, while both the algorithms were influenced by this operator, iCEP was hit more badly than C5.0, which maintained an F-score around 0.8-0.9 (not dramatically low, even if nowhere near being good).

In addition, some scenarios were impossible to test with iCEP due to the prohibitive computational complexity, time and resources demanded by the application; in particular, iCEP could not work in presence of any number of *selection constraints* greater than 2 whereas C5.0 managed to do that without significantly worsening its efficiency or running time.

Another noticeable advantage yielded by C5.0 is its support for *multiple rules* and multiple types of composite events, where we observe a satisfying performance even in presence of concurrent rules and noisy events; on the other hand, iCEP can only learn one rule at a time, for each history.

In conclusion, a comparison between C5.0 and iCEP is summarized in the following [Table 5.4], in which we consider *good* results F-scores higher than 0.95, *mediocre* results F-scores higher than 0.9 and *bad* results F-scores below 0.9 (based on the average of the medians of the boxplots presented).

Parameter	**C5.0**	**iCEP**
Number of event types	good	mediocre
Number of relevant types	good	good
Number of attribute values	good	good
Number of selection constraints	good	good
Number of negation constraints	good	bad
Number of aggregate constraints	good ($W = 10$) bad ($W = 20$)	good ($W = 10$) bad ($W = 20$)
Number of rules	good	not supported
Frequency of noisy events	good	good

Table 5.4: C5.0 and iCEP, final comparison.

Chapter 6

Related Work

This chapter examines some related work in the context of automated CEP rule learning. Currently, it seems that very little research is going on about this topic, which mostly remains a new and unexplored field.

There have been many known attempts to employ Inductive Logic Programming (ILP) [27] for the automated learning of patterns expressed through some kind of first-order logic formalism; for example, OLED [19] allows to learn clauses even in presence of unobserved predicates thanks to a mix of ILP and abductive-logic programming. According to the authors, the system is also able to deal with larger datasets while providing a significant speed-up in training time if compared to other solutions like ILED [34]; furthermore, OLED learns with a *single pass* over the training input without revising already learned clauses, thus making it a good *online* learning tool.

As OLED, also our system adopts a one-pass strategy for learning, even if it does not rely on logic and reasoning but on an entirely independent off-the-shelf machine learning technique (C5.0). While this constitutes a downside with respect to tools that output pattern predicates, since they can be analyzed and modified by domain experts, we argue that decision trees could be successfully employed in online Complex Event Recognition as well, thanks to their ability to deal with very large training datasets without a significant impact on speed and performance, which on the other hand is a common problem among logic-based techniques exploiting ILP.

Another possible advantage of employing an hypothetic decision tree-based CER is represented by its *domain-independence* (since the underlying algorithm is domain-independent as well), as opposed to most of the available work based on logic which is *domain-specific*. For instance, there is plenty of work mostly focused on human activity recognition [35, 18, 32, 19] which was only tested on datasets coming from that particular domain. The same goes for other recent works, concerning traffic management systems [24], maritime monitoring [28] and credit card fraud detection systems [12].

The work that we propose is distant from the approaches cited above, and constitutes an unexplored territory in the CEP field. Even iCEP [23], which we took as a reference point in this thesis, had to employ *ad-hoc learning strategies* to learn complex event patterns; nevertheless, it is probably the closest and most meaningful work for us, as it does rely on supervised learning concepts and "pure" machine learning while avoiding the burden of logical formalism. Despite being more efficient than iCEP, our system do not envision a strategy to learn a single *rule* that can be expressed in a TESLA-like syntax [13], and this is probably the major open issue of possible decision tree-based CER systems, since it would be difficult to employ them in conjunction with current CEP engines which rely on specific event definition languages.

About this, an interesting work is represented by autoCEP [21], an automated tool that generates CEP rules for online monitoring in product manufacturing. autoCEP is conceptually close both to iCEP and the current work but it involves *association rule mining* techniques instead of supervised learning strategies like C5.0. However, while being fast and efficient, autoCEP is both domain-dependent (as it is specifically designed to support industry processes) and still resorts to ad-hoc adaptations of generic learning techniques, while we aimed at building a domain-independent system relying on general-purpose, well-known learning algorithms.

Nonetheless, autoCEP includes an *algorithmic processing* that transforms the rules extracted through association rule mining into a single CEP rule, which is something that we did not develop but it could be a relevant prompt for future work in this sense. In fact, if it were possible to algorithmically process the output decision tree built by C5.0 (or, in alternative, an equivalent *set* of rules obtained from the tree), that would be the starting point to find out the degree of practical applicability of

decision tree-based CER systems with regard to current mainstream CEP engines and languages.

Conclusions

In this thesis we addressed the problem of employing general-purpose Machine Learning tools for the automated detection of patterns leading to composite events within streams of data.

We observed that a novel approach, consisting of decision tree-based classifiers, may be able to provide a feasible solution for the problem. In fact, we showed how it is possible to encode a series of previously collected event traces as a training dataset that can be exploited for learning purposes, given enough assumptions on the underlying model for events and rules which was adopted throughout the course of this work.

The encoding strategy developed, along with the method employed to classify traces of events, apparently succeeded in this task, allowing the well-known C5.0 algorithm to work in the context of CEP. This has been confirmed by many experimental evaluations conducted on the algorithm, in which C5.0 showed respectable performance even when compared with a framework specifically designed to learn from traces through the means of ad-hoc techniques and that represents the current state of the art in the field of automated CEP rule learning.

Our system not only outperformed ad-hoc solutions in almost every tested scenario, but we found out that its range of applicability is also much broader, covering a variety of interesting and complex situations which include, for example, the presence of negation constraints in rules and the presence of multiple kinds of concurrent composite events, triggered by separate rules but included within the same scenario.

Undoubtedly the performance measurements of precision, recall and F-score proved the accuracy and efficiency of the system proposed. Nonetheless, it has to be observed that our strategy remains at a very early stage of development, it is highly dependent on a particular event model which is distant from many mainstream paradigms and this raises concerns about the practical applicability of the system to real-world contexts, which still remains an open issue that we did not investigate.

The problem is even made worse by the fact that decision tree models are intrinsically distant from rule-based CEP engines, so it is not possible to integrate the learning architecture developed "as is" into already existing CEP systems. About this, we argue that a possible prompt for future development could be an extension of the application that algorithmically processes the output decision tree in order to derive a single rule meeting the syntactic requirements of CEP engines.

Lastly, a crucial aspect worth investigating concerns the computational impact of the use of decision trees, which becomes even more important if the idea is to employ them for online learning, characterized by strict throughput and latency constraints. As an offline learning tool, however, the present work showed that decision trees are already adequate, as the time and computational requirements are much more relaxed if compared to an online setting.

Appendix A

Detailed Results

In this appendix we show additional and more precise plots regarding the basic tests performed on C5.0 and iCEP, which include also precision and recall measurements instead of F-score only.

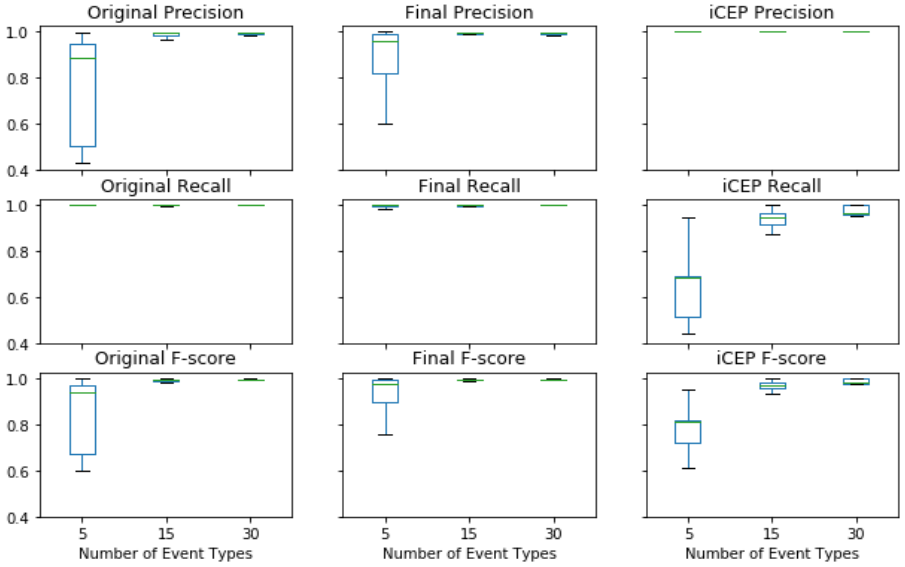

Figure A.1: Detailed plots, number of event types.

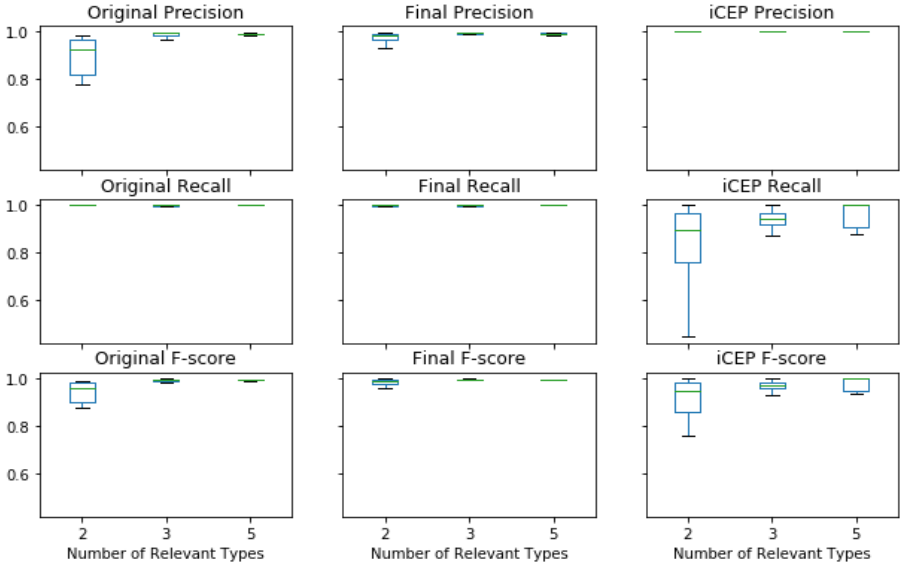

Figure A.2: Detailed plots, number of relevant types.

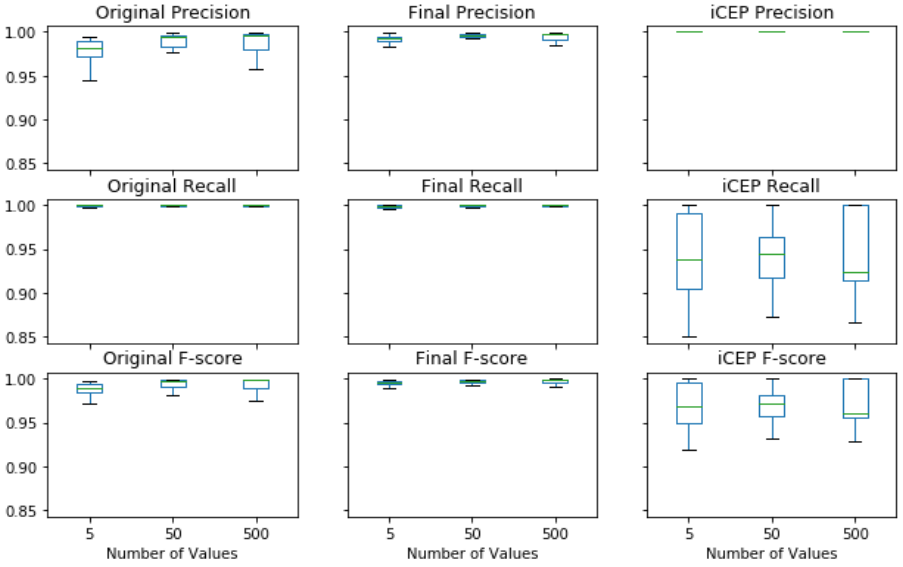

Figure A.3: Detailed plots, number of attribute values.

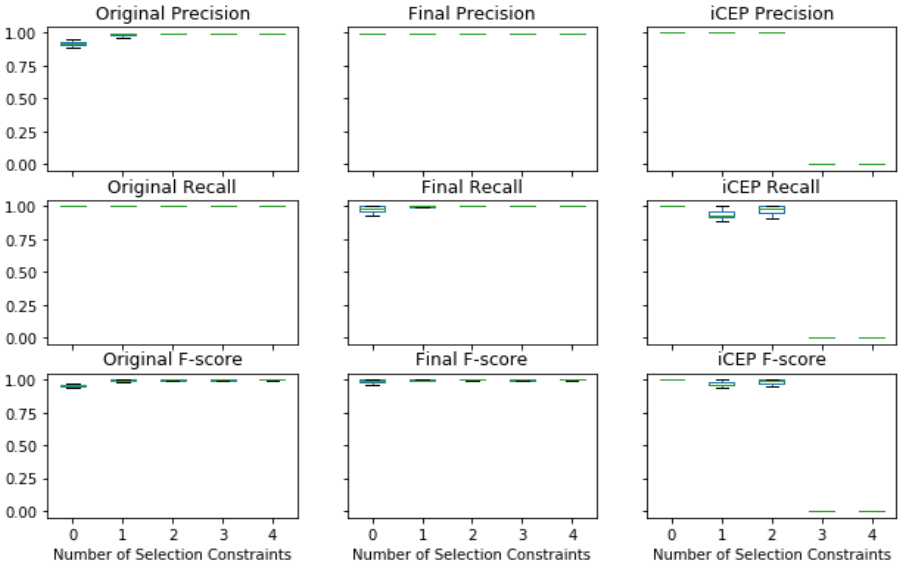

Figure A.4: Detailed plots, number of selection constraints.

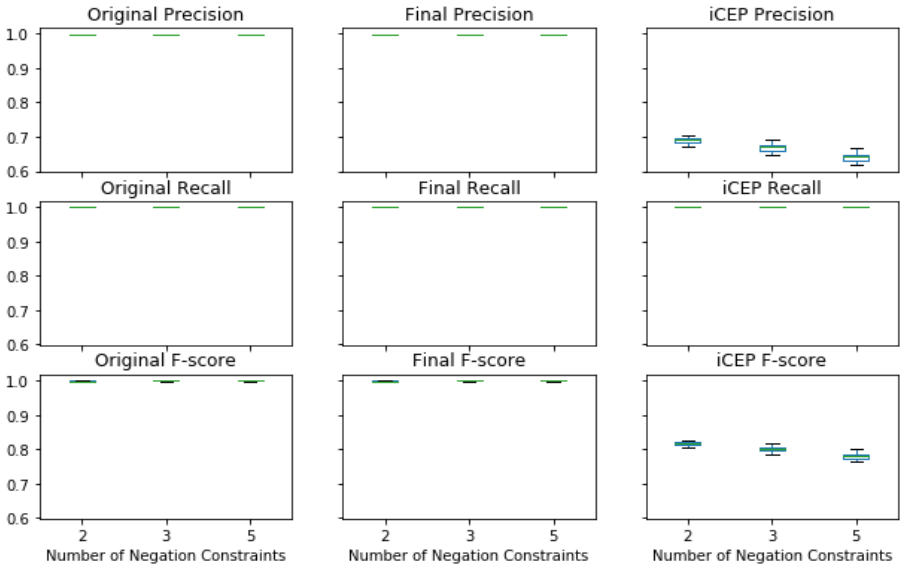

Figure A.5: Detailed plots, number of negation constraints.

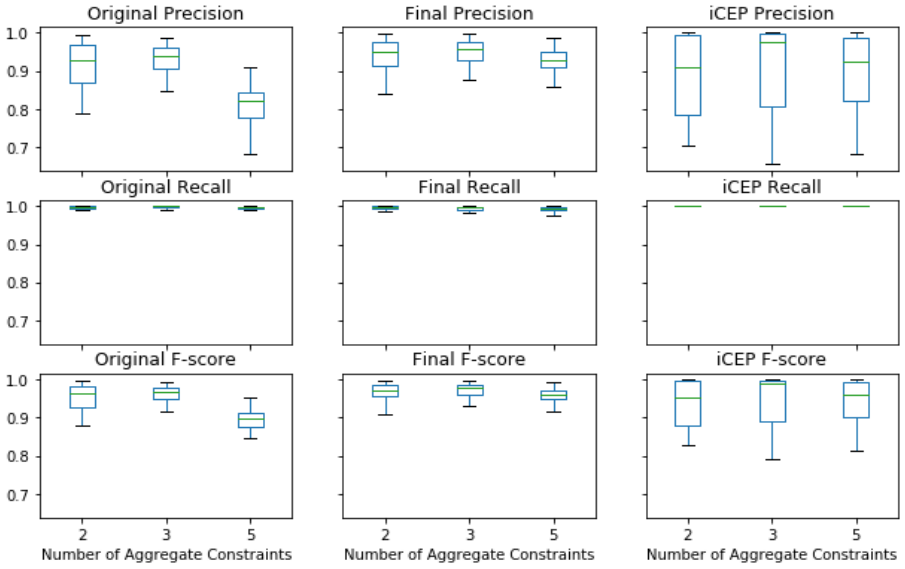

Figure A.6: Detailed plots, number of aggregate constraints with $W = 10$.

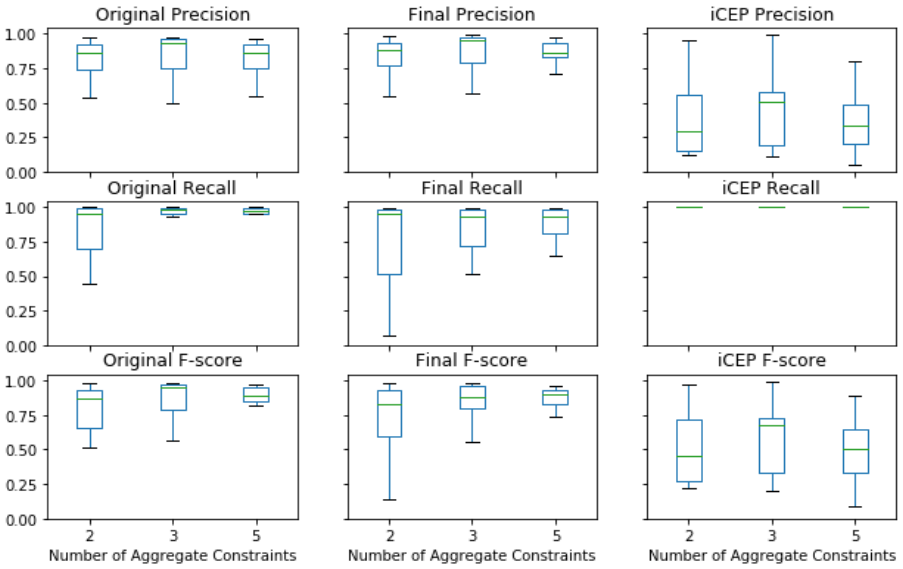

Figure A.7: Detailed plots, number of aggregate constraints with $W = 20$.

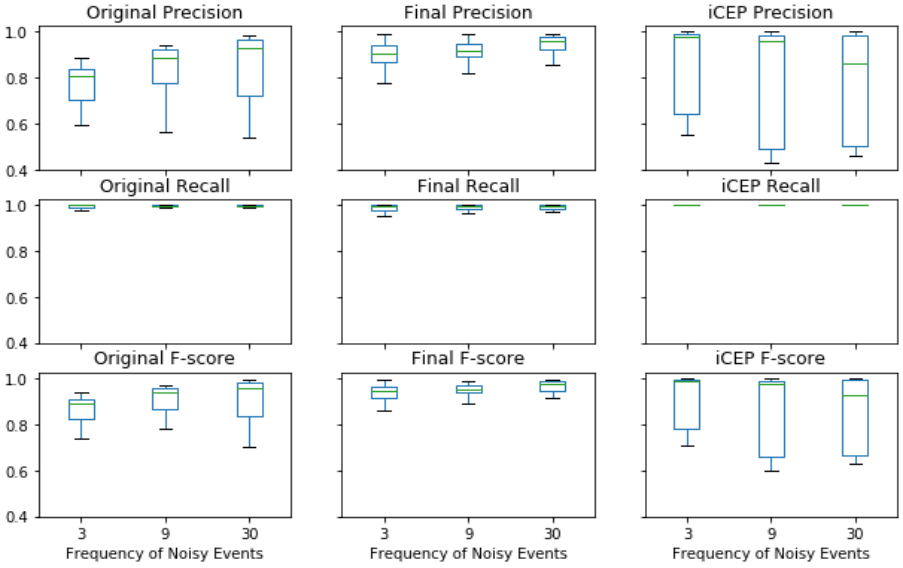

Figure A.8: Detailed plots, frequency of noisy events.

Appendix B

Alternative Testing Method

It follows the set of tests conducted with the alternative method described; it can be observed that they are very similar to the ones obtained with the "standard" method. The detailed plots are showed afterwards.

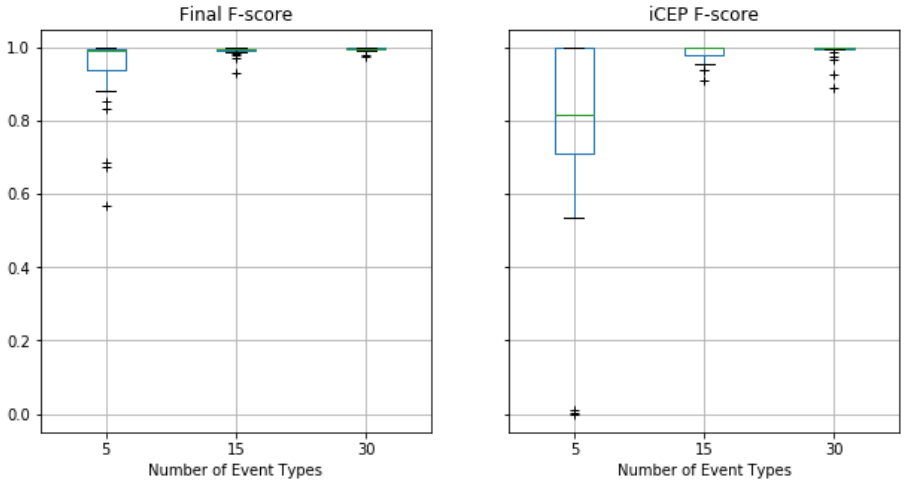

Figure B.1: C5.0 and iCEP F-scores based on the number of event types (alternative method).

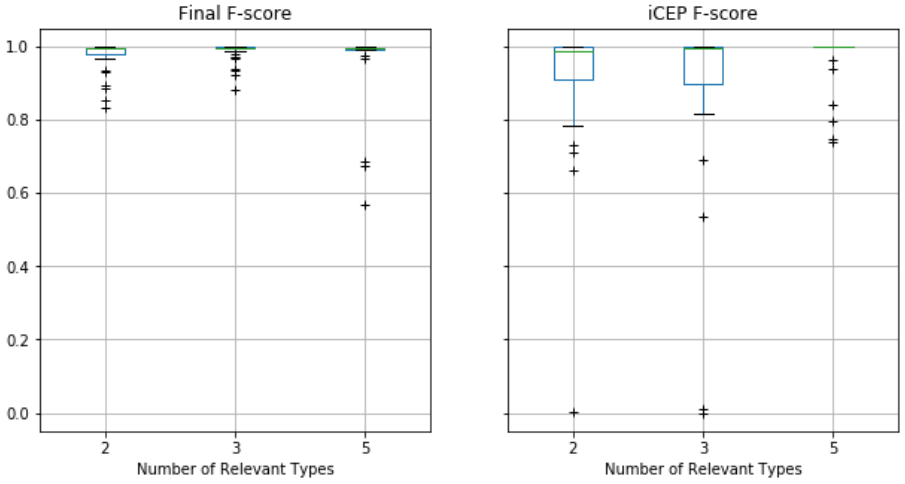

Figure B.2: C5.0 and iCEP F-scores based on the number of relevant types (alternative method).

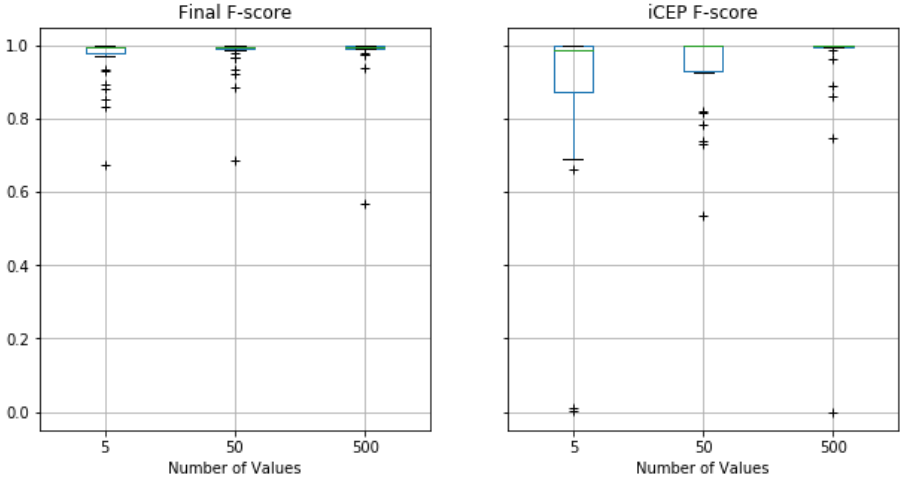

Figure B.3: C5.0 and iCEP F-scores based on the number of attribute values (alternative method).

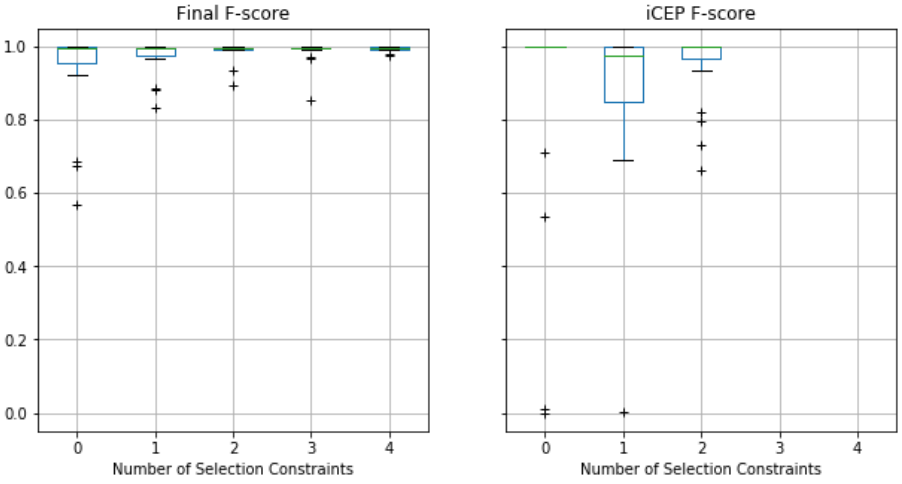

Figure B.4: C5.0 and iCEP F-scores based on the number of selection constraints (alternative method).

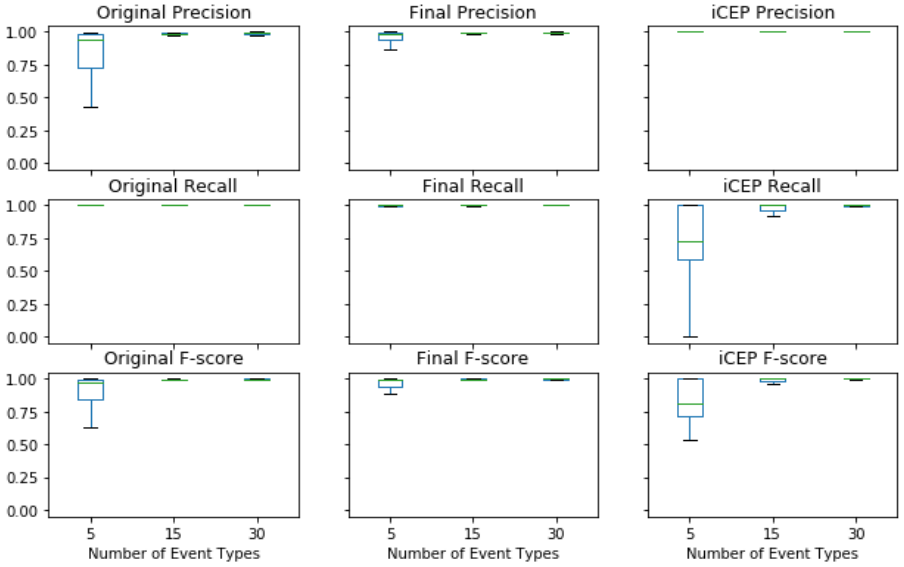

Figure B.5: Detailed plots, number of event types (alternative method).

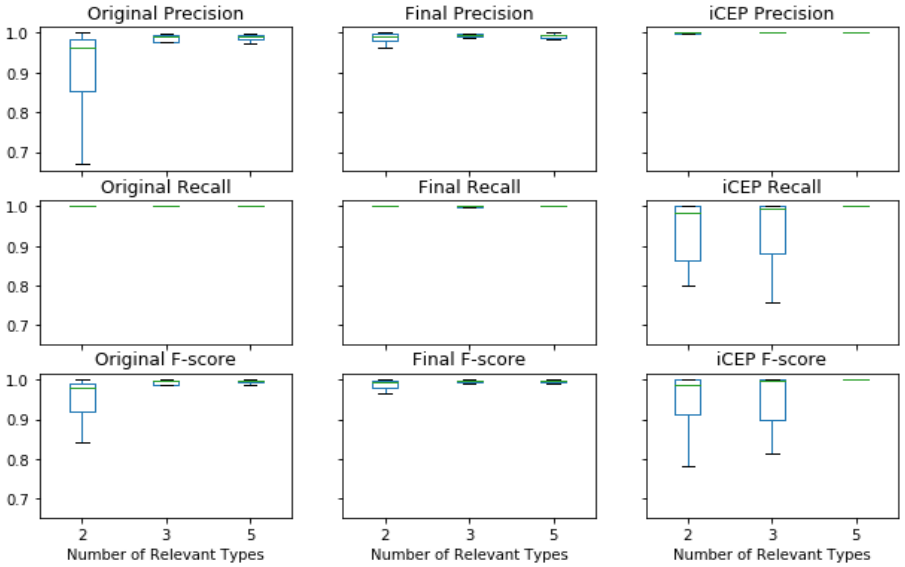

Figure B.6: Detailed plots, number of relevant types (alternative method)

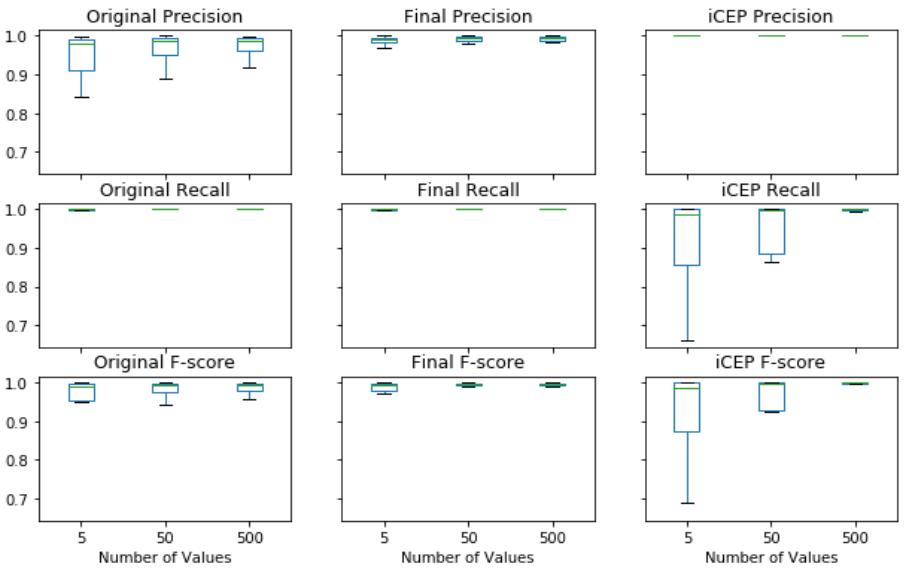

Figure B.7: Detailed plots, number of values (alternative method).

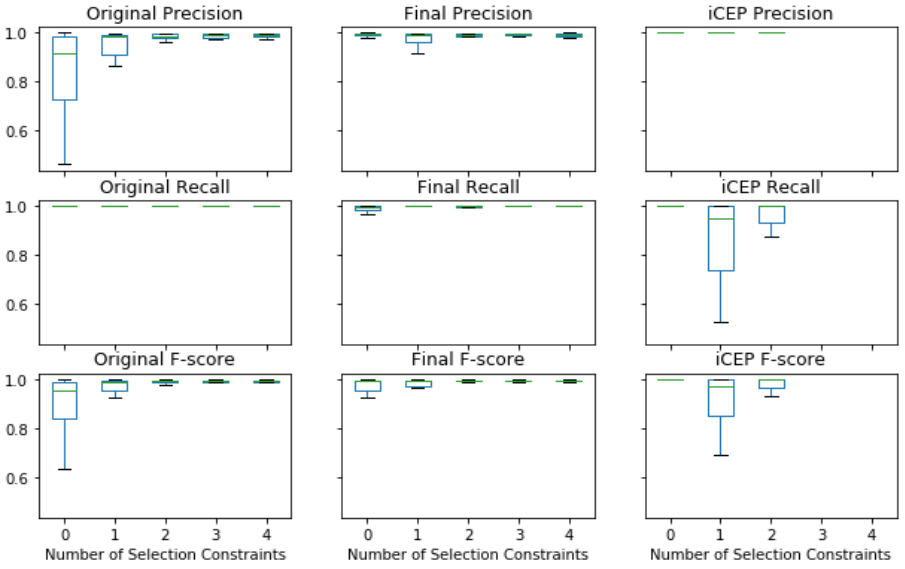

Figure B.8: Detailed plots, number of selection constraints (alternative method).

Appendix C

Miscellaneous Plots

Here, we present interesting plots which correlate different parameters not previously taken into account.

For example, it is interesting to notice how the misclassification costs estimation helps to get high final values of F-score even in settings that create trouble to C5.0, such as when dealing with universes of 5 event types:

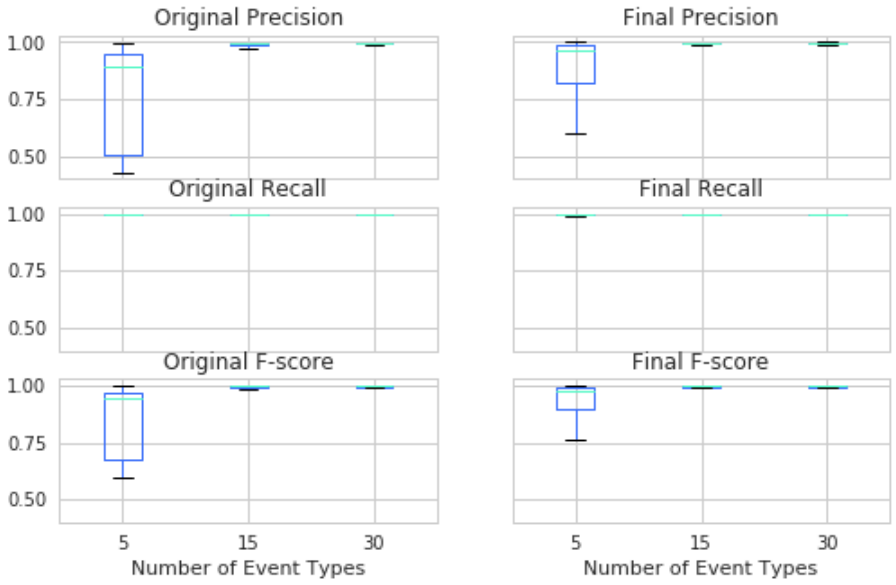

Figure C.1: C5.0 performance before and after employing misclassification costs (based on the number of event types).

Originally, with 5 events the variability is much higher whereas the median drops in the 0.9 area.

It is easy to see why the number of false positives increases with 5 types, since events are uniformly distributed and windows are constant and 10 time units large; as a consequence, the number of traces containing type repetitions is higher as well as the number of repetitions itself, leading to a greater number of false positive sub-traces obtained from the split of original traces, resulting in lower precision and F-score.

Furthermore, we should consider the following points:

- the ratio between the number of relevant types and the number of total event types is generally higher with respect to the other cases; as a consequence, negative traces are more likely to contain relevant events as well and it is harder to classify traces by exploiting the attribute `Event`$_x$`Detected`.
- The number of false positives introduced in the dataset is many orders of magnitude higher when dealing with universes composed

by only 5 types; hence, the resulting decision trees computed are less accurate and more likely to perform incorrect predictions.

This is confirmed by the following plot, showing how the number of event types correlates with the number of errors (false positives) introduced in the training dataset (it can be observed that such number is significantly higher with 5 types):

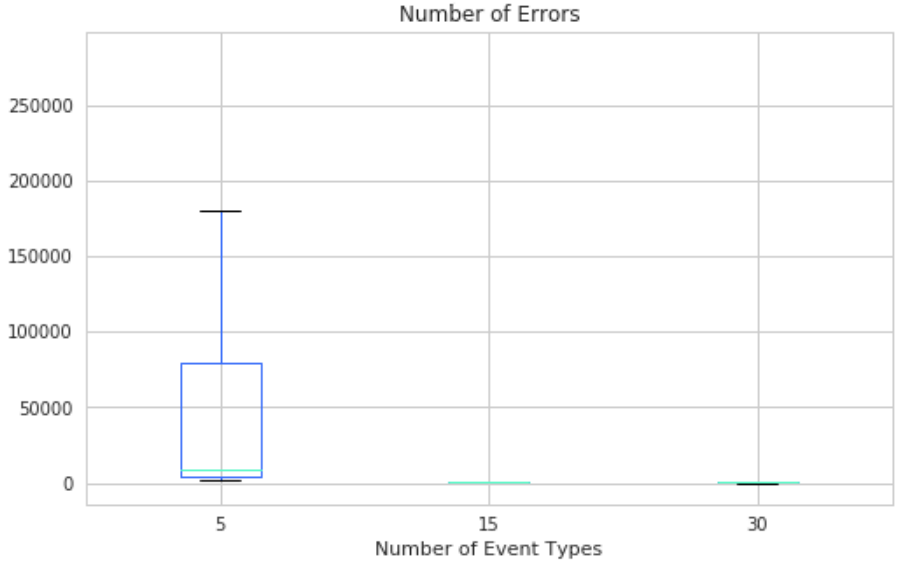

Figure C.2: Number of false positives introduced in the training dataset as a function of the number of event types.

Another difficult setting for C5.0 corresponds to the particular situation in which it is not present any selection constraint. In this case, both precision and recall drop in the 0.9 area, meaning that the overall number of incorrect predictions significantly increases.

As in the case of event types, the reduced performance is due to the fact that C5.0 cannot fully exploit all the attributes of the encoding scheme; here, since there are no selection constraints, the attributes of the kind $\texttt{Event}_x\texttt{Attribute}_k$ are no longer helpful and the algorithm can only detect positive traces based on the $\texttt{Event}_x\texttt{Detected}$ and $\texttt{Event}_x\texttt{PrecedesEvent}_y$ attributes.

Once again, the gap between the original and final performance measurements are due to the misclassification costs estimated.

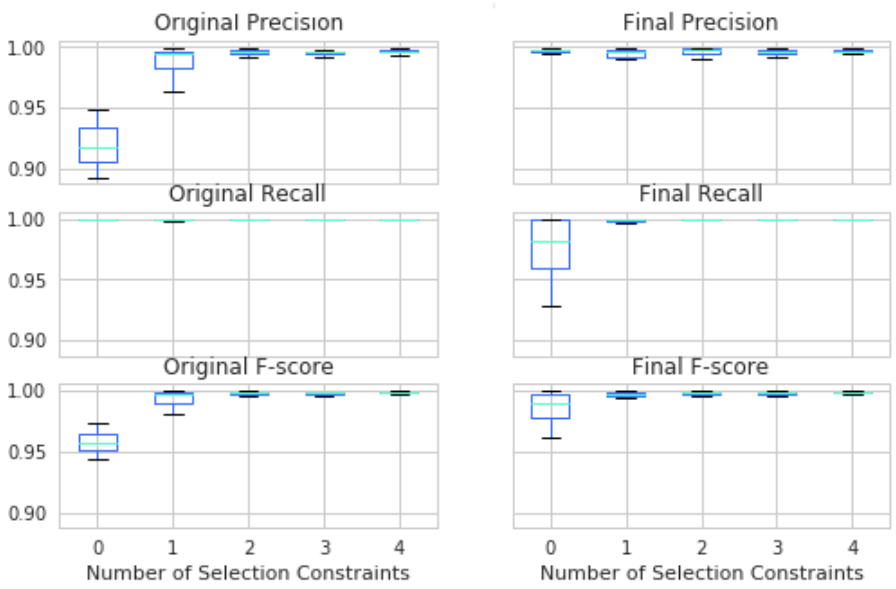

Figure C.3: C5.0 performance before and after employing misclassification costs (based on the number of selection constraints).

Lastly, it is interesting to see how the average misclassification costs estimated increase for parameter values in correspondence of which C5.0 is mostly impaired, with regard to both event types and selection constraints. Nonetheless, this confirms the efficacy of our method for the estimation of misclassification costs.

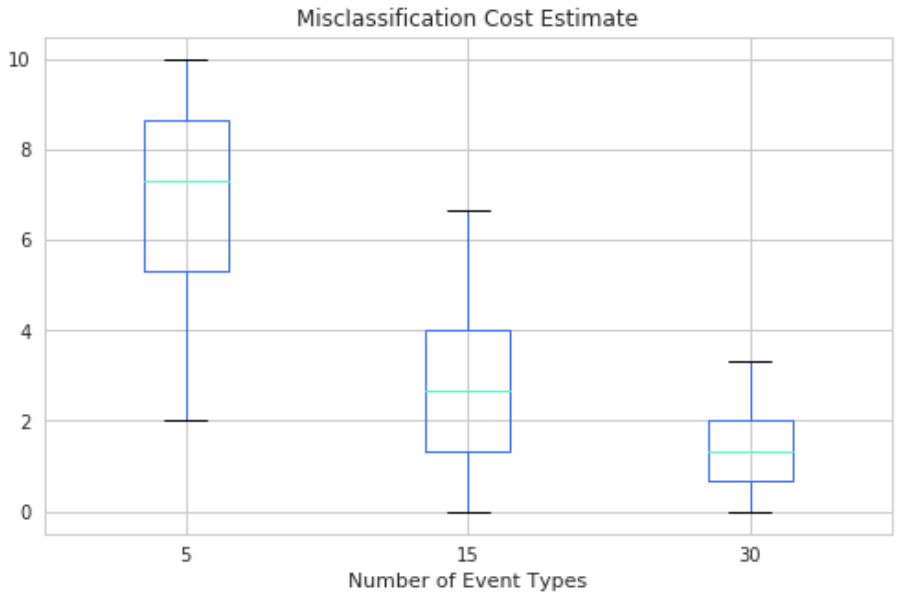

Figure C.4: Misclassification costs estimated based on the number of event types.

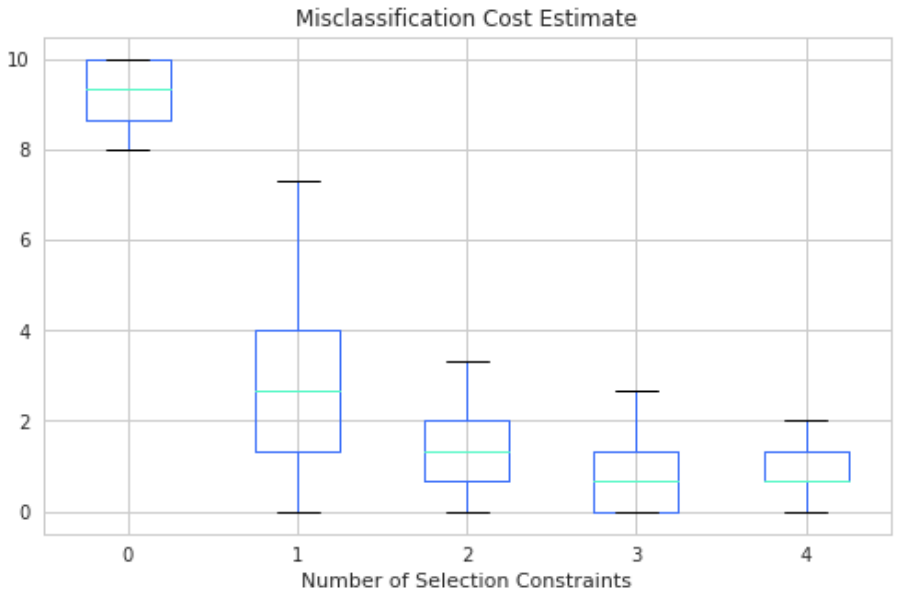

Figure C.5: Misclassification costs estimated based on the number of selection constraints.

Bibliography

[1] C5.0: An Informal Tutorial. https://www.rulequest.com/see5-unix.html.

[2] C5.0 download page. http://rulequest.com/download.html.

[3] EsperTech website. http://www.espertech.com/esper.

[4] Intel AI Academy For Students, Week 1 Slides. https://software.intel.com/en-us/ai-academy/students/kits/machine-learning-101.

[5] "Is See5/C5.0 Better Than C4.5?". http://rulequest.com/see5-comparison.html.

[6] Logs and .csv files - download page. https://github.com/SamueleDL/ML_CEP.

[7] Oracle CEP webpage. http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html.

[8] Weka Data Mining Tool webpage. https://www.cs.waikato.ac.nz/ml/weka/.

[9] Agrawal, J., Diao, Y., Gyllstrom, D., and Immerman, N. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 147–160.

[10] Alevizos, E., Skarlatidis, A., Artikis, A., and Paliouras, G. Complex event processing under uncertainty: A short survey. In *EDBT/ICDT Workshops* (2015), vol. 1330 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 97–103.

[11] Allen, J. F. Maintaining knowledge about temporal intervals. *Commun. ACM 26*, 11 (Nov. 1983), 832–843.

[12] Artikis, A., Katzouris, N., Correia, I., Baber, C., Morar, N., Skarbovsky, I., Fournier, F., and Paliouras, G. A prototype for credit card fraud management: Industry paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems* (New York, NY, USA, 2017), DEBS '17, ACM, pp. 249–260.

[13] Cugola, G., and Margara, A. Tesla: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems* (2010), ACM, pp. 50–61.

[14] Cugola, G., and Margara, A. Complex event processing with t-rex. *Journal of Systems and Software 85*, 8 (2012), 1709 – 1728.

[15] Cugola, G., and Margara, A. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv. 44*, 3 (June 2012), 15:1–15:62.

[16] Cugola, G., Margara, A., Matteucci, M., and Tamburrelli, G. Introducing uncertainty in complex event processing: model, implementation, and validation. *Computing 97*, 2 (Feb 2015), 103–144.

[17] Gyllstrom, D., Wu, E., Chae, H.-J., Diao, Y., Stahlberg, P., and Anderson, G. Sase: Complex event processing over streams. *arXiv preprint cs/0612128* (2006).

[18] Kanaujia, A., Choe, T. E., and Deng, H. Complex events recognition under uncertainty in a sensor network.

[19] Katzouris, N., Artikis, A., and Paliouras, G. Online learning of event definitions. *Theory and Practice of Logic Programming 16*, 5-6 (2016), 817–833.

[20] Kowalski, R., and Sergot, M. A logic-based calculus of events. *New Generation Computing 4*, 1 (Mar 1986), 67–95.

[21] Lei, Q. J., et al. Online monitoring of manufacturing process based on autocep. *International Journal of Online Engineering (iJOE) 13*, 06 (2017), 22–34.

[22] Lichman, M. UCI Machine Learning Repository. `https://archive.ics.uci.edu/ml/datasets/lenses`, 2013.

[23] Margara, A., Cugola, G., and Tamburrelli, G. Learning from the past: Automated rule generation for complex event processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems* (New York, NY, USA, 2014), DEBS '14, ACM, pp. 47–58.

[24] Michelioudakis, E., Artikis, A., and Paliouras, G. Online structure learning for traffic management. In *International Conference on Inductive Logic Programming* (2016), Springer, pp. 27–39.

[25] Michelioudakis, E., Skarlatidis, A., Paliouras, G., and Artikis, A. OSL: Online structure learning using background knowledge axiomatization. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2016), Springer, pp. 232–247.

[26] Molinaro, C., Moscato, V., Picariello, A., Pugliese, A., Rullo, A., and Subrahmanian, V. S. Padua: Parallel architecture to detect unexplained activities. *ACM Trans. Internet Technol. 14*, 1 (Aug. 2014), 3:1–3:28.

[27] Muggleton, S., and de Raedt, L. Inductive logic programming: Theory and methods. *The Journal of Logic Programming 19-20* (1994), 629 – 679. Special Issue: Ten Years of Logic Programming.

[28] Patroumpas, K., Alevizos, E., Artikis, A., Vodas, M., Pelekis, N., and Theodoridis, Y. Online event recognition from moving vessel trajectories. *GeoInformatica 21*, 2 (Apr 2017), 389–427.

[29] Quinlan, J. R. Induction of decision trees. *Mach. Learn. 1*, 1 (Mar. 1986), 81–106.

[30] Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, pp. 22–25.

[31] Ré, C., Letchner, J., Balazinksa, M., and Suciu, D. Event queries on correlated probabilistic streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 715–728.

[32] Sadilek, A., and Kautz, H. Location-based reasoning about complex multi-agent behavior. *Journal of Artificial Intelligence Research 43* (2012), 87–133.

[33] Shen, Z., Kawashima, H., and Kitagawa, H. Probabilistic event stream processing with lineage. In *Proc. of Data Engineering Workshop* (2008).

[34] Skarlatidis, A., Paliouras, G., Artikis, A., and Vouros, G. A. Probabilistic event calculus for event recognition. *ACM Trans. Comput. Logic 16*, 2 (Feb. 2015), 11:1–11:37.

[35] Song, Y. C., Kautz, H. A., Li, Y., and Luo, J. A general framework for recognizing complex events in markov logic.

[36] Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*, 4th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016.

[37] Zhang, H., Diao, Y., and Immerman, N. On complexity and optimization of expensive queries in complex event processing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 217–228.