INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

e-txt2db: Giving structure to unstrutured data

Gonçalo Fernandes Simões

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Outubro 2009

Agradecimentos

Aos meus pais (António e Teresa) e irmãos (Diogo e Bárbara) por terem, desde sempre, acreditado em mim. Em toda a minha vida motivaram-me a ser cada vez melhor dando-me as condições óptimas para a minha vida pessoal e académica. Sem eles não era nem metade do que sou hoje. Por causa desta Tese as férias deles ao Egipto ficaram um pouco estragadas mas eu deixo-lhes aqui a promessa: da próxima vez estarei lá convosco para que possamos gozar as férias ao máximo, juntos. E à minha tia Luz sempre preocupada com o meu ar pálido, a falta de bronze da praia, o tempo imenso de volta da tese, perguntando se, ao menos, não podia sair um bocadinho para ir correr.

Ao João Bastos... um fantástico colega de grupo e amigo! Trabalhamos em conjunto desde o 3º ano de Licenciatura sempre com muito sucesso nos nossos trabalhos. Sem ele de certeza que o meu Mestrado teria sido muito mais difícil. No último ano para além de colegas de grupo partilhámos a sala 2-N9.27 onde tivemos inúmeras discussões sobre as nossas teses e passámos excelentes momentos a ver vídeos do youtube, a ouvir música e a discutir futebol (esta parte não deve ter sido tão agradável para ele, uma vez que metade da conversa era eu a dizer que o FC Porto era o maior).

Ao João Silva (Norton), outro dos meus colegas da sala 2-N9.27, pelas fantásticas discussões sobre as nossas teses. Com ele, eu sabia que podia discutir os problemas da minha tese pois estava claramente à altura do desafio. Para além disso, era o responsável por mais de metade dos vídeos ridículos do youtube ou das músicas estranhas que eram tocadas e nos levaram a grandes momentos de boa disposição.

Ao Ruben Cardoso, que consegue sempre provar que, por melhor que eu seja na teoria, na prática ainda tenho muito para aprender. Ao trabalhar com ele era impossível avançar num projecto sem conseguir argumentar que a solução que estávamos a desenvolver era boa. O treino que ele me deu permitiu-me encarar o desafio da tese de Mestrado muito mais facilmente. O nosso grande projecto é para continuar: dentro de uns tempos eu, ele e o João Bastos vamos estar ricos! Ahahahahahah!

À professora Helena Galhardas pelo excelente trabalho de orientação! Eu admito... muitas vezes as correcções ao meu documento deixavam-me a praguejar. No entanto, tenho a certeza de que todas estas correcções serviram para tornar a minha tese muito melhor. Muito obrigado pela paciência e pelas horas à espera que eu enviasse um capítulo (Professora, já sabe como é: quando eu digo que envio até Segunda-feira é até à madrugada de Segunda para Terça). O Mestrado está a terminar mas agora virão quatro anos como minha orientadora de Doutoramento. Não tenho dúvidas de que com a ajuda dela o trabalho que irei desenvolver nos próximos anos será ainda melhor.

À professora Luísa Coheur, não só por ter sido minha co-orientadora, mas também porque foi a principal responsável por me trazer para a área de Extracção de Informação. Começou por ser minha professora na cadeira de Língua Natural (a minha cadeira preferida do curso), onde me ensinou muita coisa que foi útil para a minha tese. Foi a principal motivadora da criação do SIGIE (Special Interest Group in Information Extraction), que me levou a dar passos mais largos na área de Extracção de Informação. Como co-orientadora foi fantástica... para além das correcções ao meu documento esteve sempre disponível para partilhar conhecimento que foi e será útil ao meu trabalho. Uns dias antes de entregar a tese a professora falou comigo e disse que apesar de já não ser minha co-orientadora no Doutoramento queria continuar a saber novidades sobre o meu trabalho e estaria disponível para me ajudar no que precisasse. Aqui fica uma promessa e um desejo: vou continuar a dar-lhe novidades e espero que no futuro possa voltar a trabalhar com a professora.

Finalmente, quero agradecer a todos os que ajudaram a realizar os testes da minha tese. Alguns passaram mais do que uma hora a realizar testes sem receberem nada em troca. Um muito obrigado para todos eles.

Lisboa, 25 de Novembro de 2009
Gonçalo Fernandes Simões

Aos meus pais e irmãos...

Structure is more important than content in the transmission of information,

Abbie Hoffman

Resumo

Uma grande quantidade de informação manipulada nas organizações está armazenada na forma de documentos não estruturados ou semi-estruturados (por exemplo, relatórios e mensagens de correio electrónico). Não é fácil, para uma ferramenta da software, utilizar esses dados, o que leva a que muita dessa informação seja ignorada. A área de Extracção de Informação oferece um conjunto de técnicas que permitem extrair segmentos de texto desses documentos de modo a torná-las úteis a um utilizador num dado domínio.

Tipicamente, para especificar programas de Extracção de Informação um utilizador pode usar: *(i)* linguagens de programação com suporte de frameworks de Extracção de Informação; *(ii)* linguagens declarativas de Extracção de Informação ou *(iii)* ferramentas ETL (Extraction, Transformation and Loading). Todas estas soluções têm vantagens e desvantagens.

Esta tese propõe uma framework para Extracção de Informação chamada E-txt2db, que procura combinar as vantagens das soluções existentes para especificação de programas de Extracção de Informação. O E-txt2db oferece operadores declarativos baseados na semântica das seguintes tarefas de Extracção de Informação: segmentação, classificação, associação, normalização e correferência. O trabalho desta tese foca-se no desenvolvimento do operador de classificação do E-txt2db, sendo proposta a sua formalização, uma API Java para criação, execução e avaliação de modelos de classificação e ainda uma sintaxe estilo SQL para a especificação de programas de classificação.

Abstract

A large amount of information handled in organizations is stored as unstructured or semi-structured documents (e.g., reports or e-mail messages). It is not easy, for a software tool, to use these types of documents, which means that much of this information is ignored. The Information Extraction area offers a set of techniques that help extracting text segments from these documents in order to make them useful for a user in a given domain.

Typically, to specify Information Extraction programs, a user can use: *(i)* programming languages with support of Information Extraction frameworks; *(ii)* declarative languages for Information Extraction; or *(iii)* ETL (Extraction, Transformation and Loading) tools. All these solutions have advantages and disadvantages.

This thesis proposes a framework for Information Extraction called E-txt2db, that tries to combine the advantages of the existing solutions for the specification of Information Extraction programs. E-txt2db offers declarative operators based on the semantics of the following Information Extraction tasks: segmentation, classification, association, normalization, coreference. The work of this thesis focuses on the development of the classification operator of E-txt2db, proposing its formalization, a Java API to create, execute and evaluate classification models and an SQL-like syntax to specify classification programs.

Palavras Chave
Keywords

Palavras Chave

Extracção de Informação

Linguagens Declarativas

Aprendizagem Automática

Keywords

Information Extraction

Declarative Languages

Machine Learning

Table of contents

List of figures

List of tables

1 Introduction

Statistics of 2005 indicate that more than 80% of business knowledge was stored in digital format (Kaiser & Miksch, 2005). With the increasing use of information technologies in business, it is expected that this percentage grows even more. Much of the information stored is vital to the business only after properly processed. Unfortunately, no Human Being is able to read, understand and synthesize large amounts of data in a reasonable time (Cowie & Lehnert, 1996).

A solution for this problem is the use of software tools for searching, accessing and analyzing data. However, much data in companies (such as e-mail messages, reports and Web pages) are not organized in a structured way. Thus, it is difficult for a software tool to use and analyze them (Sapsford & Jupp, 1996). Consider the example of a mobile telecommunication operator receiving several suggestions for new services via custormers' e-mail messages. The operator must analyze these messages in order to understand which new services should be offered according to the customer profile. To this end, the company can use data analysis algorithms. However, to ensure that the algorithms can be applied, the customer data must be presented in a format that is consumed by the data analysis algorithm. An example of such format is the separation of the e-mail contents into the fields age, residence and occupation.

The question that remains open is how to transform unstructured data into structured information in a way that it can be used by data analysis tools. *Information Extraction* is a scientific area that encloses a set of techniques to extract relevant information in a given domain from unstructured or semi-structured text, and represent it in a coherent format that is understood by other software applications.

1.1 Motivating Example

Seminars4U is a newly founded portal in which scientific researchers can search for information about seminars that may be interesting for them. The *Seminars4U* portal offers search criteria like the theme of the seminar, speaker, location or date.

In order to support searches using different criteria, *Seminars4U* decided to develop a relational database that stores the refined information about seminar announcements in a structured format. The idea is that for each user search, the *Seminars4U* portal accesses the database to determine which seminar announcements match the search criteria.

After designing the database schema and developing the portal, the development team of *Seminars4U* needs to populate the database with seminar announcements. To do so, they decided to use information available in e-mails and Web pages. The problem is that the information of these data sources is in an unstructured and semi-structured format, respectively. Therefore, it is not trivial to map this kind of information into the relational database.

The Information Technologies team of *Seminars4U* proposed a technological solution to this problem. The corresponding architecture is shown in Figure 1.1. The data sources used by the company are e-mails and the Web sites on the left of the figure. The portal answers each user search query, by looking into the seminar announcement database. As illustrated by the black box in the figure, there must exist an automatic mechanism able to extract the relevant information from the sources and insert it into the seminar announcement database.

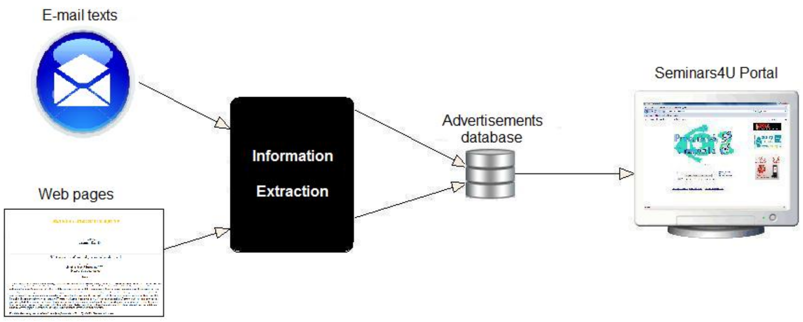


Figure 1.1: Architecture for the retrieval of announcements for the company *Seminars4U*

Let us consider a simple case in which the announcements database contains a single relation: seminar(id,title,speaker,location,date,startTime,endTime). Consider as well the following text excerpt from an e-mail message:

CENTER FOR INNOVATION IN LEARNING (CIL)
EDUCATION SEMINAR SERIES
"Using a Cognitive Architecture to Design Instructions"

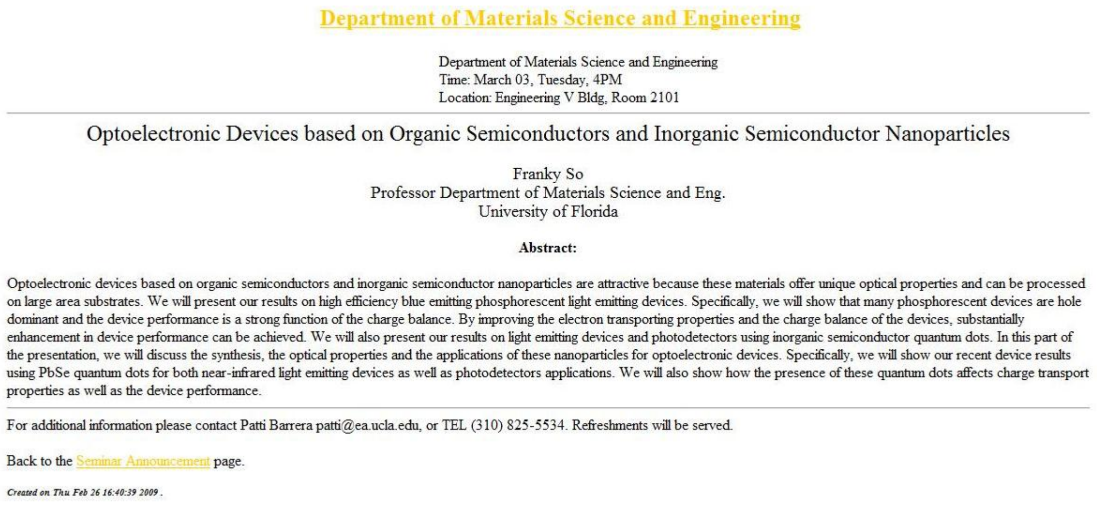
Joe Mertz
Center for Innovation in Learning, CMU

Friday, February 17
12:00pm-1:00pm
Student Center Room 207 (CMU)
ABSTRACT: In my talk, I will describe how a cognitive model was used as a simulated student to help design instructions for training circuit board assemblers. The model was built in the Soar cognitive architecture, and was initially endowed with only minimal prerequisite knowledge of the task, and an ability to learn instructions. Lessons for teaching expert assembly skills were developed by iteratively drafting and testing instructions on the simulated student.

If you want to attend this presentation, please contact Pamela Yocca at 268-7675 before Wednesday, February 15 at 15:00pm

The relevant information to extract from this text should be stored in the following tuple from the seminar relation: seminar(1,"Using a Cognitive Architecture to Design Instructions", "Joe Mertz", "Student Center Room 207 (CMU)", "Friday, February 17", "12:00pm", "1:00pm"). The transformation that is able to extract the values and fill the corresponding attributes can be easily performed by a Human Being. In fact, people can deal with the semantic ambiguity of the text. However, for a machine this is not easy. Consider the extraction of a date. Even if there is a simple way to extract the starting and the ending times from the text, the machine still has the problem of determining which of the times is the starting time and the ending time since there are several times appearing in the text. Difficulties like this one are common when extracting information from Natural Language texts.

As another example, consider the Web page with an announcement for a seminar shown in Figure 1.2 [1]. This page is one of many available in the Web site of UCLA Materials Science and Engineering Department.



Department of Materials Science and Engineering

Department of Materials Science and Engineering
Time: March 03, Tuesday, 4PM
Location: Engineering V Bldg, Room 2101

Optoelectronic Devices based on Organic Semiconductors and Inorganic Semiconductor Nanoparticles

Franky So
Professor Department of Materials Science and Eng.
University of Florida

Abstract:

Optoelectronic devices based on organic semiconductors and inorganic semiconductor nanoparticles are attractive because these materials offer unique optical properties and can be processed on large area substrates. We will present our results on high efficiency blue emitting phosphorescent light emitting devices. Specifically, we will show that many phosphorescent devices are hole dominant and the device performance is a strong function of the charge balance. By improving the electron transporting properties and the charge balance of the devices, substantially enhancement in device performance can be achieved. We will also present our results on light emitting devices and photodetectors using inorganic semiconductor quantum dots. In this part of the presentation, we will discuss the synthesis, the optical properties and the applications of these nanoparticles for optoelectronic devices. Specifically, we will show our recent device results using PbSe quantum dots for both near-infrared light emitting devices as well as photodetectors applications. We will also show how the presence of these quantum dots affects charge transport properties as well as the device performance.

For additional information please contact Patti Barrera patti@ea.ucla.edu, or TEL (310) 825-5534. Refreshments will be served.

Back to the Seminar Announcement page.

Created on Thu Feb 26 16:40:39 2009 .

Figure 1.2: Example of a Web page from which *Seminars4U* wants to extract information

Part of the information of this Web page can be represented in the tuple: seminar(2,“Optoelectronic Devices based on Organic Semiconductors and Inorganic Semiconductor Nanoparticles”, “Franky So”, “Engineering V Bldg, Room 2101”, “March 03, Tuesday”, “4PM”, NULL). The original information is semi-structured since it is an HTML page. However, there are still some difficulties for automatically extracting useful information. For instance, note that the text has no information about the ending time of the seminar. A program that automatically extracts data needs to deal with this problem. It becomes even more complicated if we analyze other seminar announcement pages from the same source. Even though the structure is similar for the pages, the available information may be different for each announcement. For example, there are some pages in which the date is not available, and the starting and ending times are shown with a different notation.

The Information Technologies team of *Seminars4U* proposes the use of Information Extraction techniques to extract the information required to populate the announcements database from unstructured

[1]http://mse.ea.ucla.edu/cal/seminars/20090303-So.htm

and semi-structured texts.

1.2 Problem

An Information Extraction solution, like the one required in the motivating example of Section 1.1, can be developed using the following alternatives: *(i)* a programming language; *(ii)* a declarative language for Information Extraction; or *(iii)* an ETL (*Extract Transform and Load*) tool.

The first approach is to use a programming language, like C or Java to write an Information Extraction program. With these languages, a user can write a program from scratch for a given input data source because they have high expressivity. However, implementing Information Extraction techniques from scratch is an hard work that takes too much time. For this reason, users who use these languages to develop Information Extraction programs are typically assisted by Information Extraction frameworks (e.g., Gate (Cunningham et al., 2002) or Mallet (A. K. McCallum, 2002)). In these frameworks, some Information Extraction functionalities are pre-defined and implemented. The code produced is more reusable, the development time and the number of code lines developed are significantly shorter when compared with the solution of writing a program from scratch. Many of these frameworks offer a wide selection of state-of-the-art Information Extraction techniques. The user may select which one to use in each situation. The main disadvantage of these frameworks is the fact that each Information Extraction technique is implemented by a fixed algorithm. Thus, it cannot be optimized as the queries in a Relational Database Management System. Another disadvantage is that these frameworks are usually hard to learn due to their complexity and sometimes to the lack of documentation.

The second approach consists of using a declarative language to specify an Information Extraction program (e.g., Alog (Shen et al., 2008) or AQL (Reiss et al., 2008)). Declarative languages allow a separation between the logic specification (what is executed) and the physical execution (how it is executed). With this kind of approach, it is possible to have multiple execution possibilities for a given logical specification of an Information Extraction program. It is then possible to choose the more efficient execution alternative for each situation. Another advantage of declarative languages is the high level of abstraction they offer. They do not require much programming knowledge to create an application and it is easy to learn how to use them. The inconveniences of these languages are the following two. First, their expressivity is limited, since there are some tasks for which the predicates or operators supported are not enough. Second, the techniques offered are usually limited. For example, these languages usually do not support machine learning techniques.

Finally, it is possible to use the functionalities of ETL tools such as Microsoft SQL Server Integration Services (Hamilton, 2007). These tools offer pre-defined operators for extraction, transformation and loading of data from several sources. The operators offered by these tools make it easy to specify and maintain an ETL program. Moreover, they are able to deal with multiple data sources. However, the tools have limited expressive power. In fact, ETL tools were not designed specifically for Information Extraction and many of the state-of-the-art Information Extraction techniques are not available (in particular the machine learning techniques). To extend the available techniques and functionalities of ETL tools, the user is usually forced to write ad-hoc code. Another drawback of this approach is the difficulty to optimize programs. Since the supported operators run fixed algorithms, it is not possible to choose, among different execution possibilities, the most efficient for a given input data set.

Summarizing, the characteristics of ETL tools make them a weak solution against Information Extraction frameworks and declarative languages for Information Extraction. But, as we analyzed before, even these two solutions are not perfect. The question we raise is whether there is a solution that combines the advantages of both the Information Extraction frameworks and the declarative languages for Information Extraction.

1.3 Objectives

The initial objective of this thesis was to develop a solution for specifying and executing Information Extraction programs, that combines the advantages of the solutions described in Section 1.2 and that overcomes their problems.

The solution we propose is a framework for Information Extraction that decomposes this activity into several tasks and offers a declarative operator for each. The advantages of this approach are the following: *(i)* it is possible to easily specify an Information Extraction program; *(ii)* it is possible to choose, for each task, the techniques and algorithms that better fit the objective of a particular application; *(iii)* it is easy to locally refine an Information Extraction program since the module responsible for each task is completely independent from the others; *(iv)* it is possible to customize an Information Extraction activity according to an application's needs, by reordering, selecting and composing operators; *(v)* it is possible to automatically optimize the Information Extraction process in a similar way to the queries in a Relational Database Management System because the operators are declarative; and *(vi)* it is possible to use different types of input data in the process.

The decomposition of the Information Extraction activity into tasks adopted in this thesis is shown in Figure 1.3 and will be developed in Chapter 2. As introduced above, the solution we propose is to develop one operator for each of these tasks.

Due to time constraints, we could not focus our effort in operators for all the Information Extraction tasks. Instead, we decided to perform a proof-of-concept with the proposal and implementation of a single operator, for the Classification task. We made this decision for the following reasons: *(i)* the Classification task produces very important results in the context of Information Extraction. There

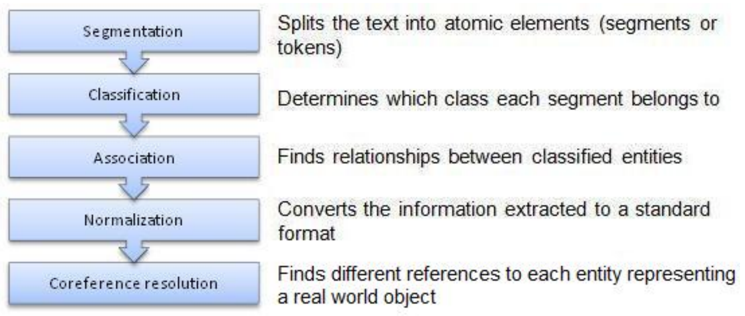

Segmentation
Splits the text into atomic elements (segments or tokens)
Classification
Determines which class each segment belongs to
Association
Finds relationships between classified entities
Normalization
Converts the information extracted to a standard format
Coreference resolution
Finds different references to each entity representing a real world object

Figure 1.3: Decomposition the Information Extraction activity into tasks

are some tasks for which the user is only interested in extracting entities and do not really need to extract relationships between them; *(ii)* there are some Natural Language Processing activities that can be developed as a Classification task (e.g., Part-of-speech tagging, Named Entity Recognition). This fact increases the application opportunities of our operator; *(iii)* there are several implementations of machine learning techniques for Classification available in open source frameworks. These implementations can be used to enrich the number of techniques available for executing our operator.

The fact that only the Classification operator is available imposes a limitation when evaluating E-txt2db. It is not possible to evaluate the customization of the Information Extraction process, since it is only operator available. Moreover, in this thesis we do not focus on the study of optimization techniques for Information Extraction. We only show that is is possible to change the algorithm used by a technique without changing the Information Extraction program specification. For this, we manually changed the algorithm without considering heuristics nor cost models.

1.4 Contributions

This dissertation addresses the problems in the existing solutions to specify and execute Information Extraction programs. The main contributions of this work are:

- Analysis and decomposition of the Information Extraction activity into tasks with a description of the most used techniques for each task. This contribution resulted in a short paper published at Inforum 2009 (Simões et al., 2009).
- State-of-the-art of the existing solutions for specifying and executing Information Extraction programs, with a special focus on Information Extraction frameworks and declarative languages for Information Extraction.
- A framework, called E-txt2db, instantiated for the Classification task. The instantiation of E-txt2db addressed in this thesis encloses:

  - The proposal of an operator for Classification as an extension of the Relational Algebra,
  - A Java implementation for the semantics of this operator,
  - An SQL-like syntax for the Classification operator that supports the declarative specification and the execution of Classification models.

- Validation of the Classification operator, highlighting the characteristics that make this operator a good solution for the specification and execution of Classification programs

1.5 Document organization

The remaining of this thesis is organized into five chapters. In Chapter 2, the basic concepts about information extraction are described. Chapter 3 presents the state-of-the-art of existing solutions to specify Information Extraction programs, with a special focus on Information Extraction frameworks and declarative languages for Information Extraction. Chapter 4 describes the E-txt2db framework, namely the semantics of the Classification operator, its Java implementation and an SQL-like syntax for the creation, execution and evaluation of Classification programs. Chapter 5 describes the experiments performed to validate the instantiation of E-txt2db for the Classification operator. Finally, Chapter 6 summarizes the main topics addressed in this thesis and gives some ideas about the future work.

2 Concepts

In this chapter, we present the basic concepts related to Information Extraction. First, we introduce a definition of Information Extraction. Second, we describe the approaches that are typically followed by Information Extraction techniques. Third, we propose a division of the Information Extraction process into tasks. Finally, we describe the measures used to evaluate the quality of Information Extraction results.

2.1 Definition of Information Extraction

Many definitions of Information Extraction, are highly connected to the used techniques and the corresponding scientific community. For instance, Hamish Cunningham (Cunningham, 2005) defines Information Extraction as "a technology based on analyzing Natural Language in order to extract snippets of information". According to this author, the process receives texts (or speech) and produces data in a fixed and unambiguous format. This definition is strongly connected to the Natural Language Processing community. However, Information Extraction is not only applied to Natural Language texts. In fact, we can perform extraction tasks over semi-structured texts such as HTML or XML files.

Andrew McCallum (A. McCallum, 2005), defines Information Extraction as the "process of filling the fields and records of a database from unstructured or loosely formatted text". This definition considers that Information Extraction must fill records in a database. This vision is a little restrictive because we can perform an Information Extraction without using a database as the output.

In this thesis, we decided that Information Extraction should be defined without the support from both the application area and the techniques used. We wanted a generic definition which was able to describe the objectives of Information Extraction. A definition that fulfills this requirements is given in (Cowie & Lehnert, 1996): "Information Extraction starts with a collection of texts, then transforms them into information that is more readily digested and analyzed. It isolates relevant text fragments, extracts relevant information from the fragments, and then pieces together the targeted information in a coherent framework".

2.2 *Approaches to Information Extraction*

There are two approaches widely used by Information Extraction techniques: *(i)* based on rules or *(ii)* based on machine learning.

2.2.1 Based on rules

This approach is based on the definition of *rules* to extract the relevant information from text. The programmer should be familiar with the domain and the activity the system will perform. In texts with a certain level of structure, it is common to specify the extraction rules using regular expressions. Although this technique is simple to use and interpret, it can only extract very simple and regular patterns.

Another common approach based on rules is the use of Natural Language Processing techniques. These techniques are usually applied for extracting information from Natural Language texts, since they are able to deal with the irregularities of Natural Language (Applet & Israel, 1999). In this case, the idea is to use syntactic (e.g., structure of a sentence) and/or semantic (e.g., logical representation of a sentence) information to perform the extraction.

According to (Applet & Israel, 1999), one of the advantages of rule-based systems is that they are conceptually simple to conceive. Another advantage is that Information Extraction systems based on rules obtain better results when are applied the metrics described in Section 2.4 because they can use specific properties of each text source.

However, the development process of a rule-based Information Extraction system may need a huge number of iterations. If the rules are too specific for the domain, changes in the specification due to domain modifications can be hard to introduce. Sometimes, it becomes easier to develop a new system from scratch. Another problem of this approach is that it needs resources that may not be available, like dictionaries and grammars.

2.2.2 Based on Machine Learning

To try to overcome the difficulties that domain changes bring to a rule-based Information Extraction system, some approaches use *machine learning* algorithms for Information Extraction. According to (Soderland et al., 1999), machine learning allows an efficient adaptation of an Information Extraction system to new extraction domains.

The idea behind this approach is to use an annotated corpus to train the system using a machine learning algorithm (we will examine some of these algorithms with detail in Section 2.3). The corpus is typically produced by an expert with sufficient knowledge about the extraction domain and the required results of the Information Extraction activity.

According to (Silva et al., 2005) we can divide the Information Extraction systems based on machine learning in the following groups:

- *Based on finite automata*: aims at learning the extraction rules in the form of a finite automata. The input of the automata is constituted by text segments. During the Information Extraction procedure some text segments are accepted (leading to a state transition) and all the other are ignored. The accepted segments will fill a data structure that will be the output of the Information Extraction process.
- *Based on pattern matching*: the system learns extraction rules in the form of regular expressions.
- *Based on classifiers*: the input text is divided into fragments. These fragments are candidates for filling a field in the output data structure. Statistical methods are applied to each fragment in order to determine which field of the output structure it may fill.

The first two types of systems have the advantage of being easier to understand, since rules are described using a symbolic language. However, they are not very suitable for texts with a large variation in structure. Systems based on classifiers use several characteristics of the fragments, thus achieving good results for recognizing entities from the text. Still, they have the limitation of doing a classification based on the characteristics of each individual term and they lose some information about the relationship between fragments. Due to these limitations, other techniques began to emerge. Nowadays, the techniques based on Markov Models are being widely used (see Section 2.3).

With the evolution of machine learning techniques, there has been some works developing solutions to reduce the need to use big annotated corpus for training. A particular solution that is very popular nowadays is to use bootstrapping techniques to generate models with minimal intervention from the user. The idea is to start with a small annotated corpus to generate an Information Extraction program. Then, the program iteratively trains itself by extracting information from non-annotated corpus and using the extraction results that offer a higher value of confidence as training corpus to update the previously created program. Examples of works that use approaches similar to this one are described in (Canisius & Sporleder, 2007) and (Carlson & Schafer, 2008)

2.3 Information Extraction tasks

There is no unanimity in what concerns the decomposition of the Information Extraction activity into tasks. In this thesis, we use the decomposition of the Information Extraction activity that is usually adopted by the machine learning community (A. McCallum, 2005). The considered tasks are: Segmentation, Classification, Association, Normalization and Coreference resolution.

2.3.1 Segmentation

The *Segmentation* task divides the text into atomic elements, called segments or tokens. Some authors refer to Segmentation as a trivial process for Western languages. However, this view is not entirely correct. Next, we present some problems that can be found in Segmentation for Western languages (Santos, 2002):

- *Whitespaces*: space-separated words may not match the different segments.For example, the string "New York" should be considered a segment.
- *Hyphens*: compound words in which there is a separation with an hyphen (e.g., "daughter-in-law") should be part of the same segment. However, there are situations in which we could separate a segment with an hyphen in two words (e.g., non-lawyer).
- *Apostrophes*: in languages like English or French, it is very common to find situations in which apostrophes are used in contractions. These situations can be difficult to handle. For instance, the expression "s" in "John's" can be the verb "to be" or the possessive form.
- *Full stop*: the full stop usually marks the end of a sentence. However, it can also be used to separate letters in an abbreviation (e.g., "United Nations" can occur as "U.N.").

These problems show that the Segmentation task is not as trivial as it looks, even for Western languages. Usually, they are solved using rules that show how these cases should be handled.

Major problems related to this task can be found in oriental languages. For example, the Chinese does not have whitespaces between words (Haizhou et al., 1998). For this reason, solving the problems described above is not enough in this language. In these cases, it is typically necessary to use external resources. Dictionaries and grammars can be used in order to accomplish the task of Segmentation using lexical or syntactic analysis. Another approach for Segmentation in Chinese uses techniques based on statistics. An example is the system described in (Haizhou et al., 1998), which uses *N-grams* and the *Viterbi algorithm* (Forney, 1973) for Segmentation.

The techniques based on *N-grams* resort to word counts to define a conditional probabilistic model. In the model, the probability of a given word depends on the last N-1 words.

Consider a vocabulary with only two words, "A" and "B", and a training corpus "A B A A B B A". We want to train a bigram model so we need two types of counts: unigram count and bigram count.

Table 2.1.1 presents the frequency of each word in the training corpus. Table 2.1.2 presents, for each word in the lines, the frequency with which it appears in the text after the word in the columns. Note that, in these tables, we consider two symbols that do not appear in the text: $< s >$ which represents the beginning of the corpus and $< /s >$ which represents the ending of the corpus.

Table 2.1: Unigram (1) and Bigram (2) counts for the training corpus $A\ B\ A\ A\ B\ B\ A$

Table 2.1.1

$< s >$	A	B	$< /s >$
1	4	3	1

Table 2.1.2

	$< s >$	A	B
A	1	1	2
B	0	2	1
$< /s >$	0	1	0

As an example of using this model we calculate the probability that the sentence "ABA" occurs. With a bigram model we can compute the occurrence probability of this sentence by Equation 2.1.

$$P(\text{“}ABA'') = P(A| < s >)P(B|A)P(A|B)P(< /s > |A) \tag{2.1}$$

where P(i|j) is computed by the Equation 2.2

$$P(i|j) = \frac{Table2.1.2(i,j)}{Table2.1.1(j)} \tag{2.2}$$

Thus, we have P(A| $< s >) = \frac{1}{1}$, P(B$|A) = \frac{2}{4}$, P(A$|B) = \frac{2}{3}$, P($< /s > |A) = \frac{1}{1}$, which means that $P(\text{“}ABA'') = \frac{1}{1}x\frac{2}{4}x\frac{2}{3}x\frac{1}{1} = \frac{1}{3}$

The advantage of the techniques based in *N-grams* is the fact that they are capable of using information about the neighborhood of each word without a rich lexical knowledge. However, these techniques are too dependent of the training corpus.

The *Viterbi Algorithm* is a dynamic programming algorithm used to estimate the most likely sequence of states in a state machine, given a set of observable events. These events constitutes the input sequence of the state machine. The algorithm assumes some facts that may not necessarily be true. First, it assumes that the observed events are in sequence with the states of the states machine. Then it assumes that the probability of the sequence depends only on the event observed in a finite number of previous iterations.

The most likely sequence of states for the observable events, in each iteration, is through the computation of the Viterbi score, which may be computed in different ways depending on the technique used. A detailed example of the *Viterbi Algorithm* will be introduced in Section 2.3.2 as a technique for Classification.

Another statistical approach to Chinese word Segmentation is described in (Teahan et al., 2000). In this work, the lack of space between segments is treated as spelling mistakes. The idea is to use a training corpus to create a language model that suggests the introduction of spaces between words to correct errors.

2.3.2 Classification

The *Classification* task determines the type of each segment obtained in the Segmentation task. In other words, it determines the field of the output data structure where the input segment fits. The result of this task is the Classification of a set of segments as entities, which are elements of a given class potentially relevant for the extraction domain. In this task, it is possible to get good results. Some systems have results over 90% which is almost at Human level. For example, for a named entity recognition task there are some systems that are able to get results around 95% (Cunningham, 2005).

The rule-based techniques used in the Classification task are usually based on linguistic resources, such as dictionaries and grammars (Farmakiotou et al., 2000). The input segments are compared with the elements of the dictionary until there is a match. Additionally, some techniques of morphological analysis can be used as support. The grammar supports the recognition of terms that are not in the dictionary and solves possible ambiguities when an entity has different categories in the dictionary (e.g., when there are homonymy relationships).

One of the most popular approaches to undertake Classification is machine learning. Machine learning techniques used in this task are usually supervised, which means that an annotated corpus is needed. Five of the most common supervised learning techniques are the Hidden Markov Models (HMM), Maximum Entropy Markov Models (MEMM) (A. McCallum et al., 2000), Conditional Random Fields (CRF) (Lafferty et al., 2001), Support Vector Machines (Isozaki & Kazawa, 2002) and Decision Trees (Sekine et al., 1998).

HMM have been widely used in text Classification tasks such as Part-of-Speech tagging that consists of classifying words according to the morphological class. These models are based on: *(i)* a set of hidden states that usually have a physical meaning (e.g., in the classification of tokens we can have each state associated with a type of entity); *(ii)* a set of observable symbols that correspond to observable events (e.g. tokens that occur at each time), *(iii)* a function that returns the probability distribution of the state transition, *(iv)* a function that returns the distribution of probabilities of observing a given symbol in a given state and *(v)* a vector corresponding to the distribution of probabilities in the initial state. A probabilistic model is built during the training phase. The Classification is then obtained through the use of the generated model for finding the most likely classification for the test dada. The most likely classification can be computed using the Viterbi Algorithm.

Consider the classification of segments in seminar announcement documents. We assume the existence a trained HMM model for this task. This model considers for each word the probability of being a speaker, a location or a non-classified word (NEG):

P(*Lewis*|speaker) = 0.6	P(*teaches*|NEG) = 1.0
P(*Lewis*|NEG) = 0.4	P(*cryptography*|location) = 0.2
P(*Washington*|speaker) = 0.4	P(*cryptography*|NEG) = 0.8
P(*Washington*|location) = 0.6	

We do not present the probabilities for all the classifications of each word. If one probability is not presented we will consider to be zero. The model also considers the bigram probability of a given classification that depends on the word classification of the word before. This probability is presented as a Markov chain in Figure 2.1:

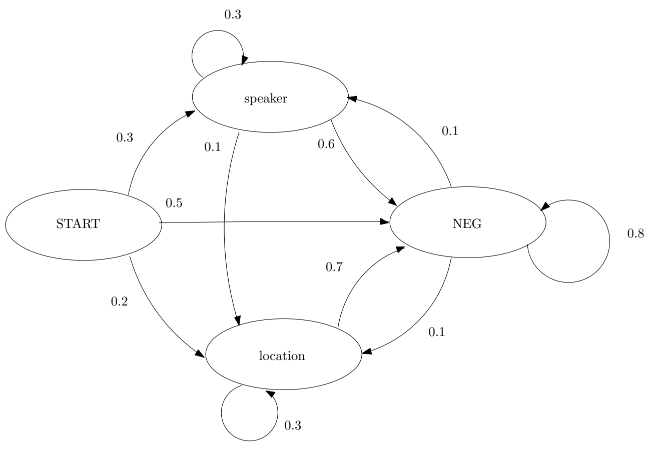


Figure 2.1: A Markov Model capturing the bigram probabilities of this model

We can use this model and the *Viterbi algorithm* to predict the classification for the sentence "Lewis Washington teaches cryptography".

The *Viterbi algorithm* is divided into three steps. The first one is the *Initialization Step* in which we compute a score for the first word (w_1) in the sentence. This score is given by Equation 2.3:

$$v_1(j) = P(w_1|L_j) * P(L_j|START) \tag{2.3}$$

where L_i is each of the possible classifications: speaker, location and NEG. For the example, the first word is "Lewis". The scores for this word are presented in Figure 2.2.

In the second step of the algorithm (*Iteration Step*), we scan all the other words and compute the best score for each classification using the Equation 2.4:

$$v_t(j) = max_{i=1}^{N} v_{t-1}(i) P(Tag_j|Tag_i) P(Word_t|Tag_j) \tag{2.4}$$

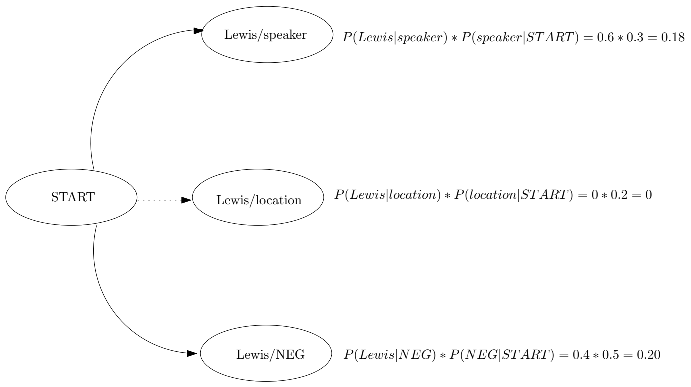

START
Lewis/speaker
$P(Lewis|speaker) * P(speaker|START) = 0.6 * 0.3 = 0.18$
Lewis/location
$P(Lewis|location) * P(location|START) = 0 * 0.2 = 0$
Lewis/NEG
$P(Lewis|NEG) * P(NEG|START) = 0.4 * 0.5 = 0.20$

Figure 2.2: Score for the word "flies" in the *Initialization Step*

Figure 2.3 shows the Iteration Step computation for the word "Washington". The solid arrows show the path which corresponds to the maximum of $v_{t-1}(i)P(Tag_j|Tag_i)$ in Equation (2.4).

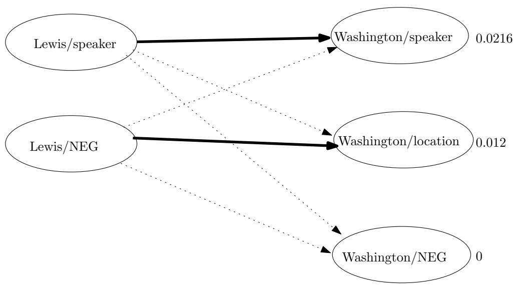


Figure 2.3: Score for the word "Washington" in the beginning of the *Iteration Step*

This step is repeated for all the words in the sentence. The result after the step is performed for the last word is shown in Figure 2.4.

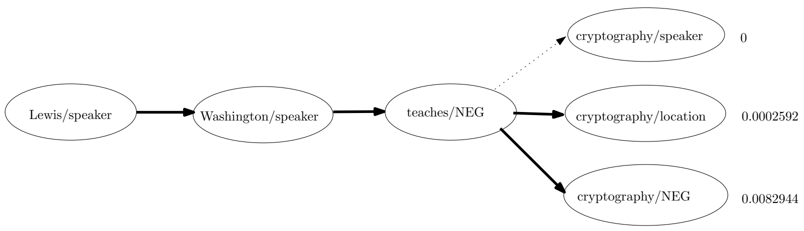


Figure 2.4: Score for the word "cryptography" in the *Iteration Step*

In the final step of the Viterbi algorithm (Backward step) the path with the highest score is scanned from the last one to the first one. In this scan, the algorithm gives a final classification to each word. In the example, we can see that the path with highest score is the one in which "Lewis" and "Washington" are part of the speaker and "teaches" and "cryptography" are not classified.

The HMM models for the Classification task are based only on two kinds of probabilities: $P(Tag|Tag)$ and $P(Word|Tag)$. This means that the introduction of other knowledge in the Classification process (e.g., information about capitalization ou letter size) is not easy. In HMM, if we want to model this knowledge we need to code it only with the two kinds of probabilities presented before. In order to solve these problems, some approaches began to emerge that seek to modify the HMM. Examples of these approaches are Maximum Entropy Markov Models (MEMM) (A. McCallum et al., 2000) and Conditional Random Fields (CRF) (Lafferty et al., 2001).

Both MEMM and CRF are motivated by the principle of maximum entropy, which is a framework that allows to estimate the probability distribution from a set of training data. The principle of maximum entropy states that a probability distribution built from incomplete information, such as a finite training data, is that which has maximum entropy subject to a set of constraints representing the information available. It is proven that a probability that maximizes the entropy must be as uniform as possible. Any other distribution will involve unwarranted assumptions (Wallach, 2004).

While the HMM use a generative model, which produces a probability density model over all variables in a system, the MEMM use a discriminative model, which makes no direct attempt to model the underlying distribution of the variables. Instead of having separate models for $P(Tag|Tag)$ and $P(Word|Tag)$, the MEMM train a single probabilistic model to estimate $P(Tag_i|Word_i, Tag_{i-1})$. This probabilistic model uses a Maximum Entropy classifier that estimates the probability for each local tag given an observed word, a tag for the prior word and a set of features corresponding to the modeling of additional information we want to introduce in the process.

Like in the HMM, the MEMM use the Viterbi algorithm to infere the tags associated to each word. The only difference between the two algorithms is in the way the probabilities are computed. In the HMM, the Viterbi score is given by Equation 2.5:

$$v_t(j) = max_{i=1}^{N} v_{t-1}(i) P(Tag_j|Tag_i) P(Word_t|Tag_j) \tag{2.5}$$

The MEMM uses a modified version of Equation 2.5 that is presented in Equation 2.6:

$$v_t(j) = max_{i=1}^{N} v_{t-1}(i) P(Tag_j|Tag_i, Word_t) \tag{2.6}$$

MEMM are a good solution if we want to introduce additional information in the statistical model but they suffer from a limitation that makes them weaker than HMM when no features are used. Consider the following example of a statistical model:

If we observe the local probabilities for each observation we can see that for state 1, the transition to state 2 is usually the one with the highest probability. We can see that the same happens with state 2. But if we use the Viterbi algorithm to compute the most probable path we find out that it is the path 1→1→1→1.

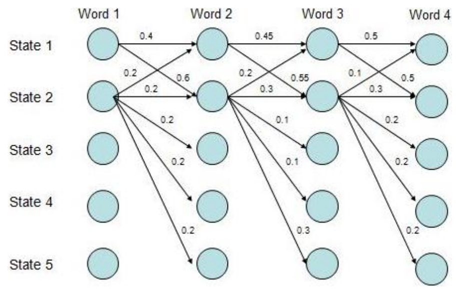


Figure 2.5: Example of a MEMM model with the Label Bias Problem

This happens because there are five transitions out of state 2 and only two transitions out of state 1. Since the probabilities of the transition from a given state in a MEMM are normalized, the model usually prefers states with lower number of transitions. This gives an unfair advantage to these states. This is called the *label bias problem*.

CRF were developed as a solution that offers all the advantages of MEMM but overcomes the label bias problem. The difference between CRF and MEMM is that the MEMM model uses a conditional probabilities exponencial model for each state while CRF uses a single exponencial model for the joint probability of the entire sequence of labels given the observation sequence. With this model it is possible to normalize the probabilities at a global level.

The Viterbi score used in the Viterbi algorithm for CRF is different from the ones used in HMM and MEMM. It is given by:

$$v_t(j) = max_{i=1}^{N} v_{t-1}(i) e^{\sum_k (\lambda_k f_k(Tag_i, Tag_j, Sentence, t))} \tag{2.7}$$

In Equation 2.7 f_k is a feature function. For each feature representing additional knowledge about words of the text we want to use in the process there will be a different feature function. λ_k is a parameter estimated in the training phase that represents the importance of each feature function in the process.

Typically CRF outperforms both HMM and MEMM (Lafferty et al., 2001) but the training of a CRF model is very expensive, which makes it difficult to use if we want to keep updating our model over time (forcing us to train a model in each iteration).

Support Vector Machines assume that it is possible to map the segments of the text in a vector space according to linguistic (e.g., lexical information) or graphical (e.g., position or style in the text) properties of the segments and the words in its neighboring. Text segments are mapped into a vector space. Then, the idea is to separate positive elements (elements that belong to the class) from negative elements (elements that do not belong to the class) of a given class by an hyperplane. In the training phase, the objective is to find the hyperplanes that achieve the better separation between positive and negative examples. In the testing phase, the Classification is performed by looking at which side of the hyperplane the input is. By other words, if it is a positive or a negative element of a given class.

Let us consider the example of a classification of lines in a text as titles. We will use two graphical properties to classify the lines: the size of the letters and the line numbers. For the training phase some positive and negative examples are given. The examples are shown in a bidimentional space in Figure 2.6. The positive examples are the dots and the negative examples are the stars. Figure 2.6 also shows the best choice for the hyperplane.

In the testing phase, each line is mapped in the bidimentional space as done with the training examples and the classifier uses the hyperplane to classify it. Figure 2.6 shows two new lines (a square and a cross) that will be classified with this model. The square is positioned below the hyperplane, so it is considered that it is not a title. The cross is positioned above the hyperplane, so it will be classified as a title.

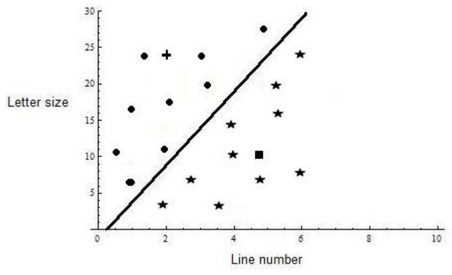


Figure 2.6: Testing phase in a SVM classifier

Decision Trees are also used for text Classification. An example is described in (Sekine et al., 1998) for classifying Japanese texts. During training, a probabilistic decision tree is built based on the words' morphologic classification, type of characters and information of the dictionary in the neighborhood of the word (word right before, the word itself and the word right after). In the testing phase, the properties of the neighborhood of each word are analyzed and compared with the decision tree in order to associate a probability of belonging to a given class. Given that the probabilities of all the segments are computed, the task consists in discovering the most consistent sequence of probabilities. For that, the Viterbi Algorithm can be used.

Let us consider a task in which we want to classify words as locations. We will use the decision tree of Figure 2.7 for this task.

From this decision tree we can get the probabilities of a given word to be or not to be a Location. Consider the sentence "I am going to Washington". To compute the probabilities associated to "Washington" we start from the root of the tree. Since "Washington" starts with capital letter, the algorithm follows the left path and gets to another node. In this node we need to observe the word before "Washington" and see if it is a preposition. Since "to" is a preposition we follow the left path again. We

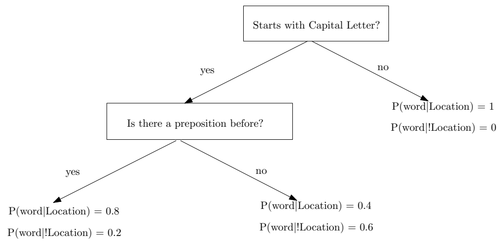


Figure 2.7: Simple Decision Tree to compute the probability of a word being a Location

conclude that "Washington" has 80% chance of being a Location.

The probabilities computed for each word of the text using the decision tree are used in the Viterbi Algorithm. As in the example of the HMM, we will use the probabilities model to find the sequence of classifications with highest score.

2.3.3 Association

The *Association* task seeks to find how the different entities found in the Classification task are related. The systems that perform extraction of relations are less common that the ones that perform the Classification task (A. McCallum, 2005). This happens due to the difficulty in achieving good results in this task.

Many techniques in the Association task are based on rules. The simplest approach uses a set of patterns to extract a limited set of relationships. For instance, we can extract an affiliation relationhip between a person and a company with the rule (2.8), in which, upon detection of the standard <Person> works for <Company>, the values of <Person> and <Company> are inserted in the relation worksFor(Person,Company).

$$< Person > worksfor < Company > \rightarrow worksFor(< Person >, < Company >) \tag{2.8}$$

Such an approach solves only very simple cases, where a big variety of relationships is not expected, since we need to develop a different rule to extract a different relationship.

Another rule-based approach for Association, which is more generic than the last one, is the use of *syntactic analysis*. Often, the relationships that we want to extract are grammatical relationships (Grishman, 1997). For example, a verb may indicate a relationship between two entities. A complete syntactic analysis, where all the text is analyzed on the same syntactic tree, has the inconvenient of being very expensive and cause many errors (Grishman, 1997). In Information Extraction, partial syntactic analysis is used, dividing the text into parts where each part has its syntactic tree. With this kind of analysis some linguistic patterns, as conjunctions or modifiers, are ignored.

The Association task can also use machine learning techniques. One of the first approaches to machine learning in this task (Miller et al., 1998) was based on probabilistic context-free grammars. These grammars differ from regular context-free grammars because they have a probability value associated to each rule. When the syntactic analysis is undertaken, it is possible to find many syntactic trees. By using probabilistic rules, the probability of each tree is computed and the most probable tree is chosen.

Since HMM (refered in Section 2.3.2) are usually more suitable for modeling local problems, its use for the Association task is not the most appropriate (Zelenko et al., 2003). However, other approaches like MEMM and CRF (Section 2.3.2) may be adapted for the Association task. To do so, all we need to do is to add information about the neighborhood of a word in the form of features. Instead of tagging each word with a given class, these models would tag with relationships with other words in the text.

2.3.4 Normalization

The *Normalization* task is required because some information types do not conform to a standard format. Consider, as an example, the format of the hours. In texts from different sources we can find the same time represented in distinct formats such as "3pm", "3h", "15h", "15:00", "1500". This format heterogeneity may pose difficulties in the comparison between entities.

The Normalization task transforms information to a standard format defined by the user. In the example of the hours mentioned above, the author can define that all the hours shall be converted to a standard format, for example, "15h00".

This task is typically achieved, through the use of conversion rules that produce a standard format previously chosen.

2.3.5 Coreference resolution

Coreference arises whenever the same real world entity is refereed in different ways in a text fragment. This problem may arise due to the use of: *(i)* different names describing the same entity (e.g., the entity "Bill Gates" can be found in the text as "William Gates"), *(ii)* classification expressions (e.g., a few years ago, "Bill Gates" was referred as "the world's richest man"), *(iii)* pronouns (e.g., in the sequence of sentences "Bill Gates is the world's richest man. He was a founder of Microsoft", the pronoun "He" refers to "Bill Gates"). The last case is known in the Natural Language community as *anaphora*, which consists of using a pronoun as a reference to a concept in a linguistic neighborhood.

The techniques for handling coreference can be rule-based or machine learning-based. Rule-based approaches usually take into account semantic information about entities. With this information the detection of correferent entities is done through filtering. This means that only entities whose semantics information coincides can be correferent. The filtering can be done manually or using independent resources like the Wordnet [1] (Fellbaum, 1998). Wordnet is a semantic lexicon that is available for multiple languages (including English and Portuguese). In the filtering process, it could be used to discover the semantic information associated to each word from the process. At the end of the filtering phase, it is necessary to determine which entities have the highest probability of being correferent. For that, it is possible to use information about the relative position in the text (e.g., in the case of anaphora, it is expected that the correferent entities are sufficiently close in sentence).

A machine leaning approach for Coreference resolution is described in (Cardie & Wagstaff, 1999). This approach is based on clustering algorithms for grouping similar entities. Initially, the entities of the document are analised from the ending to the beginning of the document and the distance between each one of them is computed. The distance is computed using an incompatibility function and a set of weighting constants through the Equation (2.9).

$$dist(E_i, E_j) = \sum_{f \epsilon F} w_f incompatibility_f(E_i, E_j) \tag{2.9}$$

The incompatibility function uses attributes like name, position, number, semantic class and gender to determine the difference between two entities. If two entities have an high incompatibility for a given attribute, it is probable that they can not be correferent.

If the distance between two entities, E_i e E_j, is less than the cluster radius which is pre-defined, then the entities belong to the same cluster. Consider the following example: “Bill Gates was one of Microsoft’s founders. He is not the president of the company anymore”. With this approach, the distance between “Bill Gates” and “He” is expected to be small. The same is expected for the entities “Microsoft” and “company”. Thus, the result must have at least two cluster: $C_1 = \{BillGates, He\}$ e $C_2 = \{Microsoft, company\}$. If two entities are contained in the same cluster of the final result, they are considered correferent.

Another machine learning approach for Coreference resolution is based on decision trees. The tree is build with the data from a corpus tagged with coreference relationships. Analogously to the clustering approach, the construction of the tree makes use of a set of attributes such as name, position, number. After the training, the text is processed from left to right and each entity is compared with all the previous entities. For each pair, the tree is used to check if its elements are correferent.

Consider as an example, the decision tree of Figure 2.8.

Consider again the example “Bill Gates was one of Microsoft’s founders. He is not the president of the company anymore”. Let us use the decision tree to check if the pair of elements (Bill Gates, He)

[1] http://wordnet.princeton.edu/

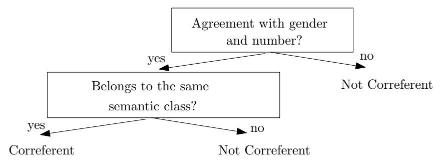

Agreement with gender and number?
yes
no
Not Correferent
Belongs to the same semantic class?
yes
no
Correferent
Not Correferent

Figure 2.8: Simplified example of a decision tree for Coreference resolution

are correferent. As the two elements are singular and male, in the first level of the tree, we choose the option "yes". Since "Bill Gates" and "He" belong to the class "Person" (could have been concluded in the Classification task), we choose again the option "yes" and we get to a leaf node, concluding that the elements of the pair are correferent. Let us consider now the pair (Bill Gates, Microsoft). The two elements of the pair are singular but since "Microsoft" is neutral and "Bill Gates" is male, there is no agreement in gender. Thus, we chose the right path of the tree and get to a leaf node indicating that the elements of the pair are not correferent.

Coreference resolution and Normalization are the less generic tasks of the Information Extraction process (Applet & Israel, 1999), since they use heuristics and rules that are specific to the data domain.

2.4 Evaluation of Information Extraction Systems

Between 1987 and 1997 some conferences took place where Information Extraction Systems were evaluated. These conferences were called Message Understanding Conferences (MUC) and were founded by DARPA [2] to encourage the development of new and better Information Extraction techniques.

In addition to the development that these conferences brought to this area, MUCs were important because they produced a consensus on how the Information Extraction systems should be evaluated. Initially, the measures used were based in the metrics from Information Retrieval: recall and precision. Although the names remain, the method for calculating the measures were amended in order to consider the general cases of Information Extraction. This Section defines these measures.

2.4.1 Recall

Recall gives the ratio between the amount of information correctly extracted from the texts and the information available in texts. Thus, recall measures the amount of relevant information extracted and is given by Equation (2.10):

$$recall = \frac{C}{P} \tag{2.10}$$

[2]http://www.darpa.mil/

where C represents the number of correctly extracted records while P represents the total number of records that should be filled.

The disadvantage of this measure is the fact that it returns high values when we extract all the correct and incorrect information from the text.

2.4.2 Precision

Precision is the ratio between the amount of information correctly extracted from the texts and all the information extracted. The precision is then a measure of confidence on the information extracted and is given by Equation (2.11):

$$precision = \frac{C}{C + I + O} \tag{2.11}$$

where C represents the number of records correctly filled, I represents the number of records incorrectly filled and O represents the number of records that have been filled and should not have been.

The disadvantage of this measure is that we can get high results extracting only information that we are sure to be right and ignoring information that are in the text and may be relevant

2.4.3 F-measure

The values of recall and precision may enter in conflict. When we try to increase the recall, the value of precision may decrease and vice versa. The *F-measure* was adopted to measure the general performance of a system, balancing the values of recall and precision. It is given by Equation (2.12):

$$Fmeasure = \frac{(\beta^2 + 1) \times P \times R}{\beta^2 \times P + R} \tag{2.12}$$

where R represents the recall, P represents the precision, β is an adaptation value of the equation that allows to define the relative weight of recall and precision. The value β can be interpreted as the number of times that the recall is more important than accuracy. A value for β that is often used is 1, in order to give the same weight to recall and precision. In this case, the *F-measure* value is obtained through Equation (2.13):

$$F1 = \frac{2 \times P \times R}{P + R} \tag{2.13}$$

3 Related work

There are several approaches to develop an Information Extraction solution. In fact, an Information Extraction program can be developed by using one of the following tool: *(i)* a programming language; *(ii)* a declarative language for Information Extraction; or *(iii)* an ETL (*Extract Transform and Load*) tool. In Section 1.2, we described the advantages and disadvantages of each of these approaches. Table 3.1 presents a summary of this description.

Table 3.1: Advantages and disadvantages of the available solutions to specify Information Extraction programs

	Programming languages	Declarative language	ETL tools
Expressivity	+	-	+/-
IE techniques	+	+/-	-
Maintainability	-	+	-
Learning Curve	-	+	-
Development time	-	+	-
Automatic optimization	-	+	-

This chapter presents two types of state-of-the-art solutions available to develop an Information Extraction program. Since ETL tools have disadvantages that make them not a competitive solution against Information Extraction frameworks and declarative languages, we will not present them. Instead, we start by presenting an overview of some of the most used Information Extraction frameworks in Section 3.1. Then we describe the main declarative language proposals for Information Extraction in Section 3.2.

3.1 Information Extraction frameworks

In the last decade, some frameworks to specify and execute Information Extraction programs have appeared. These frameworks usually offer a set of pre-defined modules that make it possible that an Information Extraction program does not need to be created from scratch. Instead, it results from the composition of different modules where each module implements an algorithm to perform an Information Extraction task. The big advantage of these frameworks is the fact that it makes it easier to write programs by avoiding the implementation of complex Information Extraction algorithms from scratch. Due to its modular nature, refining the resulting Information Extraction programs is also easier. The disadvantage of these frameworks consists on the difficulty to optimize the programs written. In fact, the modules are pre-defined, thus, it is not possible to change the execution algorithm.

In this section, we present the following four Information Extraction frameworks: Minorthird[1], Lingpipe[2], Mallet[3] and Gate[4] (Cunningham et al., 2002). There are many other Information Extraction frameworks (e.g., NLTK[5], OpenNLP[6], YamCha[7]) that we could present here. Unfortunately, it would be impossible to talk about all of them so we needed to make a choice. We present Minorthird because of the wide variety of machine learning techniques supported. Lingpipe supports a large number of Information Extraction tasks and supports both machine learning and rule-based approaches to Information Extraction. Mallet was chosen as an example of a framework that allows to optimize the code by adding code that divides the training of machine learning models into several threads. Finally, Gate is presented to show a framework that divides the Information Extraction activity into tasks and associates an operator to each task.

3.1.1 Minorthird

Minorthird[8] (Cohen, 2004) is a collection of Java classes that allows to perform document classification and sequencial classification using machine learning algorithms. Minorthird offers a wide variety of algorithms for Classification including HMM, CRF, MEMM and SVM (see Section 2.3.2 for a detailed explanation of each technique).

Minorthird contains loaders for several formats of text but the recommended format for input data is plain text with XML tags classifying certain segments. Minorthird does not allow the user to define how a text should be segmented. It always segments a text using spaces and punctuation marks.

An useful functionality of Minorthird is its capability to automatically extract additional information about each text segment. In fact, it supports the extraction of information about whether a segment is written in capital letters, the size of a segment or a pattern that is matched by the segment (e.g., the pattern "[A-Z]+[a-z]" is matched by the segment "Joe"). This functionality makes it easier for the user to take maximum advantage of the capabilities of CRF, MEMM and SVM in the Classification process (Section 2.3.2).

Besides the Java API, Minorthird also offers a Graphical User Interface that allows to test its functionalities in a visual way. For example, it is possible to train and test a Classification model and then

[1]http://minorthird.sourceforge.net/
[2]http://alias-i.com/lingpipe/
[3]http://mallet.cs.umass.edu/
[4]http://gate.ac.uk/ie/
[5]http://www.nltk.org/code
[6]http://opennlp.sourceforge.net/
[7]http://chasen.org/ taku/software/yamcha/
[8]http://minorthird.sourceforge.net/

visualize the text with the classified segments highlighted with a different color. This is an interesting functionality for a user who just wants to perform some experiments.

Minorthird is a good solution for performing some Information Extraction experiments using different machine learning techniques. Unfortunately, since it is a framework for specific machine learning techniques, it does not support any rule-based technique.

3.1.2 Lingpipe

Lingpipe[9] is a Java framework that supports Natural Language Processing tasks like tokenization, sentence detection, named entity detection, coreference resolution, classification, part-of-speech tagging and fuzzy dictionary matching. Lingpipe also contains some classes that allow to load corpus of different formats such as NCBI's MedTag corpora (Smith et al., 2005), the Brown Corpus[10] or the Linguistic Data Consortium's English Gigaword Corpus[11]. In this section we focus on the modules that can be used for the Information Extraction tasks introduced in Section 2.3.

Lingpipe supports the Segmentation task through rule-based and statistical techniques. The rule-based approach is used for sentence detection, in which we consider that an atomic segment is a sentence. Rule-based models for sentence detection use the following three sets of strings: *(i) Possible stops*: segments that are allowed to be the last token of a sentence (e.g., ".", "!", "?"); *(ii) Impossible Penultimates*: segments that are not allowed to be the penultimate token of the sentence (e.g., "Mr"); and *(iii) Impossible Starts*: segments that are not allowed to be in the beginning of the sentence (e.g., ".", "!", "?").

The statistical approach to Segmentation uses n-grams and edit distance and handle a Segmentation task as an orthographic correction task. Lingpipe uses a segmented corpus to train an orthographic correction model. This model treats the non-existence of spaces in the corpus as orthographical errors, proposing the insertion of spaces in the text. There are two demos available in Lingpipe that use this Segmentation approach. One of them is used for Chinese language word Segmentation (see Section 2.3.1) and the other is used for hyphenization and sylabization in the English language.

Lingpipe offers different approaches to perform the Classification task. Two rule-based approaches can be used: *regular expressions* and *dictionary-based* (Section 2.3.2). The only machine learning technique available for the Classification task is HMM (Section 2.3.2).

Coreference resolution in Lingpipe is rule-based and can be used for English texts. It uses three types of information namely gender, honorifics in the neighborhood of the segment and the fact that a segment is a pronoun for an approach similar to the one described in Section 2.9.

[9]http://alias-i.com/lingpipe/
[10]http://khnt.aksis.uib.no/icame/manuals/brown/
[11]http://www.ldc.upenn.edu/

Lingpipe has the advantage of supporting both rule-based and machine learning approaches for Information Extraction. Since there is a big number of demos and tutorials available with the framework, it is easy to learn how to use it. The main drawback of Lingpipe is that the variety of machine learning techniques offered is not large (in comparison with Minorthird).

3.1.3 Mallet

Mallet (A. K. McCallum, 2002) is a Java framework that offers implementations of machine learning algorithms for Natural Language activities. The only Information Extraction task that this framework performs is Classification. Mallet offers three algorithms for Classification: HMM, MEMM and CRF (see Section 2.3.2 for a detailed explanation of each technique).

The training and testing corpus used in Mallet needs to be manually segmented. The format accepted is plain text with one segment per line. If we want to use additional features besides the value of the word in the Classification process (e.g., capitalization, segment size), the extraction of the features must be done manually.

The big advantage of Mallet is the execution time of the supported techniques, specially CRF. For example, it is possible to write a small piece of code that divides the training process of a CRF through different threads allowing to take advantage of the multi-core processors (remember that the main drawback of CRF is the training time). The major limitations of Mallet are: *(i)* the fact that it only supports machine learning techniques; *(ii)* the only Information Extraction task supported is Classification; *(iii)* the training and testing corpus must be manually segmented.

3.1.4 Gate

Gate (Cunningham et al., 2002) is an infrastructure for Natural Language Processing software development. Gate provides some reusable modules for different activities (some of them for Information Extraction). These modules are all extendable and customizable so that the programmers can adapt them for their own needs.

The Information Extraction techniques supported in Gate are included in a module called ANNIE (standing for *A Nearly-New Information Extraction* system). ANNIE consists of a set of modules that allow to perform Information Extraction tasks we mentioned in Section 2.3 (Cunningham et al., 2009). In the rest of this section, the following modules are described: the *tokenizer*, the *sentence splitter*, the *gazetteers*, the *Part-of-Speech tagger*, the *semantic tagger*, the *OrthoMatcher* and the *pronomial coreference*.

The *tokenizer* performs the Segmentation task as described in Section 2.3.1. The Gate tokenizer uses regular expressions to specify a model that splits a text into tokens such as numbers, punctuation and words. A *tokenizer* rule has a left hand side and a right hand side. The left hand side is a regular expression that has to be matched by the input text. The right hand side describes features that may be added to the segment (e.g., features that may be used in other modules).

The *sentence splitter* also performs the Segmentation task but it splits the text into sentences. The splitter performs a Segmentation task using punctuation or line breaks instead of splitting it into words. The splitter uses a list of abbreviations to help determining if a punctuation mark is associated to an abbreviation or if it is a sentence breaker. Gate also offers a *sentence splitter* called *regex sentence splitter* that uses regular expressions to perform the sentence split.

The *gazetteers* are lists of plain text files containing one token per line. Each list represents a set of tokens of a given class (e.g., names of cities, organizations) so it is associated with a given type. The gazetteers can be used to perform the dictionary-based Classification task as described in Section 2.3.2.

The *Part-of-Speech tagger* is a modified version of the Brill Tagger (Brill, 1992). The Brill Tagger is a rule-based *Part-of-Speech tagger* which automatically generates its rules and tags using supervised learning. Gate's version uses a default lexicon and a model resulting from training with a large corpus taken from the Wall Street Journal. The *semantic tagger* uses regular expression rules which act on previously acquired features in order to perform a Classification task over the input text.

Both the *OrthoMatcher* and the *pronomial coreference* perform a Coreference resolution task as described in Section 2.9. The *OrthoMatcher* resolves coreference between entities tagged by the *semantic tagger* by using a rule-based approach. The *pronomial coreference* is responsible for the anaphora resolution.

An Information Extraction activity can be specified in the Graphical User Interface by pipelining some of the modules described above. It is possible to enrich Gate using different plugins like a machine learning plugin and a plugin that connects Gate to Minorthird.

3.1.5 Discussion

In this section, we compare the Information Extraction frameworks described in the previous sections (Table 3.2). Our analysis is based on the following criteria: *(i) IE tasks*: the list of Information Extraction tasks that the framework supports, according to the corresponding description presented in Section 2.3; *(ii) IE approaches*: the Information Extraction approaches that the framework supports, according to the corresponding description presented in Section 2.2; *(iii) Multiple IE techniques*: boolean value that indicates whether the framework offers more that two techniques for each task; *(iv) Automatic Optimization*: boolean value that indicates whether the framework is open for automatic optimization; and *(v) Text type*: the text types the Information Extraction techniques supported by the framework can be applied to (unstructured or semi-structured).

Table 3.2: Comparison between Information Extraction frameworks

	Minorthird	Lingpipe	Mallet	Gate
IE tasks	Classification	Segmentation, Classification, Coreference	Classification	Segmentation, Classification, Association, Coreference
IE Approaches	Machine Learning	Machine Learning, Rules-based	Machine Learning	Rules-based
Multiple IE techniques	yes	no	yes	no
Automatic Optimization	no	no	no	no
Text type	Unstructured or Semi-structured	Unstructured or Semi-structured	Unstructured or Semi-structured	Unstructured or Semi-structured

Minorthird only performs the Classification task and the only techniques available are based on machine learning. In reality, the advantage of Minorthird is the fact that it is so simple that performing simple experiments with it becomes easy. The number of machine learning techniques available is large, thus allowing the user to easily try different techniques. The recommended format for the corpus is very intuitive and data sets obeying to this format are available with the Minorthird distribution.

Lingpipe is a very comprehensive software package that performs different Information Extraction tasks and supports both rule-based and machine learning techniques. Even though this framework is not as easy to use as Minorthird, the demos and tutorials available in the Web site turn it easy to learn. The main limitation of Lingpipe is the fact that for each Information Extraction task it usually offers only one rule-based technique and one machine learning technique. This is a limited when compared to what Minorthird offers for the classification.

Analogously to Minorthird, Mallet supports only Classification tasks based on machine learning. The advantage of Mallet over Minorthird is the fact that it is possible to distribute the training of Classification models through several threads in order to take advantage of the multi-core processors that are widely used these days. Even with this optimization, Mallet is not automatically optimizable because it is necessary for the user to write specific code to use this optimization. Mallet is not as easy to use as Minorthird because the corpus format must be divided with one segment per line.

Gate is the most comprehensive framework of the ones analyzed. It performs the biggest number of Information Extraction tasks. Even though it uses rule-based techniques by default, some plugins to support machine learning techniques can be installed. Gate offers pre-defined software modules for specific tasks and not for techniques, unlike the other three frameworks. An Information Extraction solution is achieved by pipelining these modules. The resulting Information Extraction programs are easily maintained and the user can manipulate the modules according to the application requirements.

3.2 Declarative Information Extraction

An alternative approach to specify an Information Extraction program is to use a declarative language. A declarative language is used to specify what someone wants to obtain as a result of the program. Then, there exists a software component (e.g., an optimizer) that determines the best solution that meets the specification requirements. This way, it is possible to conceive multiple execution algorithms for a given logical specification, thus opening the possibility to choose the more efficient algorithm for each situation. Declarative languages also offer a high level of abstraction, and the programmer does not need to be an expert in programming to develop a program. The disadvantage of these languages is that they usually have limited expressivity and do not give the possibility to execute a large variety of tasks.

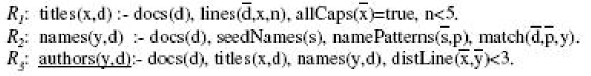
In the literature, there are two widely used types of declarative languages for Information Extraction: *(i)* based on Datalog; *(ii)* based on Relational Algebra. In this section, we present the following three declarative languages based on Datalog: XLog (Shen et al., 2007), ALog (Shen et al., 2008), Elog (Gottlob et al., 2004). We also present one extensions of the Relational Algebra that can be used for Information Extraction: System T (Reiss et al., 2008).

3.2.1 XLog

XLog (Shen et al., 2007) is a language based on Datalog (Ceri et al., 1989). The XLog language supports the specification of Information Extraction programs through procedural predicates, called *p-predicates*, that are embedded in the language. These predicates have the format $q(\bar{a}_1, ..., \bar{a}_n, b_1, ..., b_m)$ where a_i and b_i are variables. It receives a tuple $(a_1, ..., a_n)$ and produces a set of tuples with the format $(a_1, ..., a_n, b_1, ..., b_m)$. In this section, we consider that the input variables are represented with a bar above the symbol. A p-predicate is associated to a procedure written in a procedural language. In addition to the *p-predicates* already available in the language, XLog offers the possibility to integrate p-predicates developed by the user.

To understand the XLog semantics, we introduce some basic concepts. An important concept is the notion of *string* which is a sequence of characters. Another important concept is a *document*, which is represented by a tuple (*id, content*), where *id* is a key and *content* is a set of characters that represents the text of the document. A *document span* is a substring of the document represented by a tuple (*id, doc, begin, end*) where *id* is a key, *doc* is the id of the document, *begin* and *end* are the beginning and ending positions of the content in the document *doc* respectively.

Figure 3.1 presents an example of an XLog program (Shen et al., 2007) that we use to illustrate the characteristics of the language in what follows. This program extracts the name of the authors from texts that are the output of the *docs* predicate. In this example, one of the p-predicates used is $lines(\bar{d}, x, n)$. It receives a document *d* as input and produces all the lines *x* of the document assigning a number *n* to each line.



R_1: titles(x,d) :- docs(d), lines($\bar{d}$,x,n), allCaps($\bar{x}$)=true, n<5.
R_2: names(y,d) :- docs(d), seedNames(s), namePatterns($\bar{s}$,p), match($\bar{d}$,$\bar{p}$,y).
R_3: authors(y,d):- docs(d), titles(x,d), names(y,d), distLine($\bar{x}$,$\bar{y}$)<3.

Figure 3.1: An XLog program

XLog supports the notion of an *IE Predicate*, that is important for specifying Information Extraction programs. An *IE Predicate* is a *p-predicate* $q(\bar{a}_1, ..., \bar{a}_n, b_1, ..., b_m)$ where each a_i is a document or a document span and each b_i is a document span. For every output tuple $(a_1, ..., a_n, b_1, ..., b_m)$, there is at least one b_j that is contained in some a_i, which means that b_j was extracted from a_i.

In Figure 3.1, the *p-predicate* $lines(\bar{d}, x, n)$, in rule R1, is an *IE Predicate*. Given a document d, it produces all the lines of the document. However, the $namePatterns(\bar{s}, p)$ *p-predicate* in rule R2 is not an *IE Predicate* since it does not receive as input neither a document nor a document span. An important pre-defined *predicate* is *docs*(d). This predicate produces tuples with keys for all the documents that are used in the Information Extraction process. This predicate must appear in at least one rule of a XLog program.

In addition to the predicates, another important element of the language is a procedural function (*p-function*). A *p-function* returns a value instead of producing a set of tuples, as does a *p-predicate*. An example of a *p-function* in the program of Figure 3.1 is $distLine(\bar{x}, \bar{y})$, which is present in rule R3 and returns the distance between two lines, x and y. As we can see, the result of a *p-function* is typically compared with a given value.

Now, we analyze the semantics of the whole Information Extraction program represented in Figure 3.1. Rule R1 extracts the title of each document. The pre-defined *IE predicate* $lines(\bar{d}, x, n)$, as we referred before, produces all the lines of a given document and a number for each line. This rule extracts a title composed by the lines whose letters are all capital ($allCaps(\bar{x})$ which is a pre-defined function) and that are among the first five lines of the document (n<5). Rule R2 extracts names from a set of documents. The pre-defined *p-predicate* $seedNames(s)$ produces a set of tuples containing typical English names. The user-defined predicate $namePatterns(\bar{s}, p)$ produces a set of patterns corresponding to elements of names produced by $seedNames(s)$. These patterns are given as input to the pre-defined predicate $match(\bar{d}, \bar{p}, y)$ which produces only the document spans that match any pattern of *p*. Rule R3 uses the previous rules to extract the authors of a given document. In this case, an author is any name at a distance of the title of less than three lines ($distLine(\bar{x}, \bar{y})$<3, where $distLine(\bar{x}, \bar{y})$ is a pre-defined predicate).

XLog is appropriate for writing Information Extraction procedural modules. It is simple to use thanks to the pre-defined set of predicates and functions. These predicates and functions cover some of the tasks related to text processing. XLog is very flexible, as it supports any Information Extraction program. In the worst case, the Information Extraction activity is implemented as a *p-predicate* that is then used in a XLog program. Another advantage of XLog is the fact that it inherits the Datalog well-defined semantics. This characteristic makes XLog amenable to optimization techniques from Datalog described in (Shen et al., 2007) and makes the language completely extendable.

However, in some situations, XLog pre-defined predicates are not enough to develop a given Information Extraction program. In these cases, it is necessary to develop a *p-predicate*. To do so, the user starts to examine a set of data and writes rules that allow to perform the extraction. Then he builds the *p-predicate* to execute them. The construction of new predicates is very laborious and requires a long time for refining until the user gets good results.

3.2.2 ALog

The ALog language (Shen et al., 2008) is an extension of XLog that tries to overcome the complexity involved in the development of Information Extraction predicates to a specific domain. ALog provides support to declare *domain constraints* associated to text segments with little extra programming and write *approximate Information Extraction programs* in a way that the results can improve iteratively.

We start by introducing the basic concepts to understand the ALog semantics. We first introduce the concept of *approximate Information Extraction program* which is a program that we assume will produce some wrong results but will be used as the base to develop a program with better results.

A *domain constraint* is a condition that every tuple resulting from an Information Extraction program must fulfill. *Domain restrictions* are declared in description rules. A *description rule* has the form $q : -q_1, ..., q_n$ just like a traditional Xlog rule, except that the head q is an IE predicate. The semantics of a description rule is similar to that of an Xlog rule, but these rules only produce a relation when all the required input is introduced. That is, whenever we assign constant values to the input variables in the head of the rule, it produces a new relation. Each of these rules can be seen as an approximate extraction rule because they produce relations with extraction results that may not be accurate. In the example of Figure 3.2, the rule returns all the numeric elements: some correspond to a price and others do not. The user uses the approximate results to refine the program, increasing the number of description rules until he is satisfied with the result.

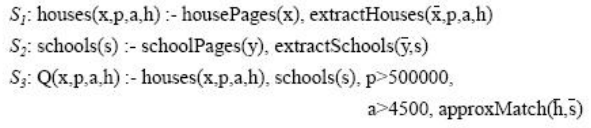
S_1: extractPrices($\bar{d}$,p) :- from($\bar{d}$,p), numeric($\bar{p}$)=yes

Figure 3.2: Simple description rule in ALog

Since the results of an ALog program are approximate, ALog supports an *annotation mechanism* to declare the kind of approximation that is performed in each rule. There are two kinds of annotations that can be used for this declaration: *existence annotation* and *attribute annotation*. An *existence annotation* indicates that each tuple in a relation produced by a rule may or may not exist in the final result. An *attribute annotation* indicates that an attribute takes a value from a given set but we are not sure about which value.

(Shen et al., 2008) proposes a methodology for developing an ALog program. First, a user should start by writing an initial XLog program which is a list of all the necessary IE predicates without associating procedures to them. Then, the user should implement each of these IE predicates predicates aiming at improving the results by iteratively adding new *domain restriction* to the *description rules*.

Figure 3.3 (Shen et al., 2008) presents an ALog approximate program in which the *IE predicates extractHouses* and *extractSchools* are approximate rules which means that there is no procedural predicate associated to them. This program extracts information about houses with a school nearby, a price above 500000 and an area above 4500. Rule S1 extracts information about houses. The predicate *housePages* extracts all the pages containing information about houses. The *IE predicates extractHouses* is expected to produce tuples with information about the houses. Rule S2 is analogous to rule S1 but it extracts information about schools instead of information about houses. Rule S3 uses the previous rules to find houses with a school in the neighborhood (as determined by the predicate *approxMatch*), and filters the price and the area.

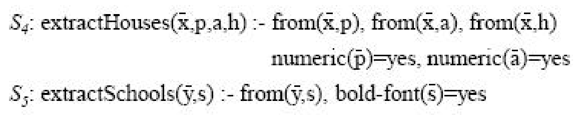


S_1: houses(x,p,a,h) :- housePages(x), extractHouses($\bar{x}$,p,a,h)
S_2: schools(s) :- schoolPages(y), extractSchools($\bar{y}$,s)
S_3: Q(x,p,a,h) :- houses(x,p,a,h), schools(s), p>500000,
a>4500, approxMatch($\bar{h}$,$\bar{s}$)

Figure 3.3: Initial program in ALog for approximate extraction

Figure 3.4, presents approximate rules for the *IE predicates extractHouses* and *extractSchools* that were shown in Figure 3.3. Rule S4 is a description rule that corresponds to the IE predicate *extractHouses*. The *from* predicate produces tuples containing segments from the input text. The function *numeric* returns true is the input is a numeric value. This rule considers that a house, the price and the area are segments of the input text and that the price and the area must be numeric values. Rule S5 is a description rule that corresponds to the IE predicate *extractSchools*. The function *bold-font* returns true if the input uses a bold typeface. This rule considers that a school is any segment of the input text with a bold typeface. Functions like *numeric* and *bold-font* are pre-defined ALog functions.

The ALog language minimizes the problems of XLog by supporting a methodology and a specific syntax to make the writing of *IE predicates* as declarative as possible. Unfortunately, it is not always possible to get good results only with the ALog declarative methodology. In these cases, the development of procedural predicates may be unavoidable.



S_4: extractHouses($\bar{x}$,p,a,h) :- from($\bar{x}$,p), from($\bar{x}$,a), from($\bar{x}$,h)
numeric($\bar{p}$)=yes, numeric($\bar{a}$)=yes
S_5: extractSchools($\bar{y}$,s) :- from($\bar{y}$,s), bold-font($\bar{s}$)=yes

Figure 3.4: Description rules in ALog

3.2.3 ELog

ELog (Gottlob et al., 2004) is a declarative language based on Datalog that is used internally in the Lixto system (Baumgartner et al., 2001). This system supports the development of programs to extract data from the Web (HTML documents). Unlike XLog and ALog, this language is not directly available to the user for the specification of Information Extraction programs.

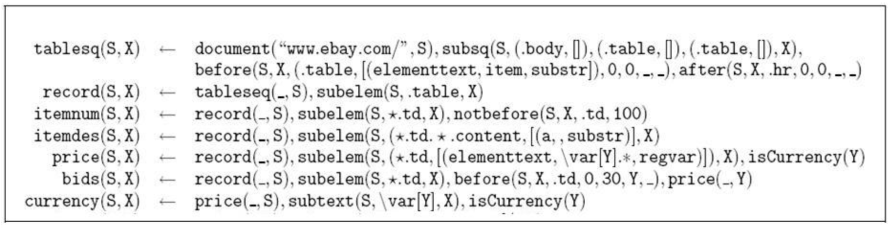
Lixto offers a graphical interface to support the development of Information Extraction programs in an interactive way. Elog is the language used internally to store extraction rules. This language offers two basic mechanisms for data extraction: tree extraction and string extraction. *Tree extraction* supports the navigation through the HTML document using the predicate *subelem*. This predicate specifies paths in the tree using regular expressions or conditions imposed on the values of the node attributes. The *string extraction* mechanism supports the extraction of information from the leafs of the HTML tree, that, usually, do not have so many structural clues that help the extraction process. This mechanism is supported by the predicate *subtext*. The use of this predicate is based on regular expressions. There are two other predicates available, *subsq* and *subatt*, to support the Information Extraction tasks. These predicates do not add any expressivity to the first two but are useful to simplify the specification of the rules. The predicate *subsq* allows to extract subsections of a node, given two children where one represents the beginning of the section and the other one represents the end of the section. The predicate *subatt* supports the extraction of the value of a given HTML attribute.

In addition, ELog offers the following set of predicates to specify conditions: *(i)* context conditions to verify conditions related with the relative position of the elements in the document (e.g., the predicates *before* ou *after*); *(ii)* internal conditions to check syntactic conditions (e.g., the predicate *isDate*) or semantic conditions (e.g., the predicates *isCurrency* and *isCountry*); *(iii)* pattern conditions to indicate whether a node matches a given pattern in the HTML document. A pattern condition can be used through the definition of a path like ".*.td which indicates that the node is the parent of any "td" node; *(iv)* interval conditions to restrict the elements depending on their appearance order. For instance, the condition "[3,7]" indicates that we should only consider from the third to the seventh extracted elements.

Figure 3.5 illustrates an example of an extraction program written with the ELog language (Gottlob et al., 2004). The goal of the program is to extract the list of products for auction from eBay Web pages[12].

[12]http://www.ebay.com/

Each entry of an eBay Web page contains a description of the product, a price in a given currency and the current number of licitations performed. In eBay Web pages, the information concerning each product is presented in an HTML table. The predicate *tablesq* in the first rule extracts sequences of tables. The records of each table are extracted with the predicate *record*. Predicates *itemdes*, *price* (in the forth, fifth and sixth rules respectively) and *bids* are used to choose, respectively, the description of the product, the price and the number of licitations of each record. The predicate *currency* extracts the currency in which the price is indicated.



```
 tablesq(S,X)  ←  document("www.ebay.com/",S), subsq(S,(.body,[]),(.table,[]),(.table,[]),X),
                  before(S,X,(.table,[(elementtext,item,substr]),0,0,_,_),after(S,X,.hr,0,0,_,_)
  record(S,X)  ←  tableseq(_,S), subelem(S,.table,X)
 itemnum(S,X)  ←  record(_,S), subelem(S,⋆.td,X), notbefore(S,X,.td,100)
 itemdes(S,X)  ←  record(_,S), subelem(S,(⋆.td.⋆.content,[(a,,substr)],X)
   price(S,X)  ←  record(_,S), subelem(S,(⋆.td,[(elementtext,\var[Y].∗,regvar)]),X), isCurrency(Y)
    bids(S,X)  ←  record(_,S), subelem(S,⋆.td,X), before(S,X,.td,0,30,Y,_), price(_,Y)
currency(S,X)  ←  price(_,S), subtext(S,\var[Y],X), isCurrency(Y)
```

Figure 3.5: Example of an extraction program with ELog

There are three main drawbacks of Elog. First, it is impossible to extract data that is not available in a Web page. Second, is the fact that it is only available as an internal language of Lixto, which means that it is not possible to directly specify programas using ELog rules. Finally, the expressivity of the ELog rules is affected due to the fact that it has a limited number of pre-defined predicates. If the ELog language was available to specify Information Extraction programs outside the Lixto system, it would be a good alternative to XLog and ALog for data extraction from the Web. For other kinds of documents, its expressive power is not enough due to the fact that the predicates available are only used to navigate in a document structured as a tree.

3.2.4 System T and the AQL Language

IBM's System T (Reiss et al., 2008) offers a data model and a set of operators for rule-based Information Extraction. The data model is a minimal extension of the Relational Model. The System T operators extend the relational algebra supporting text processing tasks. All the basic operations supported by System T read and produce text regions. A *span* is an ordered pair (*begin*, *end*) that denotes a region of a document starting at position *begin* and ending at position *end*. For instance, in the document with the text "Information extraction", the span (13,22) corresponds to the text region "extraction".

The operators of System T can be divided into two categories: span extraction operators and span aggregation operators. A *span extraction operator* identifies text regions that match a given input pattern and returns a relation of *spans* corresponding to each text region. System T offers different kinds of span extraction operators. Two of the more relevant ones are the standard regular expression matcher (ε_{re}) and the dictionary matcher (ε_d). The *standard regular expression matcher* $\varepsilon_{re}(r)$ receives a regular expression *r* and returns all the spans that match *r*. Figure 3.6 shows an example in which a standard regular expression matcher, ε_{re}, consisting of a regular expression for extracting band members, is applied to a document returned by the function *doctext()*. The *dictionary matcher* $\varepsilon_d(dict)$ receives a dictionary *dict* and returns all the spans corresponding to occurrences of some entry in *dict*. Figure 3.7 shows an example in which the dictionary matcher ε_d, that consists of a dictionary of musical instruments, is applied to the document returned by *doctext()*.

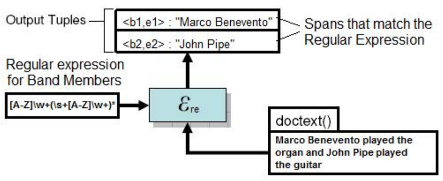

Output Tuples
<b1,e1> : "Marco Benevento"
<b2,e2> : "John Pipe"
Spans that match the Regular Expression
Regular expression for Band Members
[A-Z]\w+(\s+[A-Z]\w+)*
ε_{re}
doctext()
Marco Benevento played the organ and John Pipe played the guitar

Figure 3.6: Span extraction using the regular expression operator

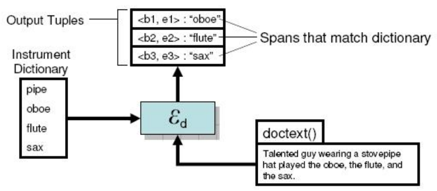


Figure 3.7: Span extraction using the dictionary operator

The span aggregation operators receive a set of input spans and perform aggregation operations over the entire input. The span aggregation operators offered in System T are the containment consolidation (Ω_c), the overlap consolidation (Ω_o) and the block (β). The first two are consolidation operators and are used to solve overlapping situations resulting from the use of multiple patterns to identify the same concept. $\Omega_c(r)$ accepts a relation *r* as input and discards spans that are totally contained inside another span of *r*. Figure 3.8 shows an application of Ω_c. The input relation contains three spans which are overlapped. The spans "Organ" and "Pipe" are included in the third span "Organ Pipe", so Ω_c only returns the span "Organ Pipe". The overlap consolidation operation $\Omega_o(r)$ receives a relation r and produces new spans by merging overlapping spans. Figure 3.9 shows an application of Ω_o. The input relation contains two spans that are overlapped. The span "Acoustic bass" and the span "bass guitar" contain the common sub-span "bass", so $\Omega_o(r)$ only returns the span "Acoustic bass guitar".

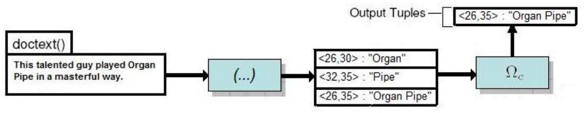


Figure 3.8: Overlap resolution using the containment consolidation operator

The block operator $\beta(n, d, r)$ receives two integers *n* and *d* and a relation *r* as input. It identifies the smallest text regions that contain a maximum number of *n* spans of *r* and that have a maximum distance

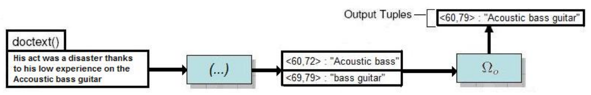

Output Tuples
<60,79> : "Acoustic bass guitar"
doctext()
His act was a disaster thanks to his low experience on the Accoustic bass guitar
(...)
<60,72> : "Acoustic bass"
<69,79> : "bass guitar"
Ω_o

Figure 3.9: Overlap resolution using the overlap consolidation operator

of d characters between them. Figure 3.9 shows an example in which β is applied with a value of 2 for n and 2 for d. The input relation contains the spans "When", "John" and "Pipe". These three spans are separated by a space character from the following span in the original text. The result contains the spans contained originally in the relation and also two new spans that result from merging "When" and "John" and merging "John" and "Pipe".

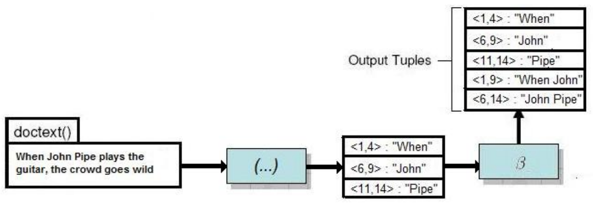


Figure 3.10: Merging text spans using the block operator

System T algebra is implemented by a SQL-like language named AQL, standing by Annotation Query Language. In 2007, the compiler and the optimizer were implemented. During 2008, AQL was introduced in some IBM products like Lotus Notes 8.01[13].

Compiling a query in AQL results in an execution plan that uses the operators of System T. Consider as an example the definition of the AQL view illustrated in Figure 3.11. The semantics of this view consists in associating spans concerning people names to spans concerning their corresponding phone number. The *select* and *from* clauses have the same semantics as in SQL. In this case, we assume that an annotator already extracted spans referring to Person, PhoneNumber and Sentence and inserted them in the corresponding relations with the same name. The relationship between people and their phone numbers is achieved using the predicates in the *where* clause. The predicate *Follows* receives two spans (*firstSpan* and *secondSpan*) and two integers (*minDistance* and *maxDistance*) and returns true if *firstSpan* and *secondSpan* are separated by a distance between *minDistance* and *maxDistance*. The predicate *Contains* receives two spans (*firstSpan* and *secondSpan*) and returns true if *firstSpan* contains *secondSpan*. The predicate *ContainsRegex* receives a regular expression (*regex*) and a span (*span*) and returns true if a *subspan* of *span* matches *regex*.

In this example, the predicate *Follows* is used to indicate that a span associated to a Person must be at a distance between 0 and 30 characters of the corresponding PhoneNumber. The predicate *Contains*

[13]http://www-01.ibm.com/software/lotus/products/notes/

```
create view PersonPhone as
select P.name as person, N.number as phone
from Person P, PhoneNumber N, Sentence S
where
    Follows(P.name. N.number, 0, 30)
    and Contains(S.sentence, P.name)
    and Contains(S.sentence, N.number)
    and ContainsRegex(/\b(phone|at)\b/,
         SpanBetween(P.name, N.number));
```

Figure 3.11: AQL view to associate People to their Phone Number

is used to force the Person and the PhoneNumber to be in the same sentence. Finally, the predicate *ContainsRegex* indicates that there must be an indicator of a phone like "phone" or "at" in the space between a Person and a PhoneNumber.

AQL is formalized as an extension of the Relational Algebra thus inheriting the advantages of the relational query languages. Its syntax is based on SQL which turns it more familiar to the users that are used to relational technology. Due to the support of regular expressions, dictionaries and the predicates that allow to solve overlap problems, AQL has a very good expressive power. The big inconvenient of the AQL language is the fact that it was developed as a framework for rule-based Information Extraction. In fact, AQL does not support the use of machine learning techniques.

3.2.5 Discussion

In this section, we compare the declarative Information Extraction approaches described in the previous sections. Our analysis is based on the following criteria: *(i) IE approaches*: the Information Extraction approaches that the framework supports, according to the corresponding description presented in Section 2.2; *(ii) Predicates/operators*: types of predicates used for Information Extraction; *(iii) Automatic Optimization*: boolean value that indicates whether the framework is open for automatic optimization; and *(iv) Text type*: the text types the Information Extraction techniques supported by the framework can be applied to (unstructured or semi-structured).

Note that the criteria are not exactly the same as the ones used in the comparison of Information Extraction frameworks. We do not consider the *Information Extraction tasks* supported by each language because these languages are not oriented to the Information Extraction tasks. Instead, they support a set of pre-defined predicates or operators that are oriented to common text processing tasks or HTML processing. Therefore, we should add a new criterion called *Predicates/operators* that indicates whether the predicates or operators of the language are used for text processing or HTML navigation.

XLog is a declarative language that allows the user to extract text from several document types. XLog is highly extensible, being possible for a user to develop new predicates with a procedural language. The biggest disadvantage of XLog is that the process of developing new predicates is very laborious and requires a long time of refining until the user gets good results.

Table 3.3: Comparison of declarative Information Extraction approaches

	XLog	ALog	ELog	System T
IE Approaches	Rules-based	Rules-based	Rules-based	Rules-based
Predicates/ operators	Text processing	Text processing	HTML navigation	Text processing
Automatic Optimization	+	+	+	+
Text type	Unstructured or Semi-structured	Unstructured or Semi-structured	Semi-structured (HTML)	Unstructured or Semi-structured

ALog is an extension of Xlog that tries to overcome the inconvenient of developing of new predicates. It offers a methodology that allows a user to iteratively develop an Information Extraction program by adding new conditions to the rules. If, after some iterations, the user is still not satisfied with the result, he develops a new predicate using a procedural language.

ELog is also an extension of Datalog but it is not as powerful as XLog and ALog. The biggest drawback of this language is the fact that Information Extraction activities can be applied only to Web documents. Another disadvantage of this language is that it is the internal language of an Information Extraction system called Lixto, and thus it is not directly available to the user.

System T is an extension of the Relational Algebra that offers several operators for Information Extraction. The algebra of the System T operators is implemented by the AQL language, a language based on SQL, which makes it more familiar to the relational technologies users. A disadvantage of AQL is the fact that it only supports operators for rule-based Information Extraction.

4 The E-txt2db framework for classification

In Chapter 3, we described two state-of-the-art approaches for specifying and executing Information Extraction programs, namely Information Extraction frameworks and declarative languages for Information Extraction. *Information Extraction frameworks* allow a user to easily specify and maintain Information Extraction programs using pre-implemented modules for several Information Extraction techniques. The disadvantages of these frameworks are the following: *(i)* difficulty to optimize the resulting programs and *(ii)* a high learning curve. An Information Extraction specification using a *declarative language* is amenable to optimization. Due to the high level of abstraction supported by declarative languages, they are usually very easy to learn. Analogously to the Information Extraction frameworks, the resulting code is highly reusable due to the pre-implemented predicates/operators offered. The drawbacks of this approach are the following two: *(i)* a limited expressive power and *(ii)* support of a limited number of Information Extraction techniques.

This thesis proposes a solution that combines the advantages of both Information Extraction frameworks and the declarative languages for Information Extraction. We propose a framework for Information Extraction named E-txt2db that offers a declarative operator for each Information Extraction task described in Section 2.3. Due to time constraints, we could not focus our effort in operators for all the Information Extraction tasks. Therefore, we decided to perform a proof-of-concept with a single operator for the Classification task.

This chapter describes the E-txt2db framework for the Classification task. First, we present an overview of the E-txt2db framework in Section 4.1. Second, we present the semantics of the Classification operator in Section 4.2. In Section 4.3, we describe the Java API of the CEE (Creation, Execution and Evaluation) engine that implements the semantics of the Classification operator. Finally, in Section 4.4, we propose of an SQL-like syntax for the Classification operator that supports the creation, execution and evaluation of Classification models.

4.1 The E-txt2db framework

E-txt2db is a framework for specifying and executing Information Extraction programs. This framework offers a declarative operator for each task that composes an Information Extraction process, namely Segmentation, Classification, Association, Normalization and Coreference Resolution, as described in

Section 2.3. For each operator, there is a set of pre-implemented techniques available to create models that describe how an Information Extraction task is executed (extraction models).

In order to specify an Information Extraction process using E-txt2db, the user writes a specification using an SQL-like language. There is a specific syntax for creating, executing and evaluating a model for each Information Extraction operator. The user that creates a Classification model chooses one of the pre-implemented techniques available according to the needs of the model he wants to develop. After the creation, he stores the model for further use. Someone who uses a previously created Classification model does not need to know which technique it uses.

The approach used by E-txt2db mixes the advantages of the Information Extraction frameworks and the declarative languages for Information Extraction described in Chapter 3. Similarly to the Information Extraction frameworks, the fact that there are several pre-implemented techniques for each operator turns it easy to specify and maintain Information Extraction programs. The code produced is highly reusable since the created extraction models are stored for further reuse. E-txt2db offers a declarative SQL-like language for specifying programs. Therefore, programs are amenable to optimization. E-txt2db supports a set of operators for extracting information. These operators can be composed to produce an Information Extraction program. This way, the user is able to specify an Information Extraction activity at a high level of abstraction. As a consequence, Information Extraction programs are easily specified.

The expressive power of E-txt2db is limited because the library of techniques available is not exhaustive. However, the E-txt2db library of techniques is easily extensible. In fact, new techniques can be added by creating one new Java class and adapting the SQL-like syntax to support it.

Figure 4.1 presents the architecture of the instantiation of the E-txt2db framework for the classification task. The output consists of relevant data extracted from the input and annotated according to classes. The architecture encloses three main modules: the Java CEE Engine, the library of techniques, and the parser. The text segmentation module corresponds to a basic implementation of the segmentation task. The Java CEE Engine is responsible for the main functionalities of the classification operator: Creation, Execution and Evaluation (CEE) of classification models. The available techniques for each operator are stored in the library of techniques. When creating a model, the Java CEE Engine accesses the library of techniques in order to use the algorithms available to implement the chosen technique. The module responsible for interpreting the specification is the parser. The parser accepts as input the specification program in the corresponding SQL-like syntax and translates it in order to call methods of the Java CEE Engine.

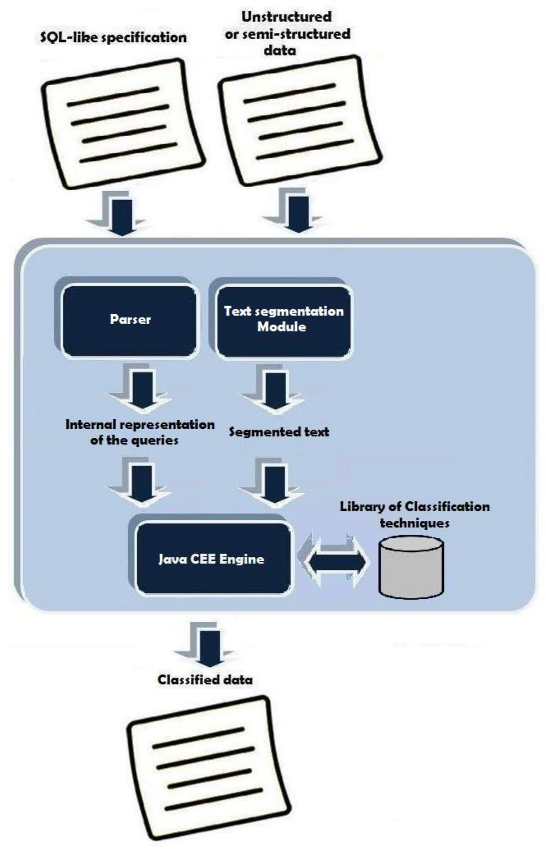

SQL-like specification
Unstructured or semi-structured data
Parser
Text segmentation Module
Internal representation of the queries
Segmented text
Library of Classification techniques
Java CEE Engine
Classified data

Figure 4.1: Architecture of the E-txt2db framework for classification

4.2 The Classification operator

This section describes the semantics of the Classification operator. First, we introduce some basic concepts. Second, we present the semantics of the Classification operator and an example.

4.2.1 Basic definitions

Let $\mathcal{C}$ be a set of symbols. Elements of $\mathcal{C}$ are called *characters*. We denote $\mathcal{D}_\mathcal{C}$ the set of all ordered multisets of $\mathcal{C}$. Each element S of $\mathcal{D}_\mathcal{C}$ is called a *character sequence*. The *length* of a character sequence S, denoted *length(S)*, is the number of characters in the sequence. The length of a character sequence must be an integer greater than or equal to 0.

Consider that S is a character sequence that can be represented as the concatenation of three other character sequences S_1, S_2 and S_3. Each of the character sequences S_i is called a *segment* of the character sequence S (Hansen, 1989). A *partition* of a character sequence S, $\mathcal{P}(S)$, is a set of segments of S such that the concatenation of all the character sequences of $\mathcal{P}(S)$ equals the character sequence S. A *classified segment* or *annotated segment* is a pair *<s,a>* where *a* is a class that is assigned to the segment *s*. A *classified* *document* or *annotated document* is a character sequence in which some of the segments are classified.

A Classification model $\mathcal{M}$ corresponds to a description of how to assign a class a to each segment of a character sequence s in order to produce a classified segment. For a given set of symbols $\mathcal{C}$, a Classification model $\mathcal{M}$ is a pair $<\mathcal{A}$,$\mathfrak{f}>$ where $\mathcal{A}$ is a set of classes that can be assigned to text segments and the function $\mathfrak{f}$: $\mathcal{D}_\mathcal{C} \rightarrow \mathcal{D}_\mathcal{C} \times \mathcal{A}$ is a computable function that describes how each segment of a given character sequence shall be classified.

4.2.2 Semantics of the Classification operator

The *Classification operator*, $\tau_\mathcal{M}$, is responsible for assigning a class to each element of a character sequence using a given Classification model $\mathcal{M}$ given by $<\mathcal{A}$,$\mathfrak{f}>$. Given a character sequence S and a set of classes $\mathcal{B} \subset \mathcal{A}$, $\tau_\mathcal{M}(S,\mathcal{B})$ assigns elements of the set $\mathcal{B}$ to segments of the character sequence S according to the function $\mathfrak{f}$. The Classification operator is then defined as:

$$\tau_{\mathcal{M}=<\mathcal{A},\mathfrak{f}>}(S,\mathcal{B}) = \{<s,b> \epsilon\ \mathcal{P}(S) \times \mathcal{B} : \mathfrak{f}(s) = <s,b>\}$$

This operator relies on the use of Classification models in a similar way as external functions are used in some relational algebra extensions like the computation of aggregates (Klug, 1982) and the mapper operator (Carreira et al., 2007).

Example 4.2.1:

Consider a Classification model $\mathcal{M}$ that classifies segments of texts about seminars according to the following classes: speaker (represented by the string "speaker"), location (represented by the string "location"), start time (represented by the string "stime") and end time (represented by the string "etime").

Let S be the character sequence that follows:

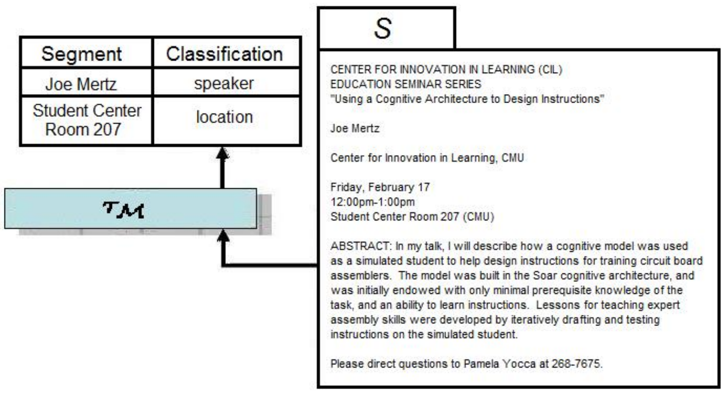
CENTER FOR INNOVATION IN LEARNING (CIL)
EDUCATION SEMINAR SERIES
”Using a Cognitive Architecture to Design Instructions“

Joe Mertz
Center for Innovation in Learning, CMU

Friday, February 17
12:00pm-1:00pm
Student Center Room 207 (CMU)
ABSTRACT: In my talk, I will describe how a cognitive model was used as a simulated student to help design instructions for training circuit board assemblers. The model was built in the Soar cognitive architecture, and was initially endowed with only minimal prerequisite knowledge of the task, and an ability to learn instructions. Lessons for teaching expert assembly skills were developed by iteratively drafting and testing instructions on the simulated student.

Please direct questions to Pamela Yocca at 268-7675.

Figure 4.2 illustrates the application of the Classification operator to the character sequence S using model $\mathcal{M}$ to classify segments as speaker or location.



Segment	Classification
Joe Mertz	speaker
Student Center Room 207	location

CENTER FOR INNOVATION IN LEARNING (CIL)
EDUCATION SEMINAR SERIES
"Using a Cognitive Architecture to Design Instructions"

Joe Mertz

Center for Innovation in Learning, CMU

Friday, February 17
12:00pm-1:00pm
Student Center Room 207 (CMU)

ABSTRACT: In my talk, I will describe how a cognitive model was used as a simulated student to help design instructions for training circuit board assemblers. The model was built in the Soar cognitive architecture, and was initially endowed with only minimal prerequisite knowledge of the task, and an ability to learn instructions. Lessons for teaching expert assembly skills were developed by iteratively drafting and testing instructions on the simulated student.

Please direct questions to Pamela Yocca at 268-7675.

Figure 4.2: Classification of text segments as speaker or location using the Classification operator

4.3 The CEE engine Java API

As referred in Section 4.1, the CEE engine for each operator is responsible for the Creation, Execution and Evaluation of extraction models. In this section, we present the instantiation of the CEE engine for the Classification operator that was developed as a proof-of-concept. Figure 4.3 shows the UML class diagram of the CEE engine for the Classification operator.

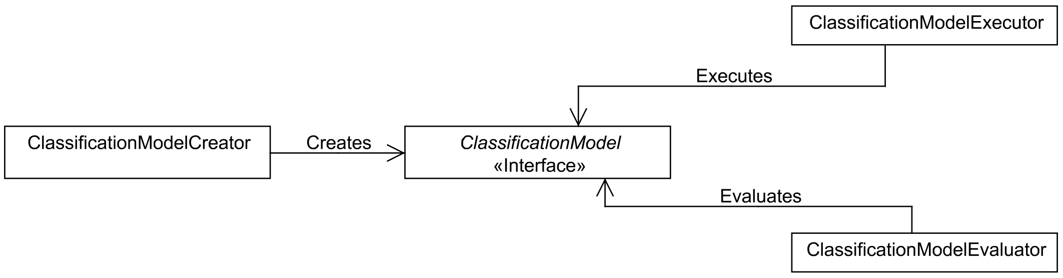

ClassificationModelExecutor
Executes
ClassificationModelCreator
Creates
ClassificationModel
«Interface»
Evaluates
ClassificationModelEvaluator

Figure 4.3: UML class diagram for the Classification operator CEE engine

The functionalities of the CEE Engine for the Classification operator are the creation, the execution and the evaluation of a Classification model. Therefore, the concept of Classification model is the most important in the Classification operator CEE Engine. Classification models are represented by instances of classes that implement the interface *ClassificationModel*.

The creation of Classification models is achieved through objects of the *ClassificationModelCreator* class. The *ClassificationModelCreator* class offer methods that produce Classification models based on several classification techniques. After the creation of a Classification model, it can be executed or evaluated with a given testing corpus. In the CEE Engine of the Classification operator, the execution of these models is performed by instances of the *ClassificationExecutor* class. In addition, the *ClassificationExecutor* class offers methods to print or return the annotated text.

The *ClassificationEvaluator* class returns the effectiveness of a Classification model, through the recall, precision and F-measure measures (see Section 2.4 for more detail about these measures).

4.3.1 The ClassificationModel interface

Classification models are the main component of the Classification operator CEE Engine. In fact, these models store information about how text segments are classified. The Classification models are represented in the CEE Engine by objects of classes that implement the *ClassificationModel* interface. Figure 4.4 shows the UML class diagram of this interface.

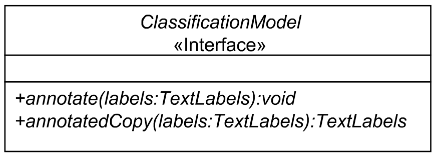

ClassificationModel
«Interface»
+annotate(labels:TextLabels):void
+annotatedCopy(labels:TextLabels):TextLabels

Figure 4.4: UML class diagram of the *ClassificationModel* interface

This interface forces the implementation of two methods: *annotate* and *annotateCopy*. The *annotate* method receives as input a text in the format internally used by the CEE engine (as determined by the *TextLabels* class) and classifies its segments. The *annotateCopy* method is a non-destructive version of the *annotate* method. It receives a text but does not change it. Instead, it returns an annotated copy of the original text.

There is a class that implements the *ClassificationModel* interface for each classification technique available in E-txt2db. Figure 4.5 presents the UML class diagram showing the classes that implement the *ClassificationModel* interface. Each of these classes correspond to an implementation of a classification technique described in Section 2.3.2. The *DictionaryClassificationModel* implements a dictionary-based approach. The *RegexClassificationModel* uses regular expressions to perform the Classification task. The *HMMClassificationModel* corresponds to an Hidden Markov Model. The *MEMMClassificationModel* implements a Maximum Entropy Markov Model. The *CRFClassificationModel* uses a Conditional Random Fields model. The *SVMClassificationModel* uses a Support Vector Machine model.

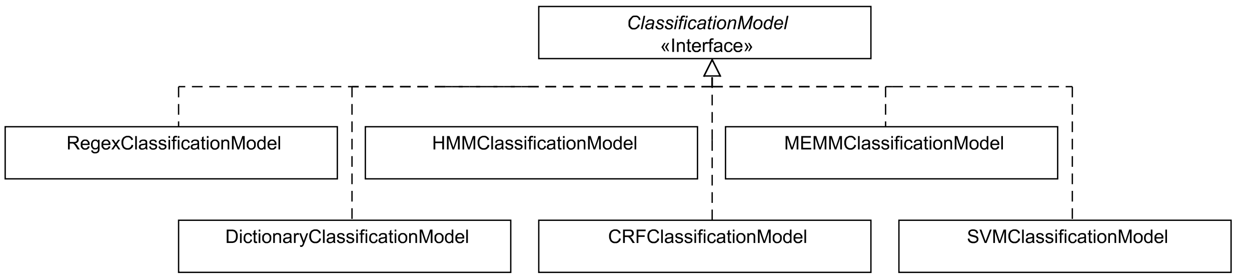


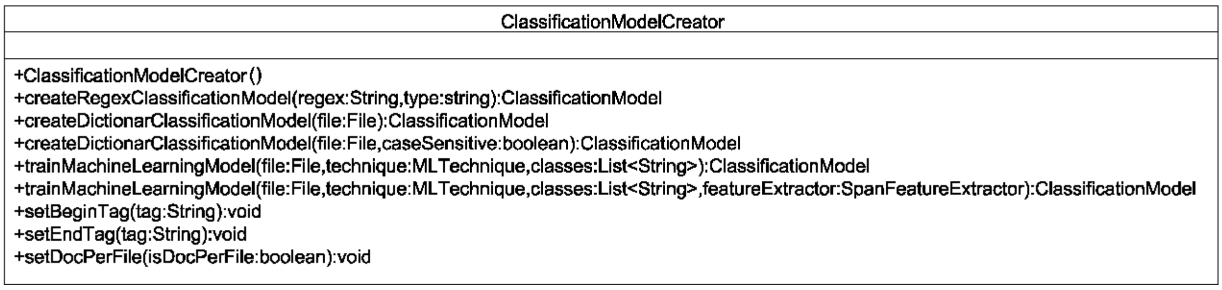
Figure 4.5: UML class diagram representing the classes that implement the *ClassificationModel* interface

The implementation of these techniques was not performed from scratch. With exception of some modifications applied to each technique to match the architecture of the CEE Engine, the Dictionary based model, the Regular Expressions model and the Hidden Markov model are the implementations offered by *Lingpipe* (see Section 3.1.2). The Maximum Entropy Markov Model, the Conditional Random Fields model and the Support Vector Machine model were adapted from the *Minorthird* implementation (see Section 3.1.1).

4.3.2 The ClassificationModelCreator class

The *ClassificationModelCreator* class is responsible for the creation of Classification models. It offers methods to create *ClassificationModel* objects for several classification techniques. Figure 4.6 presents the UML representation of this class.

The *ClassificationModelCreator* class can create Classification models that use rule-based techniques or machine learning techniques. There are two methods that return rule-based Classification models: *createRegexClassificationModel* and *createDictionaryClassificationModel*. The *createRegexClassificationModel* method returns a *ClassificationModel* object that uses regular expressions to perform a Classification task. This method receives two arguments as input: a regular expression and the class that must be assigned to the segments that mach the regular expression.



```
ClassificationModelCreator

+ClassificationModelCreator()
+createRegexClassificationModel(regex:String,type:string):ClassificationModel
+createDictionarClassificationModel(file:File):ClassificationModel
+createDictionarClassificationModel(file:File,caseSensitive:boolean):ClassificationModel
+trainMachineLearningModel(file:File,technique:MLTechnique,classes:List<String>):ClassificationModel
+trainMachineLearningModel(file:File,technique:MLTechnique,classes:List<String>,featureExtractor:SpanFeatureExtractor):ClassificationModel
+setBeginTag(tag:String):void
+setEndTag(tag:String):void
+setDocPerFile(isDocPerFile:boolean):void
```

Figure 4.6: UML class diagram representing the detailed information of the *ClassificationModelCreator* class

Example 4.3.1:

Consider the creation of a *ClassificationModel* to extract text segments that correspond to time. The corresponding Java code is the following:

```
String timeRegex = ``[0-9][0-9]:[0-9][0-9]((pm)|(am))?'';
ClassificationModelCreator creator = new ClassificationModelCreator();
ClassificationModel m = creator.createRegexClassificationModel(timeRegex,``time'');
```

The overloaded method *createDictionaryClassificationModel* returns a *ClassificationModel* for a dictionary-based classification technique. One of the versions of this method receives a Java *File* as input. In this file, each word of the dictionary is in a line with a XML tag indicating the corresponding class.

Example 4.3.2:

Consider the following dictionary.

```
<Month>January</Month>
<Month>February</Month>
<Month>March</Month>
<Month>April</Month>
<Month>May</Month>
<Month>June</Month>
(...)
```

If this dictionary is stored in a file called "monthsDictionary.txt", the Java code that creates a *ClassificationModel* to extract texts segments that correspond to months is the following:

```
File monthsDictionary = new File(``monthsDictionary.txt'');
ClassificationModelCreator creator = new ClassificationModelCreator();
ClassificationModel m = creator.createRegexClassificationModel(monthsDictionary);
```

By default, the returned *ClassificationModel* is case sensitive. If a user wants to create a case insensitive *ClassificationModel*, the second version of the *createDictionaryClassificationModel* method must be used. Other than the dictionary file, this version of the method receives a boolean value as input indicating whether the *ClassificationModel* is case sensitive.

Example 4.3.3:

The Java code introduced in Example 4.3.2 must be modified as follows to create a case insensitive dictionary

```
File monthsDictionary = new File(``monthsDictionary.txt'');
ClassificationModelCreator creator = new ClassificationModelCreator();
ClassificationModel m = creator.createRegexClassificationModel(monthsDictionary,false);
```

The overloaded method *trainMachineLearningModel* allows the user to create *ClassificationModel* objects that use machine learning techniques. One of the versions of this method receives as input a Java *File* corresponding to the training corpus, an enumerate value indicating the machine learning technique used and the list of classes that should be assigned to the text segments using the model.

By default, the format used for the training corpus is a text file with XML tags classifying text segments as represented in Example 4.3.4.

Example 4.3.4:

Consider the following document:

CENTER FOR INNOVATION IN LEARNING (CIL)
EDUCATION SEMINAR SERIES
"Using a Cognitive Architecture to Design Instructions"

<**speaker**>Joe Mertz</**speaker**>
Center for Innovation in Learning, CMU

<**date**>Friday, February 17</**date**>
12:00pm-1:00pm
<**location**>Student Center Room 207</**location**> (CMU)
ABSTRACT: In my talk, I will describe how a cognitive model was used as a simulated student to help design instructions for training circuit board assemblers. The model was built in the Soar cognitive architecture, and was initially endowed with only minimal prerequisite knowledge of the task, and an ability to learn instructions. Lessons for teaching expert assembly skills were developed by iteratively drafting and testing instructions on the simulated student.

Please direct questions to Pamela Yocca at 268-7675.

In this document, “Friday, February 17” is classified as a date, “Joe Mertz” is classified as a speaker and “Student Center Room 207” is classified as a location.

In order to support the choice of the machine learning technique used, the *ClassificationModelCreator* class contains an enumerate, called *MLTechnique*, that associates an identifier to each technique, as represented next:

- *MLTechnique.CRF*: Conditional Random Fields Model technique.
- *MLTechnique.HMM*: Hidden Markov Model technique.
- *MLTechnique.MEMM*: Maximum Entropy Markov Model technique.
- *MLTechnique.SVM*: Support Vector Machines Model technique.

Example 4.3.5:

Consider that a user wants to create a *ClassificationModel* that uses a CRF technique and train it with the file “train.txt” to extract locations and speakers from a corpus. The code to create this Classification model is:

```
File trainingCorpus = new File(``train.txt'');
List<String> listTypes = new ArrayList<String>();
listTypes.add(``speaker'');
listTypes.add(``location'');
ClassificationModelCreator creator = new ClassificationModelCreator();
ClassificationModel m = creator.trainModel(trainingCorpus,MLTechnique.CRF,listTypes);
```

In order to take full advantage of the potentialities of MEMM, CRF and SVM, it is common to use additional knowledge about the segments of the corpus when training and executing the Classification models. Each additional piece of knowledge included in a classification process is called a *feature*. By default, when a *ClassificationModelCreator* creates a *ClassificationModel* with one of these techniques it automatically extracts some features by default. Examples of these features are the size of the segment and an automatically generated regular expression that matches the segment. The problem of using these features is that they are not adapted to the corpus. For this reason, there is a second version of the *trainMachineLearningModel* method that receives the same parameters as the first one plus an object to extract features (which is an instance of the *SpanFeatureExtractor* interface supplied by *Minorthird*).

The Classification operator CEE Engine contains a class called *EditableTokenFE* that is easily customizable, and makes it possible to select the features to be used. Figure 4.7 shows the UML class diagram of the *EditableTokenFE* class.

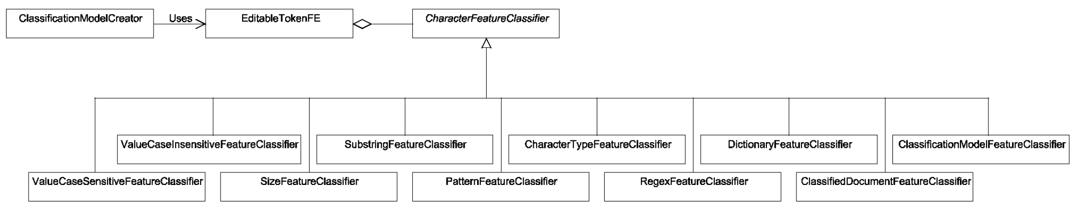

ClassificationModelCreator
Uses
EditableTokenFE
CharacterFeatureClassifier
ValueCaseInsensitiveFeatureClassifier
SubstringFeatureClassifier
CharacterTypeFeatureClassifier
DictionaryFeatureClassifier
ClassificationModelFeatureClassifier
ValueCaseSensitiveFeatureClassifier
SizeFeatureClassifier
PatternFeatureClassifier
RegexFeatureClassifier
ClassifiedDocumentFeatureClassifier

Figure 4.7: UML class diagram of the *EditableTokenFE* class

The type of features of an *EditableTokenFE* is customized by providing a list of *CharacterFeatureClassifier* objects in the constructor. There are several classes that implement the interface *CharacterFeatureClassifier*. Each one is able to extract one type of feature. Next, we describe each of these classes:

- *ValueCaseSensitiveFeatureClassifier*: returns features with information about the case sensitive value of the segment.
- *ValueCaseInsensitiveFeatureClassifier*: returns features with information about the case insensitive value of the segment. They are useful if the user wants segments with different cases to be considered equal.
- *SizeFeatureClassifier*: returns features with the number of characters of the segment. They are useful if the size of the segment indicates the class it belongs to (e.g., if a time value is always formatted as "00:00", it may be useful that the classification process considers segments with five characters to have high probability of being a time value).
- *PatternFeatureClassifier*: returns features with information about an automatically generated regular expression the segment respects. They are useful if the segments of a given class usually respect a given pattern (e.g., to classify segments as person names, it is useful to have information about which segments match the pattern "[A-Z][a-z]+").
- *CharacterTypeFeatureClassifier*: returns features with information about the type of characters that constitute the segment. They are useful if the segments of a given class usually have the same type of characters (e.g., if a time is always formatted as "00:00", the classification process should assign a high probability of being a time value to segments that start with two digits, followed by a colon, and two digits)
- *SubstringFeatureClassifier*: returns features with information about a substring of the segment. They are useful if small parts of the segment are an indicator of the class the segment belongs to (e.g., in a part-of-speech-tagging task it is useful to consider information about the last three letters of the segment, in order to find words that end with "ing" that usually correspond to verbs in the

gerund). To create a *SubstringFeatureClassifier*, a user must provide two integer numbers: the first one indicates the index of the initial character of the substring and the last one indicates the index of the last character of the substring.

- *RegexFeatureClassifier*: returns features with information of whether the segment matches a given regular expression. Analogously to the *PatternFeatureClassifier*, they are useful if the segments of a given class usually respect a given pattern. The difference is that, in this case, the regular expression is not automatically generated. Instead, the user manually provides the regular expression, which is a better solution in case of complex regular expressions that cannot be automatically generated.

- *DictionaryFeatureClassifier*: returns features with information of whether the segment is present in a dictionary. It is useful if the user can provide a dictionary with a list of elements from the class (even if it is not an exhaustive dictionary). To create a *DictionaryFeatureClassifier*, a user must provide a string with the path to the dictionary.

- *ClassificationModelFeatureClassifier*: returns features with information from a previously trained Classification model. It is useful if there is a relationship between the classification performed by the previously trained model and the one being created. For example, if there is a previously trained model that classifies segments as person names, it can be used as a feature to classify the segments as the speaker of a seminar, because the speaker is a person. To create a *ClassificationModelFeatureClassifier*, the user must provide a *ClassificationModel* object and a string representing the name of the Classification model.

- *ClassifiedDocumentFeatureClassifier*: returns features with classification manually introduced in the corpus. It is used like *ClassificationModelFeatureClassifier* but it does not need a previously classified model. Instead, a user must manually provide the classification of some segments directly in the corpus with XML tags delimiting the classified segments. To create a *ClassifiedDocumentFeatureClassifier*, the user must provide a string representing the class he wants to get from the corpus as additional information.

Example 4.3.6:

Consider again the creation of a *ClassificationModel* that uses a CRF technique, to be trained with the file "train.txt" for extracting locations and speakers from a corpus. Features should be used to take full advantage of the CRF capabilities. The user provides a previously trained *ClassificationModel* that classifies text segments as "person" (consider it was previously assigned to the variable *personClassModel*) and a dictionary called "seminarsPlaces.txt" with a list of places where seminars usually take place. The user decides to use these resources as features of the CRF model along with the case sensitive value of the word and an automatically generated pattern. The code to train this model is as follows:

```
File trainingCorpus = new File(``train.txt'');
List<String> listTypes = new ArrayList<String>();
listTypes.add(``speaker'');
listTypes.add(``location'');
List<CharacterFeatureClassifier> features = new ArrayList<CharacterFeatureClassifier>();
features.add(new ValueCaseSensitiveFeatureClassifier());
features.add(new PatternFeatureClassifier());
features.add(new DictionaryFeatureClassifier(''seminarsPlaces.txt''));
features.add(new ClassificationModelFeatureClassifier(personClassModel,''personModel''));
EditableTokenFE tokenFE = new EditableTokenFE(features);
ClassificationModelCreator creator = new ClassificationModelCreator();
ClassificationModel m = creator.trainModel(trainingCorpus, MLTechnique.CRF, listTypes,
tokenFE);
```

The *ClassificationModelCreator* class also offers some methods to turn the use of machine learning techniques more flexible. The methods *setBeginTag* and *setEndTag* allow a user to change the format of the training corpus tags by changing the beginning and the ending symbols of the tags respectively.

Example 4.3.7:

Consider that instead of using XML tags, the user wants to use a training corpus tagged as follows:

```
CENTER FOR INNOVATION IN LEARNING (CIL)
EDUCATION SEMINAR SERIES
"Using a Cognitive Architecture to Design Instructions"

[speaker]Joe Mertz[/speaker]
Center for Innovation in Learning, CMU

[date]Friday, February 17[/date]
(...)
```

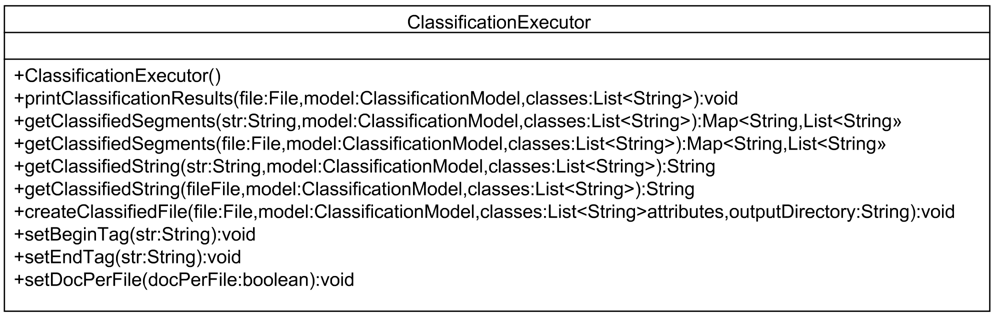
To load a training corpus like this, the user just needs to call the *setBeginTag* and *setEndTag* methods immediately after creating the *ClassificationModelCreator* object:

```
ClassificationModelCreator creator = new ClassificationModelCreator();
creator.setBeginTag(''['');
creator.setEndTag('']'');
```

Finally, the *setDocPerFile* method also changes the way the training corpus is loaded. This method is used to indicate the format of the training corpus: *(i)* one document for each file, or *(ii)* one document for each line. By default, the *ClassificationModelCreator* considers that the training corpus format is one document for each file. If a user wants to change it, he just needs to call the *setDocPerFile* method giving the boolean value *false* as parameter.

4.3.3 The ClassificationExecutor class

The class responsible for executing Classification models in the Classification operator CEE engine is the *ClassificationExecutor*. This class offers methods that apply a *ClassificationModel* to a given text and print or return the results. Figure 4.8 presents a detailed representation of this class in UML.



```
ClassificationExecutor

+ClassificationExecutor()
+printClassificationResults(file:File,model:ClassificationModel,classes:List<String>):void
+getClassifiedSegments(str:String,model:ClassificationModel,classes:List<String>):Map<String,List<String»
+getClassifiedSegments(file:File,model:ClassificationModel,classes:List<String>):Map<String,List<String»
+getClassifiedString(str:String,model:ClassificationModel,classes:List<String>):String
+getClassifiedString(fileFile,model:ClassificationModel,classes:List<String>):String
+createClassifiedFile(file:File,model:ClassificationModel,classes:List<String>attributes,outputDirectory:String):void
+setBeginTag(str:String):void
+setEndTag(str:String):void
+setDocPerFile(docPerFile:boolean):void
```

Figure 4.8: UML class diagram of the *ClassificationExecutor* class

If the user only wants to print the results of a Classification task, the method *printClassificationResults* should be used. This method receives as parameters a File containing the corpus to be classified, the Classification model to be used and a list of classes to be used in the classification process. This method only prints the classified segments for each class present in the list of classes. An example of the result of this method is as follows:

```
Segments classified as 'date':
Friday, February 17
Segments classified as 'speaker':
Joe Mertz
Segments classified as 'location':
Student Center Room 207
```

Example 4.3.8:

Consider the application of a previously trained *ClassificationModel* called *model* that classifies text segments as *speaker* and *location* that belong to the "newSeminar2009.txt" file. In order to print the results as described above we use the following code:

```
File testingFile = new File(``newSeminar2009.txt'');
List<String> listTypes = new ArrayList<String>();
listTypes.add(``speaker'');
listTypes.add(``location'');
ClassificationExecutor exec = new ClassificationExecutor();
exec.printClassificationResults(testingFile, model, listTypes);
```

If a user wants to retrieve the classified segments instead of printing them, one of the versions of the overloaded method *getClassifiedSegments* should be used. The difference between the two versions of this method is that one receives the corpus as a String and another receives it as a File. Like the *printClassificationResults* method, this method receives as input a Classification model and a list of classes to be used in the classification process. The result is returned as a Map structure in which the keys are the classes the user wants to classify and the stored values are lists containing the segments assigned to the class.

Example 4.3.9:

To retrieve the segments of the file "newSeminar2009.txt" using this method the following code should be used:

```
File testingFile = new File(``newSeminar2009.txt);"
List<String> listTypes = new ArrayList<String>();
listTypes.add(``speaker'');
listTypes.add(``location'');
ClassificationExecutor exec = new ClassificationExecutor();
Map<String,List<String>> result =exec.getClassifiedSegments(testingFile, model,
listTypes);
```

The overloaded method *getClassifiedSegments* returns the classified text as a whole instead of only the classified segments. This method returns a String with the content of the text in which the segments are classified using XML tags like the training corpus presented in Example 4.3.4. The difference between the two versions of the method *getClassifiedString* is that one receives the corpus as a String and the other receives it as a File. This method accepts as input the Classification model and a list of classes to be used in the classification process.

Example 4.3.10:

To get the string containing a classified version of "newSeminar2009.txt" using this method, the following code must be used:

```
File testingFile = new File(``newSeminar2009.txt'');
List<String> listTypes = new ArrayList<String>();
listTypes.add(``speaker'');
listTypes.add(``location'');
ClassificationExecutor exec = new ClassificationExecutor();
String result = exec.getClassifiedString(testingFile, model, listTypes);
```

The *createClassifiedFile* method creates and stores a classified version of a text in a given folder. This method accepts as input a File containing the corpus to be classified, the Classification model, a list of classes to be used in the classification process, and a String indicating the path to the directory where the classified version of the corpus must be created. The classified corpus is stored in the same format as the training corpus presented in Example 4.3.4, which is text file with XML tags classifying the segments.

Example 4.3.11:

To create a classified version of "newSeminar2009.txt" using this method, we use the following code:

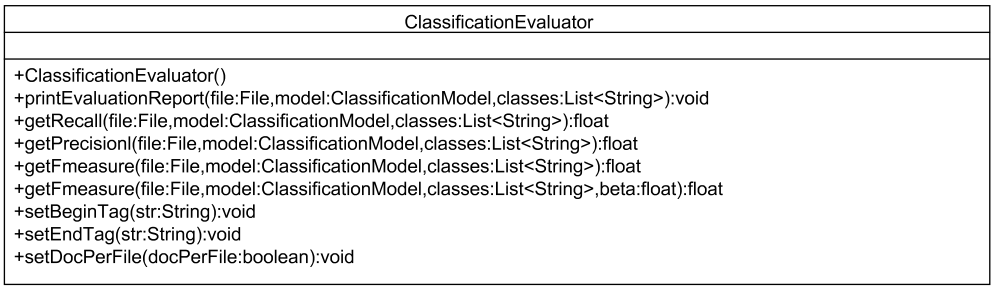
```
File testingFile = new File(``newSeminar2009.txt'');
List<String> listTypes = new ArrayList<String>();
listTypes.add(``speaker'');
listTypes.add(``location'');
String outputFile = ``results/''
ClassificationExecutor exec = new ClassificationExecutor();
exec.createClassifiedFile(testingFile, model, listTypes, outputFile);
```

Similarly to the *ClassificationModelCreator* class, the *ClassificationExecutor* also contains methods that turn the loading and creation of corpus more flexible. The semantics of the *setBeginTag*, *setEndTag* and *setDocPerFile* methods is the same as the methods with the same name in the *ClassificationModelCreator* class.

4.3.4 The ClassificationEvaluator class

The Classification operator CEE engine contains the *ClassificationEvaluator* class to compute the recall, precision and F-measure of the results obtained by applying a previously created Classification model to a given testing corpus. Figure 4.9 presents a detailed representation of this class in UML.

The *printEvaluationReport* method prints a report of how the Classification model performs for a given testing corpus. This method receives as input a file containing the testing corpus, the Classification



```
ClassificationEvaluator

+ClassificationEvaluator()
+printEvaluationReport(file:File,model:ClassificationModel,classes:List<String>):void
+getRecall(file:File,model:ClassificationModel,classes:List<String>):float
+getPrecisionl(file:File,model:ClassificationModel,classes:List<String>):float
+getFmeasure(file:File,model:ClassificationModel,classes:List<String>):float
+getFmeasure(file:File,model:ClassificationModel,classes:List<String>,beta:float):float
+setBeginTag(str:String):void
+setEndTag(str:String):void
+setDocPerFile(docPerFile:boolean):void
```

Figure 4.9: UML class diagram of the *ClassificationEvaluator* class

model and a list of classes to be used in the classification process. The testing corpus must be a classified corpus with the same format as the training corpus presented in Example 4.3.4.

The result of this method prints the values of recall, precision, F-measure with $\beta = 1$, the number of correct results, expected results and the total number of classified segments used in the computation of recall and precision.

Example 4.3.12:

Consider the previously trained *ClassificationModel*, named *model*, that classifies text segments as *speaker* and *location*, must be evaluated against the manually classified text "classifiedSeminars.txt". In order to print the results, as described above, the following code should be used:

```
File testingFile = new File(``classifiedSeminars.txt'');
List<String> listTypes = new ArrayList<String>();
listTypes.add(``speaker'');
listTypes.add(``location'');
ClassificationEvaluator eval = new ClassificationEvaluator();
eval.printEvaluationReport(testingFile, model, listTypes);
```

An example of the result of this program is as follows:

```
Recall report:
Correct results: 10
Expected results: 20
Recall: 0.5
Precision report:
Correct results: 10
Total classified: 16
Precision: 0.625
F-measure(1): 0.55556
```

The *ClassificationEvaluator* class also provides methods to retrieve the recall, precision and F-measure values. These methods are *getRecall*, *getPrecision* and *getFmeasure*. All these methods return a float value representing respectively recall, precision and F-measure. The input of the *getRecall* and *getPrecision* methods is the same as the input of *printEvaluationReport*. Since *getFmeasure* is an overloaded method, there is a version with the same input as *printEvaluationReport* and returns the F-measure for $\beta = 1$ while the other version receives an extra parameter that corresponds to the β value.

Example 4.3.13:

The following code should be used to get the recall of the results obtained when applying a model to "classifiedSeminars.txt":

```
File testingFile = new File(``classifiedSeminars.txt'');
List<String> listTypes = new ArrayList<String>();
listTypes.add(``speaker'');
listTypes.add(``location'');
ClassificationEvaluator eval = new ClassificationEvaluator();
float recall = eval.getRecall(testingFile, model, listTypes);
```

Similarly to the *ClassificationModelCreator* class, the *ClassificationEvaluator* also contains methods that turn the loading of corpus more flexible. The semantics of the methods *setBeginTag*, *setEndTag* and *setDocPerFile* is the same as the methods with the same name in the *ClassificationModelCreator* class.

4.4 SQL-like syntax for the Classification operator

We developed our own declarative language to specify Classification tasks. We decided to propose a language similar to SQL for two reasons: *(i)* many people are used to the relational technology, and thus to write queries in SQL. For all these people, the difficulty of learning our language will be low; *(ii)* in the future, we plan to integrate the Classification operator in a Relational Database Management System thus allowing a user to perform information extraction over text databases. This integration will be easy if the language for Information Extraction is an extension of SQL.

The E-txt2db classification language offers three types of queries: *(i)* creation queries, that are responsible for the creation of Classification models; *(ii)* execution queries, that perform a Classification operator execution for a given corpus and using for a given model. *(iii)* evaluation queries, that evaluate the results obtained by a Classification model. In this section, we describe the three types of queries.

To present the syntax of these queries, we use a graph-based approach. In the graph, the rectangular nodes represent keywords and the nodes with rounded tips represent blocks parameterized by the user.

4.4.1 Creation queries

The creation queries implement the functionalities of the CEE Engine *ClassificationModelCreator*. Figure 4.10 presents the generic syntax of a creation query.

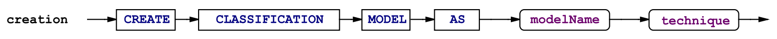


Figure 4.10: Syntax of a creation query

modelName corresponds to the name to be assigned to the Classification model. This name becomes the identifier of the Classification model after the query is executed. It must contain one or more characters from all the letters, digits, and symbols ".", ":", "-" and "_". The name must start by a letter or the symbol "_".

The *technique* block indicates the classification technique used by the Classification model. The syntax for this block is different for each classification technique, as represented in Figure 4.11.

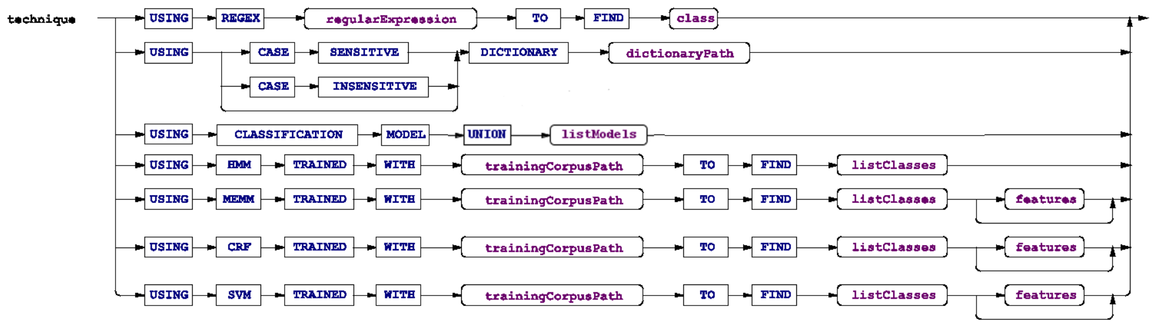


Figure 4.11: Syntax of the *technique* block

The first path of the graph represents the syntax to create a Classification model based on regular expressions. In this path, *regularExpression* is a string delimited by “”” or “”’ that indicates the regular expression the Classification model uses. The end of this path, the *class* variable, indicates the class that must be assigned to the segments that match the regular expression. The name of the class follows the same lexical rules as the name of the Classification models.

The second path of the graph corresponds to the syntax available to create a dictionary-based Classification model. In this path, *dictionaryPath* is a String indicating the path to the dictionary file. Note that, to create a dictionary, a user can optionally indicate if he wants a case sensitive or case insensitive dictionary. By default, the dictionary is case sensitive.

The third path of the graph allows a user to create a new Classification model as a union of previously created Classification models. In this path, *listModels* is a comma separated list of previously created Classification models. Applying the resulting model of this query to an input text is equivalent to consecutively applying all the models in *listModels* to the text.

The remaining paths correspond to the creation of Classification models based on machine learning techniques. In these paths, *trainingCorpusPath* is a string indicating the path to the training corpus. The *listClasses* is a list of one ore more comma separated classes that the Classification model must learn to identify using the training corpus. Alternatively, it can be the symbol "*", if the user wants the model to learn to how identify all the classes that are present in the training corpus. If the machine learning technique used is MEMM, CRF or SVM, the user can, optionally, customize the features used in the classification process. This customization is performed through the *features* block. Figure 4.12 presents the syntax of the *features* block.

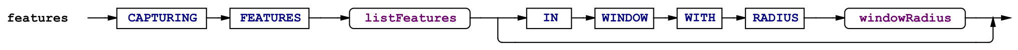


Figure 4.12: Syntax of the *features* block

The *features* block is responsible for the configuration of the feature extraction. It contains an optional block to indicate the feature extraction window using the *windowRadius* block. When extracting features from a segment, the feature extraction window is the number of segments in its neighborhood that will contribute with their features to the features of the segment. The block *windowRadius* must be a positive integer value.

The main element of the *features* block is *listFeatures*. This block is a list of comma separated features. The syntax of each feature depends on its type. Figure 4.13 presents the syntax of the *feature* block.

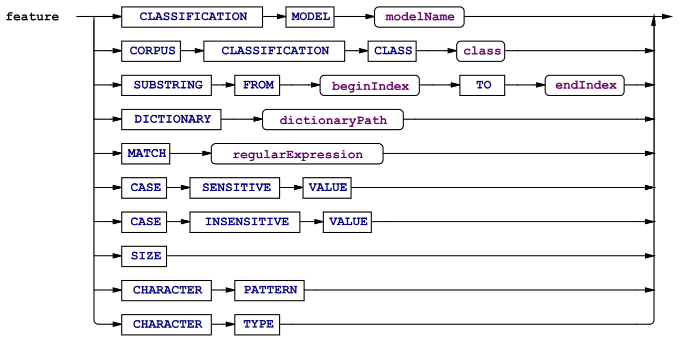


Figure 4.13: Syntax of the *feature* block

Each of the paths of the graph in Figure 4.13 corresponds to one type of feature. Each of these types is related to a CEE Engine Java class that extends the abstract class *CharacterFeatureClassifier*, as described in Section 4.3.2. Table 4.1 represents the relationship between each path of the graph and the corresponding CEE Engine Java classes.

In the `Classification model modelName` path, the `modelName` corresponds to the name of a previously created Classification model. The block `class` from the `CORPUS CLASSIFICATION CLASS class` corresponds to the name of a class present in the corpus. The `beginIndex` and

Table 4.1: Relationship between paths from Figure 4.13 and classes from the CEE Engine

SQL-like syntax for features	**CEE Engine class**
`Classification model modelName`	*ClassificationModelFeatureClassifier*
`CORPUS CLASSIFICATION CLASS class`	*ClassifiedDocumentFeatureClassifier*
`SUBSTRING FROM beginIndex TO endIndex`	*SubstringFeatureClassifier*
`DICTIONARY dictionaryPath`	*DictionaryFeatureClassifier*
`MATCH regularExpression`	*RegexFeatureClassifier*
`CASE SENSITIVE VALUE`	*ValueCaseSensitiveFeatureClassifier*
`CASE INSENSITIVE VALUE`	*ValueCaseInsensitiveFeatureClassifier*
`SIZE`	*SizeFeatureClassifier*
`CHARACTER PATTERN`	*PatternFeatureClassifier*
`CHARACTER TYPE`	*CharacterTypeFeatureClassifier*

`endIndex` from the path `SUBSTRING FROM beginIndex TO endIndex` are integers corresponding to the boundaries of the substring. In the `DICTIONARY dictionaryPath` path, the `dictionaryPath` is a String indicating the path to the dictionary file. The block `regularExpression` from the path `MATCH regularExpression` is a String indicating a regular expression.

Example 4.4.1:

Recall Example 4.3.6, where the goal is to create a Classification model that uses a CRF technique with features and train it with the file "train.txt" to extract locations and speakers from a corpus. We consider that the Classification model created is called *SpeakerLocationCRFModel*. The syntax used to create the Classification model is:

```
CREATE Classification model AS SpeakerLocationCRFModel
USING CRF TRAINED WITH 'train.txt'
TO FIND speaker, location
CAPTURING FEATURES CASE SENSITIVE VALUE,
                   CHARACTER PATTERN,
                   DICTIONARY 'seminarsPlaces.txt',
                   Classification model personModel
```

4.4.2 Execution queries

The execution queries are used to perform a Classification task using a previously created Classification model. These queries applied to a given text produce a classified copy of the text that is composed by a set of classified segments marked with XML tags. The functionality of these queries correspond to the invocation of the method *getClassifiedString* of the CEE Engine *ClassificationExecutor* class (described in Section 4.3.3). Figure 4.14 presents the generic syntax of an execution query.

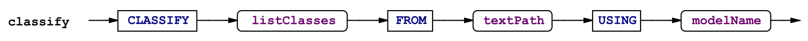


Figure 4.14: Syntax of an execution query

In this syntax, the *listClasses* block corresponds to a list of comma separated classes. Similarly to the creation queries, the name of a class follows the same lexical rules as the Classification model names. The *textPath* block is a String indicating the path to the file containing the corpus to which the Classification model is applied. Finally, the *modelName* block corresponds to the name of a previously trained Classification model.

Example 4.4.2:

Consider that we want to use the Classification model created in Example 4.4.1 to perform a Classification task in the text contained in the file "text.txt". The query to perform this task is:

```
CLASSIFY speaker, location
FROM 'text.txt'
USING SpeakerLocationCRFModel
```

4.4.3 Evaluation queries

The evaluation queries determine the effectiveness of a Classification model by measuring the recall, precision and the F-measure with a given testing corpus. Figure 4.15 presents the syntax of an evaluation query in the syntax we propose.

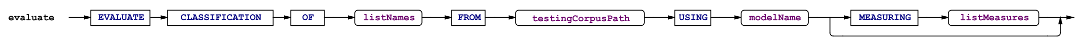


Figure 4.15: Syntax of an evaluation query

In this syntax, there are three mandatory blocks: *listClasses*, *testingCorpusPath* and *modelName*. The *listClasses* block corresponds to a list of comma separated classes. The name of the classes follows the same syntactic rules as the Classification model names. The *testingCorpusPath* block is a string indicating the path to the testing file. Finally, the *modelName* block corresponds to the name of a previously trained Classification model. If a query is executed with the mandatory blocks only, the result of the evaluation query is a text with three lines. The first line shows the value of recall, the second line presents the value of precision and the third line displays the value of the F-measure with $\beta = 1$.

The optional part of the query specifies which measures the user wants to obtain with the evaluation. The block *listMeasures* corresponds to a list of comma separated measures. A measure is represented by the *measure* block and its syntax is presented in Figure 4.16.

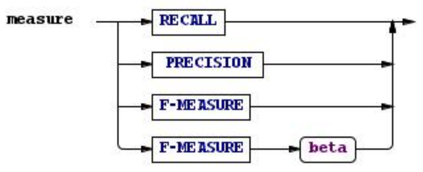


Figure 4.16: Syntax of the *measure* block

For each *measure* present in the query, there is a resulting line in its result. If the user uses the F-measure but does not specify the value of β, it is considered that $\beta = 1$.

Example 4.4.3:

Consider that we want to evaluate the Classification model created in Example 4.4.1 using the classified corpus "test.txt" by measuring the recall, precision and F-measure with $\beta = 2$. The query to perform this task is:

```
EVALUATE CLASSIFICATION OF speaker, location
FROM 'test.txt'
USING SpeakerLocationCRFModel
MEASURING recall, precision, f-measure 2
```

5 Evaluation

The framework for Information Extraction proposed in this thesis, called E-txt2db, encloses a declarative set of operators that correspond to the semantics of Information Extraction tasks. This approach joins the advantages of the Information Extraction frameworks with the advantages of the declarative languages for Information Extraction. Therefore, the E-txt2db framework makes it possible: *(i)* to easily specify an Information Extraction program; *(ii)* to easily maintain the program to better fit the objective of a particular application; *(iii)* to locally refine an Information Extraction program since the module responsible for each task is completely independent from the others; *(iv)* to accept different types of input data; *(v)* to customize an Information Extraction activity according to an application needs, by selecting, composing and reordering some operators; and *(vi)* to automatically optimize the Information Extraction process, because the operators are declarative. In order to demonstrate these advantages of E-txt2db mentioned above, we performed the set of experiments that are described in this chapter.

As referred in Chapter 4, we only developed a proof-of-concept for the Classification operator, so all the experiments performed are focused in the Classification task. This decision imposed two limitation to the evaluation of the E-txt2db framework. First, it is not possible to evaluate the customization of the Information Extraction process, since it is only possible to select and compose Classification models. Second, even though we tested the modification of the execution algorithm without changing the specification of the program, this change was performed manually and thus no cost model or heuristics were considered.

This chapter starts by describing an experience that aims at proofing that it is easier to specify and maintain Classification programs with E-txt2db than with Minorthird (Section 5.1). In this experience we could not compare E-txt2db with a declarative language for Information Extraction because we did no have access to any of these languages. Then, this chapter describes experiments that access the features of E-txt2db that distinguish it from the Information Extraction frameworks. Section 5.2 reports how easy it is to refine the results of a Classification program using E-txt2db. Section 5.3 shows that E-txt2db is able to deal with several data formats without deep changes to the code. Section 5.4 illustrates that it is possible to change the algorithm used for a given technique without changing the specification of a program, thus making the program amenable to optimization. Finally, Section 5.5 shows how a Classification task can be customized with E-txt2db through the composition of several Classification models.

5.1 Easy specification and maintenance

To determine how easy it is to specify a Classification task, we compare the specification of Classification programs with E-txt2db and Minorthird. We decided to use Minorthird instead of Lingpipe, Gate or Mallet because, by the experiences we performed during our work, it was the easiest Information Extraction framework to introduce to the students.

During the tests, ten Computer Science students were divided into two groups. One group used Minorthird to develop a Classification program while the other group used the Java API of the E-txt2db CEE Engine. Each element of the group wrote his own program individually. To help them, we provided a document with the description of the assignment (Appendix A.1), the corpus format (Appendix A.2) and the functionalities of the main classes that we recommended them to use (Appendix A.3 for E-txt2db and Appendix A.4 for Minorthird).

To evaluate how easy it is to create a Classification program with each framework, we considered the following two measures: *(i)* the time spent by each student to write the program; *(ii)* the number of code lines of the program produced.

Table 5.1 shows the results of the evaluation of the programs written by students who used E-txt2db. Table 5.2 presents the results of the evaluation of the programs written by students who used Minorthird.

Table 5.1: Evaluation of the programs written by the students using E-txt2db

	Time spent (minutes)	Number of code lines
Student 1	35	21
Student 2	19	21
Student 3	14	21
Student 4	21	21
Student 5	27	21
Average	23.2	21

Table 5.2: Evaluation of the programs written by the students using Minorthird

	Time spent (minutes)	Number of code lines
Student 6	78	72
Student 7	59	32
Student 8	73	39
Student 9	83	39
Student 10	65	38
Average	71.6	44

By comparing the results shown in Table 5.1 and 5.2, we observe that the average time spent by the students to create the Classification program with Minorthird is almost three times higher than the average time spent to create a Classification program with E-txt2db. Moreover, the resulting Minorthird programs have an average number of code lines that is almost the double of the average number of code lines of the E-txt2db programs. These results further suggests that E-txt2db can help reduce the time it takes to develop a simple Classification program using machine learning techniques programs. Moreover, the code produced by E-txt2db is more concise than the code produced by Minorthird.

To prove that it is easy to maintain a Classification program specified with E-txt2db, we also asked each of the 10 Computer Science students to apply the following three modifications to their Information Extraction program: *(i)* to change the Classification technique used in their program; *(ii)* to change the class determined by their Classification model; and *(iii)* to create an Information Extraction program that classifies segments according to several classes. A detailed decription of the modifications required can be found in Appendix A.1 as well.

All ten students were able to apply the first and second modifications instantaneously. This proved that it is really easy to change both the technique and the class in E-txt2db and Minorthird.

The third modification had different reactions among each group. The students that used the E-txt2db framework performed this modification instantaneously. The students that used Minorthird did not answer the question immediately. Their first reaction was to say that it was not possible, because the document we provided said that Minorthird could not deal with multiple classes. After thinking for about two minutes, everyone got to the solution: replicate code to create one model for each class. Even though everyone got to a solution, none of the students using Minorthird voluntarily changed their code.

5.2 Local refinement

In Section 1.3, we argued that one of the advantages of decomposing the Information Extraction activity into several tasks was to turn it easy to locally refine the results of an Information Extraction program. We performed the experiment described in this section to show how to refine the results of an Information Extraction program with the E-txt2db framework to improve the accuracy of the results obtained.

First, we specified a simple Information Extraction program that uses a corpus of 52 documents to train a Support Vector Machine model to extract information about the location, the speaker, the starting time and the ending time of a seminar. This model is then evaluated with another corpus of 52 documents to determine the precision, recall and F-measure (with β=1) values. Both the training and the testing corpora are documents that contain e-mails with Carnegie Mellon University seminar announcements [1]. The corpus uses the default data format accepted by the E-txt2db framework which is plain text with XML tags. The Information Extraction program is the following one:

[1]http://www.cs.cmu.edu/ dayne/SeminarAnnouncements/

```
CREATE Classification MODEL AS seminarsModel
USING SVM TRAINED WITH ''./SeminarsTraining/''
TO FIND speaker, location, stime, etime

EVALUATE Classification OF speaker, location, stime, etime
FROM ''./SeminarsTesting/''
USING seminarsModel
```

The accuracy results returned by the evaluation of the Classification model are shown in Table 5.3:

Table 5.3: Accuracy of a Support Vector Machines Classification program

	speaker	location	stime	etime	all
recall	27.12%	58.18%	83.64%	59.26%	56.12%
precision	72.73%	74.42%	93.88%	72.73%	80.88%
F-measure	39.51%	65.31%	88.46%	65.31%	66.27%

The values presented in Table 5.3 show that the accuracy of the model is far from good. The values of recall are very weak when extracting speaker values. The values of precision are slightly better but can still be improved. Our objective during this test is to refine the results of the program.

One problem of the initial program is that it uses the default set of features to train the Support Vector Machines model. These features are very generic, so they may not fit the characteristics of the training and testing corpus. The improvement that can be introduced is to specify a set of new features in the Information Extraction program. These features are still generic but are used as basis for further iterations of the program. The modified Information Extraction program is as follows:

```
CREATE Classification MODEL AS seminarsModel
USING SVM TRAINED WITH ''./SeminarsTraining/''
TO FIND speaker, location, stime, etime
CAPTURING FEATURES CASE SENSITIVE VALUE,
            CHARACTER PATTERN,
            CHARACTER TYPE,
            CASE INSENSITIVE VALUE

EVALUATE Classification OF speaker, location, stime, etime
FROM ''./SeminarsTesting/''
USING seminarsModel
```

The features introduced are the case sensitive value of a segment (**CASE SENSITIVE VALUE**), an automatically generated regular expression the segment respects (**CHARACTER PATTERN**), the type of characters that constitute the segment (**CHARACTER TYPE**) and the case insensitive value of a segment (**CASE INSENSITIVE VALUE**). Section 4.4.1 provides more information about these features. Table 5.4 presents the results of the evaluation obtained after the refinement.

Table 5.4: Accuracy of a Support Vector Machines Classification program with customized features

	speaker	location	stime	etime	all
recall	25.42%	56.36%	87.27%	70.37%	57.65%
precision	65.22%	72.09%	94.12%	86.36%	81.29%
F-measure	36.59%	63.27%	90.57%	77.55%	67.46%

Even with generic features, the value of the F-measure of the program improved a little. The recall and precision of the ending time increased significantly. Unfortunately, the accuracy of location and speaker slightly decreased. The next refinement intends to increase the accuracy of these classes. Since speaker is typically a person name, which is a proper name, and location is typically a proper name or a common name, it would be useful to use features that indicate the Part-of-speech tag of a word. In order to implement this, we used an English corpus, called the Brown Corpus[2], with 400 English texts classified with Part-of-speech. This corpus was used to train a model to perform a Part-of-speech Classification. Then, we use this model to generate features for our seminars model. For efficiency purposes, we decided to use Hidden Markov Models in the *Part-of-speech* classifier with only half of the Brown Corpus. The code to generate the model was:

```
CREATE Classification MODEL AS HMMPOSTagger
USING HMM TRAINED WITH ''./brownTraining/''
TO FIND *
```

The code to create and evaluate the seminars model using *Part-of-speech* features is presented next:

```
CREATE Classification MODEL AS seminarsModel
USING SVM TRAINED WITH ''./SeminarsTraining/''
TO FIND speaker, location, stime, etime
CAPTURING FEATURES CASE SENSITIVE VALUE,
            CHARACTER PATTERN,
            CHARACTER TYPE,
            CASE INSENSITIVE VALUE,
            Classification MODEL HMMPOSTagger

EVALUATE Classification OF speaker, location, stime, etime
FROM ''./SeminarsTesting/''
USING seminarsModel
```

Table 5.5 presents the results of the evaluation. We observe that the Part-of-Speech classifier considerably improved the accuracy of the Classification of location and speaker. Unfortunately, it also degraded the accuracy of the Classification of the starting and ending time. We conclude that it would

[2]http://icame.uib.no/brown/bcm.html

Table 5.5: Accuracy of a Support Vector Machines Classification program with Part-of-speech tagging

	speaker	location	stime	etime	all
recall	42.37%	56.36%	83.64%	66.67%	61.22%
precision	71.43%	75.61%	90.20%	81.82%	80.54%
F-measure	53.19%	64.58%	86.79%	73.47%	69.57%

be interesting to classify the starting time and the ending time without using the Part-of-speech classifier. A solution for this is to create two different models: one that classifies segments as location and speaker using the Part-of-speech features, and another that classifies segments as starting time and ending time, without using the Part-of-speech features. Then, the two models can be joint in order to create a third model that classifies location, speaker, starting time and ending time. The code to create and evaluate the model is presented next:

```
CREATE Classification MODEL AS seminarsSpeakerLocationModel
USING SVM TRAINED WITH ''./SeminarsTraining/''
TO FIND speaker, location
CAPTURING FEATURES CASE SENSITIVE VALUE,
            CHARACTER PATTERN,
            CHARACTER TYPE,
            CASE INSENSITIVE VALUE,
            Classification MODEL HMMPOSTagger

CREATE Classification MODEL AS seminarsTimeModel
USING SVM TRAINED WITH ''./SeminarsTraining/''
TO FIND stime, etime
CAPTURING FEATURES CASE SENSITIVE VALUE,
            CHARACTER PATTERN,
            CHARACTER TYPE,
            CASE INSENSITIVE VALUE

CREATE Classification MODEL AS seminarsModel
USING Classification MODEL UNION seminarsSpeakerLocationModel,
                  seminarsTimeModel

EVALUATE Classification OF speaker, location, stime, etime
FROM ''./SeminarsTesting/''
USING seminarsModel
```

The results of the evaluation were significantly better with this approach as shown in Table 5.6. Remark that the accuracy values of the Classification as speaker did not change but the accuracy for all the other classes significantly improved.

The results of the Classification of speaker could still be improved. In order to achieve this, we

Table 5.6: Accuracy of a Support Vector Machines Classification program built with the union of two different Classification models

	speaker	location	stime	etime	all
recall	42.37%	60.00%	89.09%	77.78%	65.31%
precision	71.43%	78.57%	96.08%	95.45%	85.33%
F-measure	53.19%	68.04%	92.45%	85.71%	73.99%

added a dictionary with approximately 2000 person names (first names and surnames) to generate features for the Support Vector Machine model. The code to create and evaluate the model is presented next:

```
CREATE Classification MODEL AS seminarsSpeakerLocationModel
USING SVM TRAINED WITH ''./SeminarsTraining/''
TO FIND speaker, location
CAPTURING FEATURES CASE SENSITIVE VALUE,
            CHARACTER PATTERN,
            CHARACTER TYPE,
            CASE INSENSITIVE VALUE,
            Classification MODEL HMMPOSTagger,
            DICTIONARY ''./namesList.txt''

CREATE Classification MODEL AS seminarsTimeModel
USING SVM TRAINED WITH ''./SeminarsTraining/''
TO FIND stime, etime
CAPTURING FEATURES CASE SENSITIVE VALUE,
            CHARACTER PATTERN,
            CHARACTER TYPE,
            CASE INSENSITIVE VALUE

CREATE Classification MODEL AS seminarsModel
USING Classification MODEL UNION seminarsSpeakerLocationModel,
                  seminarsTimeModel

EVALUATE Classification OF speaker, location, stime, etime
FROM ''./SeminarsTesting/''
USING seminarsModel
```

Table 5.7: Accuracy of a Support Vector Machines Classification program with the help of a dictionary of person names

	speaker	location	stime	etime	all
recall	47.46%	58.18%	89.09%	77.78%	66.33%
precision	71.79%	78.05%	96.08%	95.45%	84.97%
F-measure	57.14%	66.67%	92.45%	85.71%	74.50%

As shown in Table 5.7 we remark that the accuracy of the Classification of speaker slightly improved and the accuracy of the Classification of location got worse. At this point, we could go on with the refinement and try to get even better results. Probably, this would be possible with more resources (e.g., dictionary of typical seminar locations or a better names dictionary). Due to time and space constraints, we decided to stop the refinement at this point.

The important idea to take from this test is that it is easy to locally refine a Classification model. Note that, when refining a model, a different program that uses the resulting model to perform a Classification task does not need to change. For instance, consider the following program:

```
CLASSIFY speaker, location, stime, etime
FROM ''./newSeminarTexts/''
USING seminarsModel
```

This program can be used both with the initial program or with the refined program without changing anything. This is even more important if the Classification model is used as a small part of a more complex Information Extraction program. In this case, a user can refine the Classification model without any impact in the rest of the program.

5.3 Independence of the data format

To evaluate the capability of E-txt2db to deal with several text formats, we conducted a test in which information about the location, the speaker, the starting time and the ending time of a seminar was extracted from two different corpora. With this test, we wanted to show the following two properties of E-txt2db: *(i)* the E-txt2db Classification models can be used to classify segments from texts with any data format; and *(ii)* the code to specify, execute or evaluate E-txt2db Classification models does not need to undergo many changes when the data source is modified.

The first corpus was composed by 484 e-mails containing CMU seminar announcements. The format of these e-mails is plain text with XML tags classifying text segments. From the 484 e-mails, we separated 50% for training and 50% for testing.

The second corpus is composed by 88 Web pages from the UCLA Department of Materials Science and Engineering. In this case, we could not use the same format because the HTML tags would be confused by the XML tags that classify the segments. For this reason, we replaced the symbols "<" and ">" by the symbols "[" and "]". From the 88 Web pages, we separated 50% for training and 50% for testing.

Next, we present an excerpt of such text:

```
(...)
<table border=0 cellspacing=2 width=550><td>
<P align=left><IMG height=138 src=blank.gif width=114></p><B><FONT>
</td>
<td><face=Palatino size=4><br>[stime]10:30[/stime] - [etime]Noon[/etime]
<br>
[location]Penthouse - 8500 Boelter Hall[/location]
</FONT></td></table>
<hr>
<font size=+2 >
Polymer and Biopolymer Mediated Self-Assembly of Nanocomposite Materials<br>
</font>
</p>
<font size=+1 >
[speaker]Vincent Rotello[/speaker]<br>
</font>
<font size=+1 >
Professor, Department of Chemistry
<br>
University of Massachusetts</font>
(...)
```

For each of the corpora, a program was written to create and evaluate a Classification model that uses machine learning techniques. Because our interest in this test was not accuracy, HMM were chosen for fast training and evaluation. The resulting Information Extraction program to classify e-mails corpus is shown next:

```
CREATE Classification MODEL AS seminarsHMMModel
USING HMM TRAINED WITH ''./SeminarsTraining/''
TO FIND speaker, location, stime, etime

EVALUATE Classification OF speaker, location, stime, etime
FROM ''./SeminarsTesting/''
USING seminarsHMMModel
```

and the output produced by the program is the following one:

```
Recall: 0.7084257
Precision: 0.71237457
F-measure: 0.71039474
```

The Information Extraction program for classifying the Web pages corpus is shown next:

```
SET BEGIN TAG ''[''
SET END TAG '']''

CREATE Classification MODEL AS seminarsHMMModel
USING HMM TRAINED WITH ''./SeminarsHTMLTraining/''
TO FIND speaker, location, stime, etime

EVALUATE Classification OF speaker, location, stime, etime
FROM ''./SeminarsHTMLTesting/''
USING seminarsHMMModel
```

and the output produced by the program was:

```
Recall: 0.81714284
Precision: 0.94078946
F-measure: 0.8746177
```

E-txt2db can deal with both plain text and HTML input texts. The fact that the user must provide the corpus with XML tags classifying segments is no problem when dealing with HTML or XML texts, because the user can configure the program to use variations of XML tags to classify the segments. This configuration is shown in the first two lines of the program that classifies Web pages.

Since machine learning techniques are available, a Classification program specified with E-txt2db does not change much when the input data source changes. This can be observed in this test since the two programs created are very similar. The differences in the code produced are the lines to configure the format of the training corpus and the paths to the files.

5.4 Amenability to optimization

In the context of this thesis, we did not study any optimization technique for the Classification task. Therefore, we do not evaluate the use of optimization techniques in E-txt2db. Instead, we prove that E-txt2db is amenable to optimization.

The scenario built for this purpose consists of an Information Extraction program to extract person names from texts. This program used a dictionary of person names. We used two algorithms for implementing the dictionary-based Classification technique. The goal was to show that it is possible to internally replace the dictionary-based Classification algorithm without changing the specification of the Information Extraction program to extract proper names. We used a non-classified version of the *Brown corpus* with a set of 400 English texts. The Information Extraction program written is the following one:

```
CREATE Classification MODEL AS namesModel
USING DICTIONARY ''./resources/namesDictionaty.txt''

CLASSIFY name
FROM ''./brownUntagged/''
USING namesModel
```

Note that in this case we execute the model instead of evaluating it. This happens for the following two reasons: *(i)* in this case the values of recall, precision and F-measure are not important for the experiment; and *(ii)* we did not have access to any corpus with segments tagged as person names.

The first implementation of the dictionary-based Classification algorithm was developed from scratch using the Map Java class to store the entries of the dictionary and their classes. We denote this implementation as the *Map algorithm*. For each segment of the text, the algorithm checks whether it is an entry of the dictionary. If it is, it classifies the segment with the corresponding class. When using the *Map algorithm*, the Information Extraction program finished the Classification task in 4:49 minutes.

The second implementation of the dictionary-based Classification algorithm uses a Lingpipe class called *ExactDictionaryChunker*. This class implements the *Aho-Corasick algorithm* (Aho & Corasick, 1975) for the dictionary-based Classification. This algorithm is linear in the number of segments in the input plus the number of classified segments. When using the *Aho-Corasick algorithm*, the program finished the Classification task much faster, in 2:01 minutes. An E-txt2db optimizer would choose the second implementation of the Classification operator.

The most important idea to take from this test is that it is possible to change the algorithm used in the Classification task without changing the specification.

5.5 Customization

Since we only developed the Classification operator of E-txt2db in the context of this thesis, it was not possible to evaluate the possibility to customize the whole Information Extraction process. However, by analyzing the test described in Section 5.2, we can see how a Classification task can be customized. In this test, the following two types of customization are highlighted: (i) union of different Classification models; and (ii) use of Classification models to generate features.

The union of Classification models creates a new model that is capable of classifying segments with all the classes from the original ones. The experiment reported in Section 5.2 illustrates this type of customization. In fact, the Classification of speaker and location is separated from the Classification of starting time and ending time.

Using Classification models to generate features is another form of composition of Classification models. The objective of this type of customization is to produce hints that may help in the Classification process of another model. The experiment reported in Section 5.2 shows this type of customization. A Classification model for Part-of-speech tagging gives hints to classify segments as speaker or location.

Even if these customizations are local to the Classification task, they may be very useful because they offer the possibility to generate complex Classification models using simple ones that can be locally refined.

6 Conclusions

This chapter presents the main conclusions of this dissertation. First, we present a summary of the work developed in this thesis, highlighting its main contributions. Second, we point at some of the limitations of this work. Third, we present some ideas that we did not explore in the context of this thesis but that may be interesting to explore in future work. Finally, this chapter ends with a discussion about the viability of the proposed solution for the development of Information Extraction programs.

6.1 Summary and contributions

This thesis presents E-txt2db, a framework for Information Extraction that offers a set of declarative operators based on the semantics of the Information Extraction tasks described in Section 2.3: Segmentation, Classification, Association, Normalization and Coreference resolution. Due to time constraints, it was not possible to create operators for all the Information Extraction tasks, so, we decided to developed a proof-of-concept for the Classification operator. To develop this proof-of-concept, we started with a formalization of the Classification operator, then we created a Java API to create, execute and evaluate Classification models (CEE engine) and finally we proposed an SQL-like syntax for the specification of Classification programs.

The main contributions of this thesis are:

- Analysis and decomposition of the Information Extraction activity into tasks with a description of the most used techniques for each task (Section 2.3). This contribution resulted in a short paper published at Inforum 2009 (Simões et al., 2009)
- State-of-the-art of the existing solutions for specification and execution of Information Extraction programs, with a special focus on Information Extraction frameworks and declarative languages for Information Extraction (Chapter 3)
- A framework, called E-txt2db, instantiated for the Classification task (Chapter 4). The instantiation of E-txt2db addressed in this thesis encloses:
  - A proposal of an operator for Classification as an extension of the Relational Algebra
  - A Java API for the creation, execution and evaluation of Classification models

  - An SQL-like syntax for the Classification operator that supports the declarative creation, execution and evaluation of Classification models

- Validation of the Classification operator through tests that highlight the advantages of using the E-txt2db for the Classification task (Chapter 5)

6.2 Limitations

The work developed in the context of this thesis has the following limitations:

- We did not create operators for all the Information Extraction tasks. We decided to focus on the Classification operator and we proposed its formalization, implemented its CEE engine and proposed its SQL-like syntax. We made the decision of implementing the Classification operator for the following reasons: *(i)* the Classification task produces very important results in the context of Information Extraction. Usually people are only interested in extracting entities and do not need to extract relationships between them; *(ii)* there are some Natural Language Processing activities that can be developed as a Classification task (e.g., Part-of-speech tagging, Named Entity Recognition); *(iii)* there are several implementations of machine learning techniques for Classification in open source frameworks that could be used for our proof-of-concept, enriching the number of techniques available by our operator.

  This limitation had some impact in the evaluation of the proposed solution because we could not validate the capability of customizing the Information Extraction process with the composition and reordering of several operators.

- The focus of our work was on the specification of Information Extraction programs. So, even though we proved in Section 5.4 that it is possible to change the execution algorithm without changing the specification, we did not use any real optimization techniques for the Classification operator.

- The CEE engine for the Classification operator does not provide any solution to compute the confidence value of the classifications. For rule-based techniques this is not a problem because the rules are developed by a Human Being that usually knows if he can trust them. As for machine learning techniques, it would be important for the system to return the confidence value because the Classification model was not developed by a Human and a user may want to know if he can trust an answer or not. The information about the confidence value of an answer becomes even more important if a user wants to create semi-supervised or unsupervised algorithms for the Classification task because these approaches usually use high confidence classifications to improve the model over time.

6.3 Future work

During the development of this thesis, we identified the following interesting ideas that can be used to improve E-txt2db:

- This thesis proposed a declarative language specification language and a creation, execution and evaluation engine for the Classification operator. In future work, it is important to develop the operators for Segmentation, Association, Normalization and Coreference resolution within the E-txt2db framework. The development of these operators should be based on the principles already described in this thesis. Section 2.3 presents each of the Information Extraction tasks, describing some of the most common techniques that should be available for each operator. Some of the techniques are available in open-source tools like Lingpipe and Gate that are presented in Section 3.1. There are other tools that can be used to implement these operators. OpenNLP[1] offers algorithms for Segmentation and Coreference resolution and syntactic parsing tools that can be used for the Association task. NLTK[2] also offers algorithms for Segmentation, parsing tools and even an interface to access the WordNet, which can be useful for the Normalization and Coreference resolution tasks.

  The architecture of the Classification operator that is shown in Section 4.1 is generic enough to be applied to the other operators. It is expected that each operator contains two components: a CEE engine and a Library of techniques.

- Optimization of the Information Extraction process is today an hot topic for the scientific community. This makes the optimization of the E-txt2db operators one of the most interesting topics we left for future work. Some of the declarative languages described in Section 3.2 are amenable to automatic optimization. (Shen et al., 2007) describes how XLog execution plans are optimized through a cost model and heuristics and (Reiss et al., 2008) describes techniques to optimize the System T operators and how an execution plan is created. These must be analyzed to decide whether they are useful in the context of the E-txt2db framework.

  Other work on Information Extracion optimization is independent of the language used, unlike the two refered above. This should also be taken into account when addressing the optimization of E-txt2db operators. In (Agichtein & Gravano, 2003), machine learning techniques are used to derive queries that identify interesting documents for Information Extraction. (Jain & Ipeirotis, 2009) proposes a model to estimate the quality of an Information Extraction process and to retrieve interesting documents. Finally, (Ipeirotis et al., 2007) presents a cost-based optimizer for text centric tasks.

[1]http://opennlp.sourceforge.net/
[2]http://www.nltk.org/code

- Many of the input texts of an Information Extraction process are stored in relational databases. For this reason, it would be interesting to integrate the E-txt2db framework with a Relational Database Management System. Some of the challenges of this integration include the definition of a data model for the input and output data and extending the SQL language with the SQL-like syntax for the E-txt2db operators.

- The following small improvements could be applied to the E-txt2db framework. First, it would be interesting to have a mechanism to get the confidence value of the answers returned by E-txt2db. Second, the interface should be equipped with syntax highlighting and auto-complete features to help the user specifying the Information Extraction programs.

6.4 Discussion

There are several solutions to specify and execute Information Extraction programs. Among them, the most used ones are programming languages, declarative languages for Information Extraction and ETL tools. All of these solutions have advantages and disadvantages that were described in Section 1.2. Our analysis led to the conclusion that, from these four solutions, the most competitive ones are the Information Extraction frameworks and the declarative languages for Information Extraction.

This thesis proposes a framework called E-txt2db that aims at combining both the advantages of Information Extraction frameworks and the advantages of declarative languages for Information Extraction. Analogously to the Information Extraction frameworks, E-txt2db offers a wide variety of pre-implemented rule-based and machine learning techniques that can be used to specify Information Extraction programs. This turns the specification and maintenance of the programs easy because the user does not need to develop the Information Extraction algorithms from scratch. Code reusability is possible since extraction models are created and then stored for further use. Similarly to the declarative languages for Information Extraction, the resulting Information Extraction programs are amenable to optimization. Finally, due to the high level of abstraction offered by the E-txt2db operators, it is very easy to learn how to use.

This thesis focused only on the E-txt2db framework Classification operator. There is considerable amount of work left to be done until E-txt2db reaches its full potencial. However, the evaluation results presented in Section 5, gives optimistic perspectives about the advantages of this framework. In the future, we expect that the work developed for the creation of the other Information Extraction operators and for the automatic optimization of the Information Extraction process turns E-txt2db into a very comprehensive solution for the specification and execution of Information Extraction programs.

I
Apêndice

A Apendix

A.1 Assignment used for the testing of E-txt2db and Minorthird

A.1.1 Introduction

In the context of Information Extraction, *classification* is a task that determines a class for each segment of the text. By other words, if we want to populate a data structure with information from the text, it determines the field of the data structure where the input segment fits. The following example shows a text with the announcement about a seminar:

CENTER FOR INNOVATION IN LEARNING (CIL)
EDUCATION SEMINAR SERIES
"Using a Cognitive Architecture to Design Instructions"

Joe Mertz
Center for Innovation in Learning, CMU

Friday, February 17
12:00pm-1:00pm
Student Center Room 207 (CMU)
ABSTRACT: In my talk, I will describe how a cognitive model was used as a simulated student to help design instructions for training circuit board assemblers. The model was built in the Soar cognitive architecture, and was initially endowed with only minimal prerequisite knowledge of the task, and an ability to learn instructions. Lessons for teaching expert assembly skills were developed by iteratively drafting and testing instructions on the simulated student.

Please direct questions to Pamela Yocca at 268-7675.

We can perform a classification task over this text in order to identify, for instance, the seminar speaker, the location, the starting time and the ending time. We assign an XML tag to each text segment that is classified as one of the above classes. The speaker, location, starting time and ending time are assigned to the tags *speaker*, *location*, *stime* and *etime* respectively. The result is presented in the following text:

```
CENTER FOR INNOVATION IN LEARNING (CIL)
EDUCATION SEMINAR SERIES
"Using a Cognitive Architecture to Design Instructions"

<speaker>Joe Mertz</speaker>
Center for Innovation in Learning, CMU

Friday, February 17
<stime>12:00pm</stime>-<etime>1:00pm</etime>
<location>Student Center Room 207</location> (CMU)
ABSTRACT: In my talk, I will describe how a cognitive model was used as a simulated student to help
design instructions for training circuit board assemblers. The model was built in the Soar cognitive
architecture, and was initially endowed with only minimal prerequisite knowledge of the task, and an
ability to learn instructions. Lessons for teaching expert assembly skills were developed by iteratively
drafting and testing instructions on the simulated student.

Please direct questions to Pamela Yocca at 268-7675.
```

In order to perform a classification task, it is possible to use *rule-based* ou *machine learning* approaches. This document will guide you in the development of a program that uses a machine learning approach to the classification task.

A.1.2 Classification based on machine learning

This section describes the concepts that you need to be familiar with in order to perform a classification task based on machine learning.

A.1.2.1 Overview of machine learning

Usually, machine learning is used to overcome the difficulty to find rules that explicitly solve a given task. Instead of finding these rules, the user provides some examples to a machine learning algorithm so that it can discover patterns and develop its own rules. The result of the process is called a *classifier*. Thus, when we use a machine learning technique, we split the development process into two phases:

- *Training Phase*: the user gives the examples to the algorithm that produces a classifier
- *Testing Phase*: the user uses the classifier to perform a task for different instances

A.1.2.2 Training phase for Information Extraction classification

In the classification task of Information Extraction, the training phase consists on giving a set of texts with *segments* already classified (called the *training corpus*) to the algorithm. The result of this phase is usually called an *Annotator*.

The program can use different algorithms. The most popular ones are the following: Hidden Markov Models (HMM), Maximum Entropy Markov Models (MEMM), Conditional Random Fields (CRF) and Support Vector Machines (SVM). In your task you will not need to know how any of these techniques work. Just assume they are "black boxes". If you provide some examples to them, they will produce an Annotator.

A.1.2.3 Testing phase for Information Extraction classification

The testing phase in a classification task of Information Extraction uses the Annotator developed during the training phase in order to classify segments of text not classified (denoted as *testing corpus*).

A.1.3 Finding the resources you need

To perform this task, you will have at your disposal an Eclipse workspace. You will not need to load any library to the workspace nor create any class (unless you wish to). You just need to modify the *main* method of the *TestClassificationOperators* class.

The training corpus is stored in a folder inside the Eclipse workspace. The path you can use in your Java program to access it is: "*./resources/repository/data/SeminarPartial/*". The testing corpus is also stored in the Eclipse workspace in the path "*./resources/test/*".

A.1.4 Building the base of your program

You should start by developing a base classification program. The following steps will guide you to achieve that:

1. **Train a model**: develop an HMM annotator to classify text segments as *location*. You shall use the training corpus at your disposal in this step. Remember that the framework you are using is able to load entire directories. In order to load the training corpus you can create a Java File object providing the path to the directory containing all the training files as a parameter.

2. **Classify segments of plain text**: use the annotator you developed in the previous step in order to classify the segments of test files we provided.

3. **Print the results to the screen**: You do not need to present the full tagged text (we do not need to evaluate your ability to manipulate strings). Just print the segments that were classified as *location*.

A.1.5 Changing your code

After you finished developing your base program, you should be able to change it. You should go through the following steps to achieve that:

1. **Change the machine learning technique**: in the base program you used a model based on HMM. But when we develop a classification program we want to test different techniques in order to understand which one will be better for the task we are performing. Change your program to use different techniques. We want you to use the following techniques: MEMM, CRF and SVM.

2. **Change the class you extracted**: you developed an annotator that is able to classify text segments as *location* but there is other interesting information in the text. Change the type of information you want to extract to each of the following classes: *speaker*, *stime* and *etime*

3. **Annotate a given text with multiple types**: In step 2 you changed the extraction type but usually what we want is for a document to be classified not with one but with many tags. Change your program in order to classify segments from a single text with all the following possible classes: *speaker*, *location*, *stime*, *etime*, *sentence*, *paragraph*.

A.2 Description of the Training Corpus

The format used for the training corpus in our classification tasks is a text file with XML tags classifying text segments. An example of such text is presented next:

```
The final session will be held on <date>Tuesday August 1</date> at
<time>10am</time> in <location>Hamburg Hall 1000</location>
```

In this example, "Tuesday August 1"is classified as a date, "10am"is classified as time and "Hamburg Hall 1000"is classified as a location.

E-txt2bd offers methods to load such texts to a Java object with which you can easily access text spans with a given classification.

A.3 Description of the E-txt2db classes

A.3.1 E-txt2db description

E-txt2bd offers classes to train, evaluate and test classification models for Information Extraction tasks.

This section presents the *E-txt2bd* classes that can be used for the classification task. Here, we give a small description of the functionalities of each class you should use but we still recommend that you read of the *e-txt2db* API that is available in `http://web.ist.utl.pt/~ist155840/tese/javadoc/`.

A.3.1.1 etxt2db.api.ClassificationModelCreator

A *ClassificationModelCreator* offers methods to train a *ClassificationModel*. The result of the training is an object that implements the *ClassificationModel* interface. The training function allows us to choose between different rule-based and machine learning techniques using a *ClassificationModelCreator.MLTechnique*.

A.3.1.2 etxt2db.api.ClassificationModel

The *ClassificationModel* is the Minorthird interface for Information Extraction classifiers. The objects of classes that implement this interface offer methods that allow, given a *TextLabels* document, to classify text spans of this document with the class the *ClassificationModel* was trained for.

A.3.1.3 etxt2db.api.ClassificationTrainer.MLTechnique

The *ClassificationTrainer.MLTechnique* is an enumerate that contains identifiers for the machine learning techniques used in *E-txt2bd*. The techniques offered are:

- *ClassificationModelCreator.MLTechnique.CRF*: Conditional Random Fields Model technique.
- *ClassificationModelCreator.MLTechnique.HMM*: Hidden Markov Model technique.
- *ClassificationModelCreator.MLTechnique.MEMM*: Maximum Entropy Markov Model technique.
- *ClassificationModelCreator.MLTechnique.SVM*: Support Vector Machines Model technique.

A.3.1.4 etxt2db.api.ClassificationEvaluator

In order to evaluate a *ClassificationModel*, it is important to have functions that measure the recall and precision of the *ClassificationModel* for a given task. This object offers functions that evaluates an annotator given a training corpus and a testing corpus.

A.3.1.5 etxt2db.api.ClassificationExecutor

A *ClassificationExecutor* object allows to show the results of a classification task. It offers several possibilities to show the results, printing it to the screenm creating a file with the original text annotated with XML tags or returning the results in a Map object.

A.4 Description of the Minorthird classes

A.4.1 Minorthird description

Minorthird is a package that offers a Java implementation for several machine learning techniques used to perform Information Extraction tasks.

This section presents the classes and interfaces of the *Minorthird* package that may be useful for a classification task. We give a small description of the functionalities of each class you should use but we still recommend that you read the *Minorthird* API that is available in `http://minorthird.sourceforge.net/javadoc/index.html`.

A.4.1.1 edu.cmu.minorthird.text.TextLabels

This is the format used by *Minorthird* to store texts for training and testing classifiers. The classes that implement this interface are forced to implement methods that allow the access to the original text or tagged text spans.

An object of a class that implements the *TextLabels* interface can be loaded from a file using the class *TextBaseLoader* that will be presented later.

An important sub-interface of the *TextLabels* interface is the *MutableTextLabels* (also present in the package edu.cmu.minorthird.text). Objects that implement *TextBaseLoader* may only be classified in loading time because *TextBaseLoader* does not force the creation of methods to classify text spans after loading. An object that implements *MutableTextLabels* is forced to implement such methods allowing to assign text spans to a given class in classification time.

A.4.1.2 edu.cmu.minorthird.text.Span

A *Span* represents a segment in a text. In a classification task, the *Span* interface is very important, because *Minorthird* uses it to store the classification of different segments. Given a classified *TextLabels* object, it is possible to ask for all the Spans classified with a given type.

A.4.1.3 edu.cmu.minorthird.text.TextBaseLoader

The objects of this class are able to load texts from a file and export them as a *TextLabels* object.

A.4.1.4 edu.cmu.minorthird.text.learn.AnnotatorLearner

To train a classifier in *Minorthird*, we need to use two types of classes: *AnnotatorLearner* and *AnnotatorTeacher*. The *AnnotatorLearner* object learns a statistical model from a training corpus. There are many different implementations of this class corresponding to the machine learning techniques available in Minorthird. The most important ones are:

- *edu.cmu.minorthird.ui.Recommended.CRFAnnotatorLearner:* Conditional Random Fields Model learner.
- *edu.cmu.minorthird.ui.Recommended.HMMAnnotatorLearner:* Hidden Markov Model learner.
- *edu.cmu.minorthird.ui.Recommended.MEMMLearner:* Maximum Entropy Markov Model learner.
- *edu.cmu.minorthird.ui.Recommended.SVMCMMLeaner:* Support Vector Machines Model learner.
- *edu.cmu.minorthird.ui.Recommended.VPCMMLearner:* Voted Perceptron Conditional Markov Model learner.
- *edu.cmu.minorthird.ui.Recommended.VPHMMLearner:* Voted Perceptron Hidden Markov Model learner.

It is important to note that an *AnnotatorLearner* is only able to determine whether a text span belongs to a single class. It is not possible to directly perform a classification task in which we want to select between different classes. For example, if you want to perform a classification task in which you want to classify segments as person, location or neither, it is not possible. By default, a segment that belongs to a class is tagged with the String "_predicted".

An *AnnotatorLearner* is not able to learn the statistical model by itself. It needs the help of *AnnotatorTeacher* objects (see section A.2.5).

A.4.1.5 edu.cmu.minorthird.text.learn.AnnotatorTeacher

An AnnotatorTeacher object is responsible for teaching a statistical model to an *AnnotatorLearner* object. Usually, for the classification task, we use an implementation of this class called *TextLabelsAnnotatorTeacher* (also present in the package edu.cmu.minorthird.text.learn). The constructor for *TextLabelsAnnotatorTeacher* receives a training corpus in the format of a *TextLabels* object and a string corresponding to the class we will use in the classification task. Training an *AnnotatorLearner* with an *AnnotatorTeacher* results in the creation of an *Annotator* object (see section A.2.6).

A.4.1.6 edu.cmu.minorthird.text.Annotator

The *Annotator* is the Minorthird interface for Information Extraction classifiers. The classes that implement this interface offer methods that allow, given a *TextLabels* document, to classify text spans of this document with the class the *Annotator* was trained for.

Bibliography

Agichtein, E., & Gravano, L. (2003). Querying text databases for efficient information extraction. In *Proceedings of the* 19^{th} *Institute of Electrical and Electronics Engineers (IEEE) International Conference on Data Engineering (ICDE 2003)* (pp. 113–124). Bangalore, India.

Aho, A. V., & Corasick, M. J. (1975). Efficient string matching: an aid to bibliographic search. In *Communications of the ACM* (Vol. 18, pp. 333–340). New York, NY, USA.

Applet, D., & Israel, D. (1999). Introduction to information extraction technology. In *Procedings of the* 16^{th} *International Joint Conference on Artificial Intelligence.* Stockholm, Sweden.

Baumgartner, R., Flesca, S., & Gottlob, G. (2001). Declarative information extraction, web crawling, and recursive wrapping with lixto. In *Procedings of the Logic Programming and Nonmonotonic Reasoning (LPNMR 2001).* Vienna, Austria.

Brill, E. (1992). A simple rule-based part-of-speech tagger. In *Proceedings of the* 3^{rd} *Conference on Applied Natural Language Processing (ANLP 1992)* (pp. 152–155). Trento, Italy.

Canisius, S., & Sporleder, C. (2007). Bootstrapping information extraction from field books. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Conference on Computational Natural Language Learning (EMNLP-CoNLL 2007).* Prague, Czech Republic.

Cardie, C., & Wagstaff, K. (1999). Noun phrase coreference as clustering. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Workshop on Very Large Corpora (EMNLP-WVLC 1999)* (pp. 82–89). New Brunswick, NJ, USA.

Carlson, A., & Schafer, C. (2008). Bootstrapping information extraction from semi-structured web pages. In *Proceedings of European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2008)* (pp. 195–210). Berlin, Germany.

Carreira, P., Galhardas, H., Lopes, A., & Pereira, J. (2007). One-to-many data transformations through data mappers. In *Data and Knowledge Engineering* (Vol. 62, p. 483-503). Amsterdam, Netherlands.

Ceri, S., Gottlob, G., & Tanca, L. (1989). What you always wanted to know about datalog (and never dared to ask). In *Knowledge and Data Engineering, Institute of Electrical and Electronics Engineers (IEEE) Transactions* (Vol. 1, pp. 146–166).

Cohen, W. (2004). *Minorthird: Methods for identifying names and ontological relations in text using heuristics for inducing regularities from data.* http://minorthird.sourceforge.net.

Cowie, J., & Lehnert, W. (1996). Information extraction. In *Special natural language processing issue of the communications of the ACM* (Vol. 39, pp. 80–91). New York, NY, USA.

Cunningham, H. (2005). Information Extraction, Automatic. In *Encyclopedia of Language and Linguistics, 2nd Edition.* Elsevier.

Cunningham, H., Maynard, D., Bontcheva, K., & Tablan, V. (2002). Gate: an architecture for development of robust HLT applications. In *Proceedings of the* 40^{th} *Annual Meeting on Association for Computational Linguistics (ACL 2002)* (pp. 168–175). Morristown, NJ, USA.

Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V., Ursu, C., Dimitrov, M., et al. (2009). *Developing language processing components with gate version 5 (a user guide).* http://gate.ac.uk/sale/tao/split.html.

Farmakiotou, D., Karkaletsis, V., Koutsias, J., Sigletos, G., Spyropoulos, C. D., & Stamatopoulos, P. (2000). Rule-based named entity recognition for greek financial texts. In *Proceedings of the Workshop on Computational Lexicography and Multimedia Dictionaries (COMLEX 2000)* (pp. 75–78). Pyrgos, Greece.

Fellbaum, C. (Ed.). (1998). *Wordnet: An electronic lexical database (*1^{st} *edition).* MIT Press.

Forney, G. D. (1973). The Viterbi algorithm. In *Proceedings of the Institute of Electrical and Electronics Engineers (IEEE)* (Vol. 61, pp. 268–278).

Gottlob, G., Koch, C., Baumgartner, R., Herzog, M., Wien, D. T., & Flesca, S. (2004). The Lixto Data Extraction Project: Back and forth between theory and practice. In *Proceedings of the* 23^{rd} *ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems (PODS 2004).* Paris, France.

Grishman, R. (1997). Information extraction: techniques and challenges. In *Information Extraction International Summer School (SCIE 1997)* (pp. 10–27). Frascati, Italy.

Haizhou, L., Shuanhu, B., & Zhiwei, L. (1998). Chinese sentence tokenization using viterbi decoder. In *Proceedings of the International Symposium on Chinese Spoken Language Processing (ISCSLP 1998).* Singapore.

Hamilton, B. (2007). *Sql server integration services (*1^{st} *edition).* O'Reilly.

Hansen, W. (1989). The computational power of an algebra for subsequences. In *Technical Report CMU-ITC-083.* Information Technology Center, Carnegie-Mellon University.

Ipeirotis, P., Agichtein, E., Jain, P., & Gravano, L. (2007). Towards a query optimizer for text-centric tasks. In *ACM Transactions on Database Systems (TODS 2007)* (Vol. 32). New York, NY, USA.

Isozaki, H., & Kazawa, H. (2002). Efficient support vector classifiers for named entity recognition. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING02)* (pp. 390–396). Taipei, Taiwan.

Jain, A., & Ipeirotis, P. G. (2009). A quality-aware optimizer for information extraction. In *ACM Transactions on Database Systems (TODS 2009)* (Vol. 34). New York, NY, USA.

Kaiser, K., & Miksch, S. (2005). Information extraction: A survey. In *Technical Report Asgaard-TR-2005-6.* Vienna University of Technology, Institute of Software Technology and Interactive Systems.

Klug, A. (1982). Equivalence of relational algebra and relational calculus query languages having aggregate functions. In *Journal of the ACM (JACM)* (Vol. 29, pp. 699–717). New York, NY, USA.

Lafferty, J., McCallum, A., & Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Procedings of the 18th International Conference on Machine Learning (ICML 2001)* (pp. 282–289). Williamstown, MA, USA.

McCallum, A. (2005). Information extraction: Distilling structured data from unstructured text. In *ACM Queue* (Vol. 3, pp. 48–57). New York, NY, USA.

McCallum, A., Freitag, D., & Pereira, F. (2000). Maximum entropy markov models for information extraction and segmentation. In *Procedings of the 17th International Conference on Machine Learning (ICML 2000)* (pp. 591–598). Stanford, CA, USA.

McCallum, A. K. (2002). *Mallet: A machine learning for language toolkit.* http://mallet.cs.umass.edu.

Miller, S., Crystal, M., Fox, H., Ramshaw, L., Schwartz, R., Stone, R., et al. (1998). Algorithms that learn to extract information–BBN: Description of the SIFT system as used for MUC-7. In *Proceedings of the 7th Message Understanding Conference (MUC-7)* (pp. 75–89). San Francisco, CA, USA.

Reiss, F., Raghavan, S., Krishnamurthy, R., Zhu, H., & Vaithyanathan, S. (2008). An algebraic approach to rule-based information extraction. In *Proceedings of the 24th Institute of Electrical and Electronics Engineers (IEEE) International Conference on Data Engineering (ICDE 2008)* (pp. 933–942). Cancun, Mexico.

Santos, M. (2002). *Extraindo regras de associação a partir de textos.* Mestrado em Informática Aplicada, Pontifícia Universidade Católica do Paraná, Curitiba, Brasil.

Sapsford, R., & Jupp, V. (1996). *Data collection and analysis.* Sage Publications Ltd.

Sekine, S., Grishman, R., & Shinnou, H. (1998). A decision tree method for finding and classifying names in japanese texts. In *Proceedings of the* 6^{th} *Workshop on Very Large Corpora (WVLC-98)* (pp. 171–178). Montreal, Canada.

Shen, W., DeRose, P., McCann, R., Doan, A., & Ramakrishnan, R. (2008). Toward best-effort information extraction. In *Proceedings of the the* 27^{th} *SIGMOD International Conference on Management of data* (pp. 1031–1042). Vancouver, Canada.

Shen, W., Doan, A., Naughton, J. F., & Ramakrishnan, R. (2007). Declarative information extraction using datalog with embedded extraction predicates. In *Proceedings of the* 33^{th} *International Conference on Very Large Databases (vldb 2007)* (pp. 1033–1044). Vienna, Austria.

Silva, E. F. A., Barros, F. A., & Prudêncio, R. B. C. (2005). Uma Abordagem de Aprendizagem Híbrida para Extracção de Informação em Textos Semi-Estruturados. In *XXV Congresso da Sociedade Brasileira de Computação (SBC 2005).* Rio Grande do Sul, Brasil.

Simões, G., Galhardas, H., & Coheur, L. (2009). Information extraction sub-tasks: a survey. In *Proceedings of INFORUM 2009 - Simpósio de Informática.* Lisbon, Portugal.

Smith, L., Tanabe, L., Rindflesch, T., & Wilbur, W. J. (2005). Medtag: A collection of biomedical annotations. In *Proceedings of the Joint Meeting of the ISMB BioLINK Special Interest Group on Text Data Mining and the ACL Workshop on Linking Biological Literature, Ontologies and Databases* (pp. 32–37). Detroit, MI, USA.

Soderland, S., Cardie, C., & Mooney, R. (1999). Learning information extraction rules for semi-structured and free text. In *Special issue on natural language learning of the Machine Learning Journal of ACM* (Vol. 34, pp. 233–272).

Teahan, W. J., Wen, Y., Mcnab, R., & Witten, I. H. (2000). A compression-based algorithm for chinese word segmentation. In *Computational Linguistics Journal of ACM* (Vol. 26, pp. 375–393).

Wallach, H. (2004). Conditional random fields: An introduction. In *Technical Report Asgaard-MS-CIS-04-21.* Department of Computer and Information Science, University of Pennsylvania.

Zelenko, D., Aone, C., & Richardella, A. (2003, February). Kernel methods for relation extraction. In *Journal of machine learning research* (Vol. 3, pp. 1083–1106).