UNIVERSITY OF GENOA

SCHOOL OF
NATURAL SCIENCES,
MATHEMATICS AND PHYSICS

DEPARTMENT OF
COMPUTER SCIENCE,
BIOENGINEERING, ROBOTICS
AND SYSTEMS ENGINEERING

MASTER’S DEGREE IN COMPUTER SCIENCE

MOoD-TC: A GENERAL PURPOSE
MULTILINGUAL ONTOLOGY DRIVEN
TEXT CLASSIFIER

CANDIDATE
Silvio Beux

ADVISORS
Prof. Viviana Mascardi
Prof. Paolo Rosso

REVIEWER
Prof. Maura Cerioli

July 29, 2015
ACADEMIC YEAR 2014/15
# CONTENTS

1 Introduction and Motivation 5

2 Background 9
   2.1 Ontologies 9
   2.2 Text Classification 13
   2.3 Word Sense Disambiguation 20
   2.4 Sentiment Analysis 23

3 State of the Art 27

4 Tools Used 31
   4.1 BabelNet 31
      4.1.1 WordNet 32
      4.1.2 Wikipedia 33
      4.1.3 Integration 34
   4.2 TreeTagger 34
   4.3 Jena 36
   4.4 SentiWordNet 37

5 MOoD-TC 41
   5.1 User Interface 43

6 Text Classifier 45
   6.1 Document Loading 48
   6.2 Text Extraction 50
   6.3 Ontology Loading 50
   6.4 Language Detection 53
CHAPTER 1

INTRODUCTION AND MOTIVATION

In recent years the large amount of electronic data made available from a variety of sources, which include unstructured and semi-structured information, cause the need for devising automatic methods in order to extrapolate information from these data. This information is very useful especially for commercial purposes. Jackson and Moulinier [21] write: *there is no question concerning the commercial value of being able to classify documents automatically by content. There are myriad potential applications of such a capability for corporate Intranets, government departments, and Internet publishers.* One of the main fields in which the classification of texts is exploited is the Sentiment Analysis one. Sentiment Analysis, or Opinion Mining, consists in analyzing the opinions expressed by users, for example on social media, regarding products and services.

To better understand the potential that Opinion Mining systems have on the global ICT market, Forrester Research, an independent organization that deals with market research, reports the Opinion Mining as emerging field, while Gartner Inc., which operates in strategic consulting, defines social media analytics as one of the most promising technologies by estimating its greatest impact in the next three years [17]. When it was born, at the beginning of the new millennium, sentiment analysis was conceived as a research area addressing documents written in only one language. Because of the lack of multimedia social networks which were limited, at that time, to Friendster (2002), MySpace, LinkedIn and Hi5 (2003), Flickr and Facebook (2004), and the hardness of managing multilingual objects, it is no surprise that the seminal works by Turney [46] and Pang et al. [36] published in 2002 had monolingual textual documents as their sole target.
The problem of multilingualism was addressed starting from 2007 [1, 12, 31], but it is still a significant problem because each language has its own peculiar features, making the automatic management of multilingualism still an open issue.

The use of ontologies to classify texts appears to be a good alternative to the standard machine learning approaches, in the case in which a training set of documents is not available or maybe too small to properly perform the classification step, especially in the multilingual context. Given the growing importance of ontologies, our approach to multilingual text classification is ontology-driven and in particular, in the Sentiment Analysis domain, we use a particular kind of sentiment ontology in order to perform the document polarity computation.

In this thesis we describe the architecture, implementation and experiments of a Multilingual Ontology Driven Text Classifier (MOoD-TC) for classifying multilingual textual documents according to classes described in a domain ontology. The main goal of the thesis is to provide a modular and flexible framework for addressing such a multilingual text classification problem. This goal is achieved by providing a set of core modules offering functionalities which are common to any text classification problem (preprocessing, tagging, classification algorithm) plus a customizable structure for those modules which can be implemented by the developer in order to offer application-specific functionalities. In order to demonstrate the potential of our approach, we implemented one of such application-domain modules in the domain of sentiment analysis and we run experiments on it.

Our purpose is then to provide a system able to be customized by the user to obtain different behaviours, always connected to the process of classification; for example our MOoD-TC can be used for several purposes (like in the Sentiment Analysis domain), implementing additional functionalities always exploiting the potential of the multilingualism. To reach this purpose, we create a modular system that combines the core of the program (the text classifier TC) with external application domain modules (ADM), created ad hoc by the user and that can be integrated with the core system in a seamless way, provided that they respect the expected interfaces.

Figure 1.1 shows the architectural schema of MOoD-TC. ADM specializes the text classifier task by implementing functionalities for pre- and post-processing a multilingual textual document. If an ADM is used, the entire system specializes
its behaviour in the domain represented by that particular ADM (e.g., from text classifier to sentiment analyzer). In our system, TC can work alone, but an ADM is meant to work in close connection with the core system.

The core modules are implemented in order to work for the European languages (which share some common features like, for example, the relationship between noun and adjective), but it could be extended to cope with the peculiar features of other languages; in fact, thanks to the modularity of the system, it is possible to integrate different algorithms created specifically to handle that peculiarities, without modifying the entire system.

We implemented the MOoD-TC system in Java, exploiting external libraries for language identification, Part-Of-Speech tagging, multilingual translation and ontologies handling.
CHAPTER 2

BACKGROUND

In this chapter we provide the background on the three research topics that mainly characterize our thesis: ontologies, text classification - with the related issue of word-sense disambiguation -, and Sentiment Analysis.

2.1 ONTOLOGIES

An ontology is defined as “an explicit specification of a conceptualization” [19]. An ontology created for a given domain includes a set of concepts as well as relationships connecting them within the domain. Collectively, the concepts and the relationships form a foundation for reasoning about the domain.

In the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). In particular, classes are organized into hierarchies and define the types of attributes common to individual objects within the class. Moreover, classes are interconnected by relationships, indicating their semantic interdependence. Class hierarchies and class relationships form the schema level of the ontology, while the individuals (object instances or just instances) and links among them (relationship instances) form the so called ground level of the ontology.

There are several standard languages for specifying ontologies. The Resource
Description Framework (RDF)\textsuperscript{1}, proposed by World Wide Web Consortium\textsuperscript{2}(W3C), is the most popular one together with its related extensions: RDFS (RDF Schema)\textsuperscript{3} and the Web Ontology Language \textsuperscript{4}(OWL). In particular OWL, in accordance with the W3C definition, is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things. OWL is a computational logic-based language such that knowledge expressed in OWL can be exploited by computer programs, e.g., to verify the consistency of that knowledge or to make implicit knowledge explicit. OWL documents, known as ontologies, can be published in the World Wide Web and may refer to or be referred from other OWL ontologies.

An ontology can be considered as a representation vocabulary, often specialized to some domain or subject matter. More precisely, it is not the vocabulary as such that qualifies an ontology, but the conceptualizations that the terms in the vocabulary are intended to capture. Thus, translating the terms in an ontology from one language to another, for example from English to Italian, does not change the intended meaning of the ontology from a conceptual point of view. Moreover, the concepts hierarchy that can be represented with ontologies, carries a lot of interesting information about the modeled domain. For example Figure 2.1 shows two different ontologies that represent the same domain (the wines one), but with different structures. These ontologies are created with Protégé\textsuperscript{5}, a free open source editor and framework for building ontologies and intelligent systems.

Despite the fact that the two ontologies have the same concepts, the different relationships between classes make also a different level of granularity. We can see how, in the ontology on the left, we can capture much more information then in the right one. For example if we focus on the Chablis class, we understand from the subclasses relation that it is a white wine made in the Burgundy region; instead on the right ontology we can only understand that it is a wine, without further details.

By analyzing ontologies features, it is possible to notice several similarities with the databases. Again in [19], Tom Gruber explains the differences between them; these differences are summarized as follows (\textit{O}:Ontologies, \textit{D}:Databases):

\begin{itemize}
  \item \textit{O} does not contain explicit information about the data's structure.
  \item \textit{D} has no built-in logic or semantics.
  \item \textit{O} has a defined semantics.
  \item \textit{D} is a data model.
  \item \textit{O} is a knowledge model.
  \item \textit{O} is a declarative language.
  \item \textit{D} is a procedural language.
  \item \textit{O} is a language with a built-in inference mechanism.
  \item \textit{D} is a language without any built-in inference mechanism.
\end{itemize}

\begin{enumerate}
\item \url{http://www.w3.org/RDF/}
\item \url{http://www.w3.org/}
\item \url{http://www.w3.org/TR/rdf-schema/}
\item \url{http://www.w3.org/2001/sw/wiki/OWL}
\item \url{http://protege.stanford.edu/}
\end{enumerate}
2.1 Ontologies

Figure 2.1: Categorizing wines. Having all the wines and types of wine versus having several levels of categorization.

- **Purpose**
  - \( O \): sharing, reuse, defining semantics.
  - \( D \): storing large amounts of structured data.

- **Expressive power of modelling languages**
  - \( O \): close first order logic (FOL); description logics (DL)[28].
– $D$: not as powerful as FOL.

• **Standards for making queries**

  – $O$: no standards, few proposals, for example SPARQL $^6$.
  – $D$: consolidated standards, for example SQL $^7$.

• **Management systems**

  – $O$: Protégé, Jena $^8$ (not commercial).
  – $D$: Many solid, commercial tools and environments (Oracle, etc).

There are several reasons to develop an ontology, related to the importance of having a common vocabulary to share information in a specific domain using a unique know “language”. Some of these reasons are:

• To share a common knowledge about information structure, among people and agents.

• To reuse existing domain knowledge in order to introduce standard languages and services for boosting interoperability.

• To provide reasoning services on a domain for knowledge evolution and understanding.

For example, let us consider several web sites that provide medical information; if these sites share the same domain represented by a unique medical ontology of terms, the computer agents can extract and elaborate information from these sites in a standard way because they speak the same “language”. Often, developing an ontology of the domain is not a goal in itself. Developing an ontology is akin to defining a set of data and their structure for other programs to use. Problem-solving methods, domain-independent applications, and software agents use ontologies and knowledge bases built from ontologies as data. Therefore, it is possible to identify different types of ontologies according to their application context (Figure 2.2):

$^6$http://www.w3.org/TR/sparql11-overview/
$^7$http://en.wikipedia.org/wiki/SQL
$^8$http://jena.apache.org/
• *Top-level or Upper or Foundational ontologies*: ontologies which describe very general concepts that are the same across all domains.

• *Domain ontologies*: contain the vocabulary of a specific domain (i.e. medicine, physics).

• *Task ontologies*: relative to specific tasks or activities (i.e. diagnostics, selling).

• *Application ontologies*: usually a specialization of both Domain and Task ontologies.

![Taxonomy of Ontologies](image)

Figure 2.2: A taxonomy of ontologies based on their purpose (from [20]).

In this time, comprehensive ontologies are available for numerous domains. As of today, several widely adopted ontologies have been created in the area of biology [2], medicine [13] and culture [47].

Another application domain where ontologies show all their potential, as discussed in our work, is the text classification one, discussed in the next section.

### 2.2 Text Classification

In recent years the large amount of electronic data made available from a variety of sources, which include unstructured and semi-structured information, raises the need to devise automatic methods in order to extrapolate information from these
data. In this context, the text mining studies are gaining more and more importance. The main goal of text mining is to enable users to extract information from textual resources and deals with operations like retrieval, classification (supervised, unsupervised and semi-supervised) and summarization. Natural Language Processing, Data Mining, and Machine Learning techniques work together to automatically classify and discover patterns from the different types of documents [43].

Text Classification (TC), is an important part of text mining. It can be defined as the task of assigning a document to one or more classes or categories. The automatic classification of documents is an example of how Machine Learning (ML) and Natural Language Processing (NLP) can be leveraged to enable machines to better understand human language. For instance, we might want to classify a text w.r.t. a set of geographical, historical, and topic classes (e.g., understanding whether a text is about the neolithic rock art in France, like in the IndianaMAS project [29]). Again, we would automatically label each incoming news document with a topic like “sports”, “politics”, or “art”. There are many reasons that justify the importance of text classification. Some examples of domains in which text classification is commonly used are:

- **Spam filtering**: it is often desirable to classify email [11] [9] [26] in order to determine if an email is spam [41] or a legitimate one, in an automated way.

- **Email routing**: routing an email sent to a general address to a specific address or mailbox depending on topic (i.e., work, friend and so on).

- **News filtering and organization**: Most of the news services today are electronic in nature in which a large volume is created very single day by the organizations. In such cases, it is difficult to organize the news articles manually. Therefore, automated methods can be very useful for news categorization in a variety of web portals [24].

- **Language identification**: Automatically detecting the language(s) present in a document based on the content of the document.

- **Readability assessment**: Automatically determining the degree of readability of a text, either to find suitable materials for different age groups or reader types or as part of a larger text simplification system.
• **Opinion Mining**: Customer reviews or opinions are often short text documents which can be mined to determine useful information from the review. In particular the Opinion Mining process (also known as Sentiment Analysis) consists in determining the attitude of a speaker or a writer with respect to some topics or the overall contextual polarity of a document. Providing tools for ontology-driven text classification in the context of Sentiment Analysis is one of the goals of our thesis and will be discussed in detail throughout this document.

• **Opinion spam filtering**: Automatically detecting deceptive opinions. Nowadays a large number of opinion reviews are posted on the Web. Such reviews are a very important source of information for customers and companies. The former rely more than ever on online reviews to make their purchase decisions, and the latter to respond promptly to their clients’ expectations. Unfortunately, due to the business that is behind, there is an increasing number of deceptive opinions, that is, fictitious opinions that have been deliberately written to sound authentic, in order to deceive the consumers promoting a low quality product (positive deceptive opinions) or criticizing a potentially good quality one (negative deceptive opinions) [18]

Analyzing these contexts, the need for a system for automatic classification becomes clear; in fact manually categorizing and grouping text sources can be extremely laborious and time-consuming, especially for publishers, news sites, blogs and for all those organizations and persons who must manage and organize a large volume of textual documents. In the literature there are two principal approaches for automatic texts classification:

• **Machine Learning approach**: the most common used approach; it is based on standard *ML* techniques in order to classify a text with respect to a set of documents previously labelled (*training set*).

• **Knowledge based approach**: it is mainly based on *NLP* techniques in order to classify a document using the semantic knowledge like, for example, the semantic relationships among the words (e.g., synonym, antonym, etc.). Normally this approach uses ontologies to represent the knowledge model, as presented in our work.
CHAPTER 2: BACKGROUND

MACHINE LEARNING TEXT CLASSIFICATION. A Machine Learning text classification task starts with a training set $D = (d_1 \ldots d_n)$ of documents that are already labelled with a class identifier $C$ (e.g. sport, politics); the task is then to determine a classification model which is able to assign the correct class to a new document $d$. Moreover the classification can be single label or multi-label; single label document belongs to only one class and multi-label document may belongs to more than one classes. To better understand how a ML text classification works, we briefly analyze its main stages (Figure 2.3).

![Figure 2.3: Document classification process.](image)

- **Documents Collection**: This is the first step of classification process in which we are collecting the different types (format) of document like html, .pdf, .doc, web content etc.

- **Preprocessing**: In this phase the document is prepared for the next text classification step. This preparation is represented by a great amount of features. Usually in this phase there are standard actions performed on the document:
  
  - **Tokenization**: A document is treated as a string, and then partitioned into a list of tokens.
  
  - **Removing stop words**: stop words such as “the”, “a”, “and”, that occur frequently in the document are removed because normally they are not important for the purposes of text classification.

  - **Stemming word**: applying the stemming algorithm that converts words having the same root but different forms (for example, different suf-
fixes due to number and gender) into the same canonical form (a.k.a. lemmatisation).

- **Indexing and features selection**: The main idea is to reduce the complexity of the documents and make them easier to handle; the document has to be transformed from the full text version to a document vector and, after that, it is performed the Feature Selection ($FS$) [10] that is the selection of subset of features from the original documents. $FS$ is performed by keeping the words with highest score according to predetermined measure of the importance of the word.

- **Classification**: This is the real heart of the $TC$ task. The document prepared in the previous steps can be classified by three ways, unsupervised, supervised and semi supervised methods. Some techniques are described below.

- **Performance Evaluations**: This is the last step of $TC$, in which the evaluation of text classifiers is typically conducted experimentally, rather than analytically. The experimental evaluation of classifiers, rather than concentrating on issues of efficiency, usually tries to evaluate the effectiveness of a classifier, i.e. its capability of taking the right categorization decisions. Many measures have been used, like precision and recall [51], fallout, error, accuracy etc.

In supervised Machine Learning methods, a model is created based on previous observations, i.e., a training set. In the case of document classification, categories are predefined and a training dataset of documents is manually tagged as part of a category. Following the creation of a training dataset, a classifier is trained on the manually tagged dataset. The idea behind this approach is that, the classifier will then be able to predict any given document’s category from then on. Therefore, designing classification methods that effectively account for these characteristics of text is of paramount importance. Some widely adopted methods, which are commonly used for text classification are:

- **Decision Trees**: Decision trees are designed with the use of a hierarchical division of the underlying data space with the use of different text features [40]. The hierarchical division of the data space is designed in order to create
class partitions which are more skewed in terms of their class distribution. For a given text instance, we determine the partition that it is most likely to belong to, and use it for the purposes of classification.

- **Pattern (Rule)-based Classifiers**: In rule-based classifiers we determine the word patterns which are most likely to be related to the different classes. We construct a set of rules, in which the left-hand side corresponds to a word pattern, and the right-hand side corresponds to a class label. These rules are used for the purposes of classification [5].

- **SVM Classifiers**: Support Vector Machines (SVM) Classifiers attempt to partition the data space with the use of linear or non-linear delineations between the different classes. The key in such classifiers is to determine the optimal boundaries between the different classes and use them for the purpose of classification [22].

- **Neural Network Classifiers**: Neural networks are used in a wide variety of domains for the purpose of classification. We note that neural network classifiers are related to SVM classifiers; indeed, both of them are in the category of discriminative classifiers, which are in contrast with the generative classifiers [35].

- **Bayesian (Generative) Classifiers**: In Bayesian classifiers (also called generative classifiers), we attempt to build a probabilistic classifier based on modeling the underlying word features in different classes [30]. The idea is then to classify text based on the posterior probability of the documents belonging to the different classes on the basis of the word presence in the documents.

- **Other Classifiers**: Almost all classifiers can be adapted to the case of text data. Some of the other classifiers include nearest neighbor classifiers [30], and genetic algorithm-based classifiers.

In order to use this supervised ML algorithm, we need for a training set; however, it is often the case that a suitable set of well categorized (typically by humans) training documents is not available. Even if one is available, the set may be too small, or a significant portion of the documents in the training set may not have
been classified properly. This creates a serious limitation for the usefulness of the traditional text categorization methods.

**Knowledge-Based Text Classification.** This text categorization method is based only on leveraging the existing knowledge represented in a domain ontology. The positive aspect of this approach is that it is not dependent on the existence of a training set, as it relies solely on the entities, their relationships, and the taxonomy of categories represented, for example, in a ontology, that effectively becomes the classifier. Consequently, training with a set of pre-classified documents is not needed, as the ontology already includes all important facts. So the knowledge represented in such a comprehensive ontology can be used to identify topics (concepts) in a text document, provided the document thematically belongs to the domain represented in the ontology. Furthermore, if the concepts in the ontology are organized into hierarchies of higher-level categories, it should be possible to identify the category (or more categories) that best classify the content of the document.

As an example of ontology text classification, let us assume that we have a well-defined and comprehensive ontology containing knowledge about the smartphone domain. The ontology includes a wide variety of concepts about smartphone features, such as brands, display, type of connection technology and so on, organized into a hierarchy structure. Now, let us consider an article, maybe a review, describing a new smartphone:

"This phone is just perfect! Good screen and battery life, the front camera is 3.0 mpx and design is simply perfect. It support also 4g connection!"

Within this document we will be able to identify a large number of concepts present in our ontology (the bold words). As mentioned previously, this approach does not classify the document with respect to a set of classes, but it is able to classify it in respect of those categories represented by the ontology. One shortcoming of this approach is the presence of false positives; in fact we could classify texts in a wrong way because of the presence, inside the ontology, of concepts less specific to its domain, (i.e., they can be present in more contexts).

Let us consider multilingual text classification (i.e., documents in multiple languages). If we want to classify a new document in a different language, in the
In the case of supervised methods, we need a new training set for that specific language, and so on for each new document in different language that we want to classify. Several solutions have been proposed to overcome this problem. We might think, for example, to translate documents in the language \( L_T \) of the training set. This approach has a number of shortcomings, for example:

- lexical variety in \( L_T \) (e.g., English: huge vocabulary, many synonyms).
- variety of expression in source language.
- lexical ambiguity in \( L_T \) (unnecessary introduction of additional ambiguity).

These three points are well represented in the following example (Table 2.1).

The word “auto” could be translated to “car”, while “voiture” into “automobile”. These different translations, depending on the language, get variety and ambiguity in the classification; in our work we will try to handle these multilingual problems using ontologies.

### 2.3 Word Sense Disambiguation

In Natural Language Processing, word sense disambiguation (WSD) is the problem of determining which "sense" (meaning) of a word is activated by the use of the word in a particular context, a process which appears to be largely unconscious in people. WSD is a natural classification problem: Given a word and its possible senses, as defined by a dictionary, classify an occurrence of the word in context into one or more of its sense classes. The features of the context (such as neighboring words) provide the evidence for classification [14].

For example, consider two examples of the distinct senses that exist for the (written) word **bass**:

<table>
<thead>
<tr>
<th>Language</th>
<th>English machine translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italian</td>
<td>auto ( \rightarrow ) car</td>
</tr>
<tr>
<td>French</td>
<td>voiturer ( \rightarrow ) automobile</td>
</tr>
</tbody>
</table>

Table 2.1: Example of machine translation: lexical variety and ambiguity.
1. A type of fish.

2. Tones of low frequency.

and the sentences:

1. I went fishing for some **sea bass**.

2. The **bass** line of the song is too weak.

To a human being, it is obvious that the first sentence is using the word **bass**, as in the former sense above and in the second sentence, the word **bass** is being used as in the musical sense below. Developing algorithms to replicate this human ability can often be a difficult task.

Sense disambiguation is often characterized as an intermediate task, which is not an end in itself, but essential for many applications requiring broad-coverage language understanding [33]. Examples include machine translation [48] and information retrieval [44]:

- **Machine translation**: WSD is required for lexical choice in machine translation (MT) for words that have different translations for different senses. For example, in an English-French financial news translator, the English noun *change* could translate to either *changement* (“transformation”) or *monnaie* (“pocket money”). However, most translation systems do not use a separate WSD module. The lexicon is often pre-disambiguated for a given domain, or hand-crafted rules are devised, or WSD is folded into a statistical translation model, where words are translated within phrases which thereby provide context.

- **Information retrieval**: ambiguity has to be resolved in some queries. For instance, given the query “depression” should the system return documents about illness, weather systems, or economics? Current information retrieval systems (such as Web search engines), like MT, do not use a WSD module; they rely on the user typing enough context in the query to only retrieve documents relevant to the intended sense (e.g., “tropical depression”). In a process called mutual disambiguation, all the ambiguous words are disambiguated by virtue of the intended senses co-occurring in the same document.
There are four conventional approaches to WSD:

- **Dictionary- and knowledge-based methods**: these rely primarily on dictionaries, thesauri, and lexical knowledge bases, without using any corpus evidence. The Lesk method [25] is the seminal dictionary-based method. It is based on the hypothesis that words used together in text are related to each other and that the relation can be observed in the definitions of the words and their senses. Two (or more) words are disambiguated by finding the pair of dictionary senses with the greatest word overlap in their dictionary definitions. For example, when disambiguating the words in pine cone, the definitions of the appropriate senses both include the words evergreen and tree (at least in one dictionary).

- **Supervised methods**: these make use of sense-annotated corpora to train from. Supervised methods are based on the assumption that the context can provide enough evidence on its own to disambiguate words (hence, world knowledge and reasoning are deemed unnecessary).

- **Semi-supervised or minimally-supervised methods**: these make use of a secondary source of knowledge such as a small annotated corpus as seed data in a bootstrapping process, or a word-aligned bilingual corpus. The bootstrapping approach starts from a small amount of seed data for each word: either manually-tagged training examples or a small number of surefire decision rules (e.g., play in the context of bass almost always indicates the musical instrument). The seeds are used to train an initial classifier, using any supervised method. This classifier is then used on the untagged portion of the corpus to extract a larger training set, in which only the most confident classifications are included. The process repeats, each new classifier being trained on a successively larger training corpus, until the whole corpus is consumed, or until a given maximum number of iterations is reached.

- **Unsupervised methods**: these eschew (almost) completely external information and work directly from raw unannotated corpora. These methods are also known under the name of word sense discrimination. Unsupervised learning is the greatest challenge for WSD researchers. The underlying assumption is that similar senses occur in similar contexts, and thus senses can be induced
from text by clustering word occurrences using some measure of similarity of context. Then, new occurrences of the word can be classified into the closest induced clusters/senses. Performance has been lower than other methods, above, but comparisons are difficult since senses induced must be mapped to a known dictionary of word senses.

2.4 Sentiment Analysis

Sentiment Analysis (SA) is the process of identifying the orientation of opinions in a piece of text. When we talk about sentiment we refer to feeling, like attitudes, emotions or opinions; these are subjective impressions, not facts. Normally, a binary opposition in opinions is assumed: for/against, like/dislike, good/bad, etc. Generally speaking, SA aims to determine the attitude of a speaker or a writer with respect to some topic or the overall contextual polarity of a document [27] (Figure 2.4).

Sentiment Analysis is also known with other names:

- Opinion mining
- Sentiment mining
- Subjectivity analysis

Recognizing sentiments is a very natural ability of a human being; the aim of a sentiment analyzer is to find the best way to simulate this ability automatically.

There are many application fields characterized by a strong need of automated techniques to extract polarity information from documents, and where SA may provide a valuable support.

The main motivations are:
• **Consumer information, for example:**
  
  – Is this movie review positive or negative?
  
  – Is this customer email satisfied or dissatisfied?

• **Marketing, for example:**
  
  – What do people think about the new smartphone?
  
  – Trends.

• **Politics, for example:**
  
  – What do people think about this candidate or issue?
  
  – Politicians want to know voters’ views.
  
  – Voters want to know politicians’ stances and who else supports them.

Since SA comes from text classification, also in this case there are two main techniques: Machine Learning and NLP using ontologies. Even without going into the details of these techniques, it is clear how this specific domain introduces a number of additional issues to the classification task. First of all, it is necessary to identify which are the relevant features for the polarity identification, because inside a document there are different scopes of sentiment analysis (Figure 2.6).
Figure 2.6: Different scopes of Sentiment Analysis inside a document.

<table>
<thead>
<tr>
<th>WORD</th>
<th>POLARITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>great</td>
<td>POSITIVE</td>
</tr>
<tr>
<td>interesting</td>
<td>POSITIVE</td>
</tr>
<tr>
<td>dud</td>
<td>NEGATIVE</td>
</tr>
</tbody>
</table>

**DOCUMENT POLARITY:** POSITIVE

Table 2.2: Example of a general sentiment analyzer output.

People express opinions in complex ways and, in opinion texts, lexical content alone can be misleading. Let consider this example:

"His last movie was **great** and **interesting**. This one's a **dud**."

By analyzing this sentence, we know that it has a negative polarity (this film, unlike the previous one of the same director, is bad), but if we consider all the adjectives of that sentence that express feelings, a sentiment analyzer might generate the wrong answer (Table 2.2).

It is therefore clear how polarity identification is not a trivial task to be performed in an automated way. From the simplest task of SA (binary classification - positive or negative) to complex (rank the attitude of a text from 1 to 5) we need to consider other features due to:
• Subtlety expression of sentiment.
  – irony and sarcasm.
  – expression of sentiment using neutral words.

• Domain/context dependence: words/phrases can mean different things in different contexts and domains (Word Sense Disambiguation).

• Presence of superlatives and other constructs that emphasize the polarity of the sentence.

• Presence of negations that reverse the polarity of the sentence (e.g., NOT good → bad).
CHAPTER 3

STATE OF THE ART

In this section we review the state of the art on multilingual sentiment analysis and ontology driven sentiment analysis, aspects that characterize the contribution of our sentiment analysis ADM.

Multilingual sentiment analysis. In [31], Mihalcea et al. explore methods for generating subjectivity analysis – namely identifying when a private state is being expressed and identifying attributes of that private state including who is expressing the private state, the type(s) of attitude being expressed, about whom or what the private state is being expressed, the intensity of the private state, etc. [50] – in a target language $L$ by exploiting tools and resources available in English. Given a bilingual dictionary or a parallel corpus acting as a bridge between English and the selected target language $L$, the methods can be used to create tools for subjectivity analysis in $L$. Experiments are carried out with Romanian. Ahmad et al. [1] classify sentiments within a multilingual framework (English, Arabic, and Chinese) following a local grammar approach. Domain-specific keywords are selected by comparing the distribution of words in a domain-specific document to the distribution of words in a general language corpus. Words less prolific in a general language corpus are considered to be keywords. Sebastiani et al. [15] introduce a methodology based on lexical resources for sentiment analysis available in English (SentiWordNet, http://sentiwordnet.isti.cnr.it/) for determining polarity of text within a multilingual framework. The method is tested for German movie reviews selected from Amazon and is compared to a statistical polarity classifier based on n-grams. The paper by Boiy and Moens [6] describes machine learning experiments with regard to sentiment analysis in blog, review and forum texts found on the World
Wide Web and written in English, Dutch and French. The proposed approach combines methods from information retrieval, natural language processing and machine learning. An automated sentiment analysis on multilingual user generated contents from various social media and e-mails is described in [45]. The sentiment analysis is based on a four-step approach including language identification for short texts, part-of-speech tagging, subjectivity detection and polarity detection techniques. The prototype has been tested on English and Dutch. More recently, the paper [3] presents an evaluation of the use of machine translation to obtain and employ data for training multilingual sentiment classifiers. The authors demonstrate that the use of multilingual data, including that obtained through machine translation, leads to improved results in sentiment classification and that the performance of the sentiment classifiers built on machine translated data can be improved using original data from the target language. The languages explored by the authors are Turkish, Italian, Spanish, German and French. Finally, the paper [16] describes the adoption of meta-learning techniques to combine and enrich existing approaches to single and cross-domain polarity classification based on bag of words, n-grams or lexical resources, adding also other knowledge-based features. The proposed system uses the BabelNet multilingual semantic network [34] to generate word sense disambiguation and vocabulary expansion-derived features. Being based on BabelNet, the system can cope with multilingual documents. By now its evaluation has been carried out on a monolingual dataset, the Multi-Domain Sentiment Dataset (version 2.0, http://www.cs.jhu.edu/~mdredze/datasets/sentiment/). Evaluating the polarity classification approach in other languages is part of the authors’ future work.

Ontology driven sentiment analysis. One of the first papers on ontology-based sentiment classification is [39], where the ontology was used to classify and analyze online product reviews by providing lexical variations and synonyms of terms that could be met in the reviews. We implement and experiment our assumption with Support Vector Machine based on the lexical variation ontology. This work applied text classifier as the sentiment classifier. Then the sentiment classifier building and testing utilizes the lexical variations and synonyms in the ontology. In this way, these have identical weights. For instance, suppose word-1 is a synonym or variation of word-2, and that the weight of word-2 has been calculated. The weight of word-1 can be obtained from the weight of word-2. Thus, although
the online product reviews are written using different words which may share the same meaning, text classifiers can still be analyzed. In [7], Chaves and Trojahn present Hontology, a multilingual ontology for the hotel domain. Hontology has been proposed in the context of a framework for ontology-driven mining of Social Web sites contents. Comments are annotated with concepts of Hontology, which are manually labeled in Portuguese, Spanish and French. Hontology reuses concepts of other vocabularies such as Dbpedia.org and Schema.org. This approach facilitates the task of comments mining, helping managers in their decision-making process. The work on Hontology was further expanded in [8]. Affective computing is receiving increasing attention in many sectors, ranging from advertisement to politics. This work, set in a Social Semantic Web framework, presents ArsEmotica [4] is a software application for associating the predominant emotions with artistic resources of a social tagging platform. A rich emotional semantics (i.e., not limited to a positive or a negative opinion) is extracted from tagged resources through an ontology driven approach. The ArsEmotica Ontology (AEO [37]) is based on Plutchik’s model [38] and incorporates, in a unifying model, multiple ontologies which describe different aspects of the connections between media objects (e.g., the ArsMeteo artworks, http://www.arsmeteo.org/), persons and emotions. In particular, it includes an ontology of emotions which have been linked, via owl:sameAs, to the corresponding emotions in DBpedia. Furthermore, it incorporates an ontology of artifacts, derived from the alignment of a domain ontology obtained from the DB of the ArsMeteo on line portal, with the OMR (Ontology for Media Resources, http://www.w3.org/TR/mediaont-10/). The paper [23] proposes the deployment of original ontology-based techniques towards a more efficient sentiment analysis of Twitter posts. The novelty of the proposed approach is that posts are not simply characterized by a sentiment score, as is the case with machine learning-based classifiers, but instead receive a sentiment grade for each distinct notion in the post. The proposed architecture aims at providing a more detailed analysis of post opinions regarding a specific topic.

Comparison. With respect to the existing literature on multilingual sentiment analysis, our work is among the few ones that perform an evaluation involving five languages, hence demonstrating the actual multilingualism and flexibility of the approach. As far as the adopted tools are concerned, the work closer to our is [16] for the heavy exploitation of BabelNet.
CHAPTER 4

TOOLS USED

Before going into the detail of MOoD-TC architecture and functioning, we provide some background on the tools that we used in our system. In particular we focus on BabelNet, the main tool used by TC.

4.1 BABELNET

BabelNet \(^1\) [34] is a very large multilingual semantic network, based on the automatic mapping of concepts onto WordNet and Wikipedia\(^2\), the largest multilingual Web encyclopedia. The result is an “encyclopedic dictionary”, in which words (Babel Senses) in different languages (BabelNet 3.0 supports 271 languages including all European languages, most Asian languages, and even Latin) are grouped into sets of synonyms called Babel synsets.

![Figure 4.1: BabelNet structure.](http://babelnet.org/)

\(^1\)http://babelnet.org/
\(^2\)https://www.wikipedia.org/
Each *Babel synset* has different features like short definitions (*glosses*) in many languages harvested from both WordNet and Wikipedia, and many relations in the semantic network provided by WordNet (e.g., hypernymy and hyponymy, meronymy and holonymy, antonymy and synonymy, etc.). For example, a *Babel synset* can have the following form:

\{play_{EN}, theaterstück_{DE}, dramma_{IT}, obra_{ES}, \ldots, pièce de théâtre_{FR}\}

where the subscript of each concept represents the label of the corresponding language.

BabelNet offers a web application accessible directly from the web site of the project; like a search engine, the user can search a word obtaining as output all the *Babel Synsets* to which the word belongs (i.e., a *Babel synset* for each possible meaning of that word). The project provides also Java and HTTP APIs, that are freely available from the same site. In order to better understand how BabelNet works, we analyze the resources on which it is based, WordNet and Wikipedia, and how they are integrated together in the system.

### 4.1.1 WordNet

WordNet \(^3\) [32] is the main resource for lexical knowledge upon which BabelNet is based. WordNet groups English words into sets of synonyms (intuitively sets of words of the same meaning) called *synsets*. Obviously a word can have different meanings depending on the context, therefore it may appear in various *synsets*. A label that indicates the part of speech (e.g., *n* means noun, *v* means verb) and the sense number is associated with each word in the *synsets*. Sense numbers are assigned to the words based on their frequency of use in the semantically tagged corpora. Frequency of use is determined by the number of times a sense is tagged in the various semantic concordance texts. To make an example, a *synset* can be of the form:

\{play\(_n^1\), drama\(_n^1\), dramatic_play\(_n^1\)\}

\(^3\)http://wordnet.princeton.edu/
WordNet also provides a textual definition (gloss) for each synset. Moreover, WordNet connects synsets between them with lexical and semantic relationships, making a graph. Typical relations in WordNet are:

- **is-a**: models hierarchical relation between concepts (i.e., a concept is more/less general than an other one).

- **instance-of**: models membership of an entity to a concept.

- **part-of**: models relation between a concept and a larger set of concepts.

Then the sets of this kind of relationships between synsets form the lexical dictionary. The major weakness of WordNet is that it available for English only; BabelNet was born to overcome this limitation, integrating versions of WordNet in other languages.

### 4.1.2 Wikipedia

An other important element of BabelNet is Wikipedia\(^4\), the most famous Web-based multilingual encyclopedia. It is a collaborative open source medium edited by volunteers to provide a very large wide-coverage repository of encyclopaedic knowledge. Each article in Wikipedia is represented by a page (knows as Wikipage) that contains information about a specific concept or entity. Moreover, the title of a Wikipage is composed by a lemma, that defines the concept, with an optional label that specifies its meaning (in the case in which the concept has different meanings, thus ambiguous). In every Wikipedia page we can find different relations between the pages themselves; among the many Wikipedia relationships we can find:

- **Disambiguation pages**: pages that not contain the definition of a specific concept but a list of links to various pages, one for each meaning of the concept if it is ambiguous.

- **Internal links**: relation between a page and one or (normally) more concepts that appear in that page.

- **Interlanguage links**: each concept is associated with the same concept in other languages (i.e., other related Wikipedia pages).

\(^4\)http://www.wikipedia.org/
4.1.3 INTEGRATION

Wikipedia and WordNet can be seen as graphs in which every concept can be connected to the others, using the previous relations. BabelNet development starts from the integration between these two resources. More formally, BabelNet encodes knowledge as a directed graph with labels of the form $G = (V, E)$, where the set of nodes $V$ represents the concepts and entities, instead the set of edges $E$ consists of the WorNet and Wikipedia relationships among the concepts. Each edge is labelled with the kind of semantic relationship between the two considered nodes, or adding a new one in the case of unspecified semantic relationship. Therefore concepts and relationships of Babelnet are imported from WordNet and, thanks to Wikipedia, the semantic sets are built. To construct the BabelNet graph, are grouped:

1. From WordNet: all the concepts and the synsets lexical-semantic links (relations).

2. From Wikipedia: all the encyclopedia voices (concepts) and the relations between the pages via hyperlinks (relations).

At this point the two resources are overlapping in terms of concepts and relationships. A mapping between words of WordNet and Wikipedia pages is performed, as BabelNet is developed as unified resource and it is interested in the intersection of the two resources. The concepts extracted in different languages are also connected with semantic relationships, through interlanguage links provided by Wikipedia. This step ensures to obtain a multilingual and contextualized support, feature that makes BabelNet a very powerful tool.

4.2 TREE TAGGER

TreeTagger $^5$ [42] is a probabilistic language independent part-of-speech (POS in the sequel) tagger. It was developed by Helmut Schmid in the TC project at the Institute for Computational Linguistics of the University of Stuttgart. This tool allows annotation of multilingual texts with POS and lemma information and it has been successfully used to tag texts in 17 different languages. Its major strength is

---

$^5$http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/
that it is adaptable to other languages if a lexicon and a manually tagged training corpus are available. On the other hand, the not standard definition of this training corpus results into a difficult POS output management. For example, Table 4.1 shows a sample output using the English parameter file.

We can see that, to each word of the original text, is associated the corresponding POS and lemma. The lemma is the canonical form, dictionary form, or citation form of a set of words; for example, in many languages, the citation form of a verb is the infinitive or, in English, the citation form of a noun is its singular (e.g., mouse rather than mice). The keyword to identify the POS (e.g., DT for determiner, JJ for adjective), is not standard but depends on personal choices of the parameter file owner. We can see this behaviour in Table 4.2, that shows the same sample output in Italian language, using the corresponding Italian parameter file.

Now DT (determiner) becomes ART (article), and JJ becomes ADJ but they both represent an adjective. An additional problem is the POS granularity, in fact some parameter files are more specific than others; for example the word not is a

Table 4.1: Sample of treetagger output - English parameter file.

<table>
<thead>
<tr>
<th>word</th>
<th>pos</th>
<th>lemma</th>
</tr>
</thead>
<tbody>
<tr>
<td>The</td>
<td>DT</td>
<td>the</td>
</tr>
<tr>
<td>TreeTagger</td>
<td>NP</td>
<td>TreeTagger</td>
</tr>
<tr>
<td>is</td>
<td>VBZ</td>
<td>be</td>
</tr>
<tr>
<td>easy</td>
<td>JJ</td>
<td>easy</td>
</tr>
<tr>
<td>to</td>
<td>TO</td>
<td>to</td>
</tr>
<tr>
<td>use</td>
<td>VB</td>
<td>use</td>
</tr>
</tbody>
</table>

Table 4.2: Sample of treetagger output - Italian parameter file.

<table>
<thead>
<tr>
<th>word</th>
<th>pos</th>
<th>lemma</th>
</tr>
</thead>
<tbody>
<tr>
<td>il</td>
<td>ART</td>
<td>il</td>
</tr>
<tr>
<td>TreeTagger</td>
<td>NPR</td>
<td>TreeTagger</td>
</tr>
<tr>
<td>è</td>
<td>VER:fin</td>
<td>essere</td>
</tr>
<tr>
<td>facile</td>
<td>ADJ</td>
<td>facile</td>
</tr>
<tr>
<td>da</td>
<td>PRE</td>
<td>da</td>
</tr>
<tr>
<td>usare</td>
<td>VER:infi</td>
<td>usare</td>
</tr>
</tbody>
</table>

35
negative adverb, but the major part of the corpora marks it only as adverb. Since we decided to use TreeTagger for the POS tagging phase of TC, we manage these issues. In particular, we use TreeTagger for Java \(^6\), a Java wrapper around the popular TreeTagger package by Helmut Schmid; it was written in Java 5 with a focus on platform-independence and easy integration into applications.

4.3 JENA

Apache Jena \(^7\) is an Open source Semantic Web framework for Java based on W3C recommendations for RDF and OWL, developed by Brian McBride of HP. It provides a suitable programming environment for RDF, RDFS, OWL, SPARQL and an integrated rule-based inference mechanism.

The two main packages of the API are:

- **com.hp.hpl.jena.rdf.model**: package for creating and manipulating RDF graphs
  - *Model*: RDF Model.
  - *ModelMaker*: ModelMaker contains a collection of named models, methods for creating new models (both named and anonymous) and opening previously-named models, removing models, and accessing a single "default" Model for this Maker.
  - *Literal*: RDF Literal.
  - *Statement*: RDF Statement.

- **com.hp.hpl.jena.ontology**: package that provides a set of abstractions and convenience classes for accessing and manipulating ontologies represented in RDF
  - *OntModel*: an enhanced view of a Jena model that is known to contain ontology data, under a given ontology vocabulary (such as OWL).
  - *OntClass*: interface that represents an ontology node characterising a class description.

\(^6\)http://code.google.com/p/tt4j/
\(^7\)http://jena.apache.org/
* ComplementClass
* IntersectionClass
* UnionClass

- **OntResource**: provides a common super-type for all of the abstractions in this ontology representation package.
- **OntModelSpec**: encapsulates a description of the components of an ontology model, including the storage scheme, reasoner and language profile.
- **ObjectProperty**: interface encapsulating properties whose range values are restricted to individuals (as distinct from datatype valued properties).
- **DatatypeProperties**: interface that encapsulates the class of properties whose range values are datatype values (as distinct from ObjectProperty whose values are individuals).

## 4.4 SentiWordNet

SentiWordnet\(^8\) is a lexical resource for opinion mining. In Section 2.4 we analyzed the need for a way to perform the word polarity computation (i.e., a score that identifies the word as positive, negative or neutral); SentiWordNet was born for this purpose. In SentiWordNet, each WordNet synset \(s\) is associated to three numerical scores \(Obj(s)\), \(Pos(s)\) and \(Neg(s)\), describing how objective, positive, and negative the terms contained in the synset are (Figure 4.2). In particular, \(Pos(s)\) and \(Neg(s)\) assume polarity value between 0 and 1, and the objectivity score \(Obj(s)\) is easily calculated as:

\[
ObjScore = 1 - (PosScore + NegScore)
\]

The method used to develop SentiWordNet is based on the quantitative analysis of the glosses associated to synsets, and on the use of the resulting vectorial term representations for semi-supervised synset classification. The three scores are derived by combining the results produced by a committee of eight ternary classifiers, all characterized by similar accuracy levels but different classification behaviour.

\(^8\)http://sentiwordnet.isti.cnr.it/
Figure 4.2: The graphical representation adopted by SentiWordNet for representing the opinion related properties of a term sense (from [15]).

Table 4.3: Sample of SentiWordNet synset information.

<table>
<thead>
<tr>
<th>POS</th>
<th>ID</th>
<th>PosScore</th>
<th>NegScore</th>
<th>SynsetTerms</th>
<th>Gloss</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>00005205</td>
<td>0.5</td>
<td>0</td>
<td>absolute#1</td>
<td>perfect or complete or pure; &quot;absolute loyalty&quot;; &quot;absolute silence&quot;;</td>
</tr>
</tbody>
</table>

SentiWordNet consists of a singular text file in which is stored all the information of WordNet synsets and their polarity. In particular, for each synset, it is present its POS, its id, the positive and negative polarities and its gloss (Table 4.3).

The pair (POS, ID) uniquely identifies a WordNet synset. The PosScore and NegScore values are the positive and negative scores assigned by SentiWordNet to the synset, while the SynsetTerms column reports the terms, with sense number, belonging, to the synset (separated by spaces).

However we know that a word can belong to several synsets, with different polarities depending on the context. Consider for example the word “terrific”; we find it in three different synsets, with the following polarities (Figure 4.3).

In this case, the estimation of the correct polarity is a non-trivial task. If it is not possible to identify the correct context in which the word refers, the synsets polarity average computation could be a possible solution. This solution is the one proposed
NEUTRAL POLARITY
SYNSET ID: 101513619
very great or intense; "a terrific noise"; "a terrific thunderstorm storm"; "fought a terrific battle".

POSITIVE POLARITY
SYNSET ID: 01676517
extraordinarily good or great; used especially as intensifiers; "a fantastic trip to the Orient"; "the film was fantastic!"; "a howling success"; "a marvelous collection of rare books"; "had a rattling conversation about politics"; "a tremendous achievement".

NEGATIVE POLARITY
SYNSET ID: 00196449
causing extreme terror; "a terrifying wail".

by Petter Tönberg ⁹ which provided a Java class (used in our ADM SentiModule) to approximate the sentiment value of a word. We will see how the integration of SentiWordNet with a sentiment ontology allows us to obtain more precise, and contextualized, results.

Figure 4.3: Different polarities of word "terrific".

---

⁹http://sentiwordnet.isti.cnr.it/code/SentiWordNetDemoCode.java
CHAPTER 5

MOoD-TC

In the introduction, we introduced the two main components of our MOoD-TC:

- Text Classifier (TC)
- Application Domain Module (ADM)

The Text Classifier is the central core of our system. Its purpose is to classify a document \( d \) with respect to an ontology \( o \). The result will be a classification of \( d \) w.r.t. \( o \), together with the languages of \( d \) \( (L_d) \) and \( o \) \( (L_o) \), automatically identified.

Figure 5.1: The Text Classifier component.

The \textit{ClassifierObject} is the object that stores a correctly classified word (and additional information) of the document \( d \) with respect to \( o \). TC returns a list of such objects.
The Application Domain Module is a component that can work only in collaboration with the TC. Its main purpose is to process the TC input and output in order to obtain a new domain oriented tool. An ADM is composed by two sub-components: pre-processing and post-processing.

![Figure 5.2: The ADM structure.](image)

The pre-processing component shown in Figure 5.3 takes as input a text and returns a new processed text.

![Figure 5.3: The preprocessing component of ADM.](image)

The post-processing component shown in Figure 5.4, instead, takes as input the TC output and returns a domain dependent result. Figure 5.5 shows the entire pipeline of the integration process between the Text Classifier and the Application Domain Module. Before going into the details of TC and ADM, we analyze the graphical interface offered by MOoD-TC and the functionalities it offers.
5.1 User Interface

The MOoD-TC GUI allows the user to access the classification functionalities offered by TC, together with the additional functionalities offered by the Application Domain Modules (if one is selected), via a user friendly graphical interface.

There is only one button (Classify) that starts the classification process, and a file selection menu that allows the user to load ontology, text and ADM. In Figure 5.6 we show the GUI, which is subdivided into four areas:

- **Ontology area**: when the ontology is selected by the user via the file selection menu located on the top of the GUI, it is shown in this area using a look and feel similar to that of Protégé (a tree of labels), limited to the ontology classes. The user can navigate into the ontology tree expanding its nodes.

- **Text area**: the text associated with the document $d$, which can be loaded via the file selection menu, is shown in this area.

- **Module area**: when an *Application Domain Module* is loaded via the file selection menu, the list of class names (i.e., the names of those classes
Figure 5.6: MOoD-TC GUI.

- Results area: an area where the results of the classification (or, eventually, the results of the exploitation of an ADM) are shown.

To perform the classification in the correct way, selecting an ontology and text is mandatory, whereas the ADM is optional.
CHAPTER 6

TEXT CLASSIFIER

This section provides the details of the design, features and implementation of the Text Classifier component.

TC, designed and implemented to face a classification task, takes as input:

1. An ontology whose classes model the domain of interest and whose names are expressed in any language from a predefined set.

2. A document containing the text to classify written in any language from the above set.

It returns a classification of the text w.r.t. the ontology taken as input. The classification performed by TC is multilingual and exploits BabelNet and TreeTagger.

Therefore, given an ontology $o$ and a document $d$ to classify, TC identifies the classes in $o$ which $d$ belongs to. For instance, in case of a geographic ontology, TC associates with each document (for example, a tourist guide) the geographical place(s) that it describes.

The strengths of TC are the following:

- It is able to classify documents described in several languages.

- The ontologies used for carrying out the classification task can be expressed in several languages as well.

- The languages used in the documents and in the ontologies can be different.

- There is no need to state in advance the languages of the ontologies and documents, as TC can automatically recognize them.
The documents’ format can be either plain text or pdf.

As shown in Figure 6.1, we can consider the structure of TC as a functional schema in which every component performs a specific action, starting from the initial text to the final output.

More in detail, TC executes 7 main steps to complete the classification task:

1. **Document Loading**: reads the input document $d$ and transforms it in a string representation $S_d$.

2. **Text Extraction**: extracts the Bag-of-Words $BoW_d$ from the string $S_d$.

3. **Ontology Loading**: loads the model of the ontology $o$ in memory ($M_o$) and extracts a Bag-of-Words $BoW_o$ from the ontology concepts.

4. **Language Detection**: detects the language $L_d$ used in $BoW_d$ and the language $L_o$ used in $BoW_o$.

5. **Part-Of-Speech Tagging**: tags the $BoW_d$ using TreeTagger, i.e., assigns the correct POS to each word $w \in BoW_d$.

6. **Text Translation**: translates each word $w \in BoW_d$ into the language of the ontology $L_o$ using BabelNet, obtaining a new Bag-of-Words in the ontology language $BoW_{d(Lo)}$.

7. **Classification**: performs the classification step, i.e., for each $w \in BoW_{d(Lo)}$ checks if $w$ belongs to the ontology $o$ (i.e., if the ontology model $M_o$ contains $w$ or one of its synonyms).

The set of languages supported by TC, $SL$, is the intersection between three sets of languages:

- BabelNet: 271 different languages.
- TreeTagger: 19 different languages.
- Language-Detection library: 53 different languages.
The current version of TC is provided with five TreeTagger corpora for the following languages, that are included in $S_L$: English, French, German, Spanish and Italian.
CHAPTER 6: TEXT CLASSIFIER

The current prototype of TC has been developed in Java, using Eclipse on the Ubuntu 14.04.1 Linux platform.

We analyze every TC component, and more in details each step above; for each of them we provide an informal explanation, a functional description and the implementation details.

6.1 DOCUMENT LOADING

This part is devoted to loading a document $d$ from its file path, selected by the user, and converting it into a string format $S_d$.

![Figure 6.2: The document loading module.](image)

$$S_d = \text{documentLoading(File path of } d)$$

IMPLEMENTATION DETAILS. The document can be provided to TC in two ways:

1. It could be already saved in a local directory (e.g., `/home/user/sample.pdf`) or
2. It could be available online (e.g., `http://site/sample.pdf`).

In the latter case, the file is downloaded in a temporary folder by using the `copyURLToFile(...)` method provided by the `org.apache.commons.io/FileUtils` library. Then, the file is read by using different methods depending by the file type. TC currently supports txt and pdf files.

- **txt**: in this case, TC first recognizes the character encoding used by the file.

This step is fundamental in order to be able to successfully read the various
characters in the file. TC supports several different languages containing
diacritical marks like for instance, accents, tilde (e.g., in Portuguese and
Spanish documents) or cedilla (e.g., in French documents) as well as special
characters used in several languages. To this end, TC makes use of the
JUniversalCharDet library\(^1\), a Java port of the encoding detector library
employed by Mozilla; in addition to the standard Unicode-based character
encodings, this library is also able to find the encoding used for files written in
Chinese, Cyrillic, Greek, Hebrew, Japanese and Korean. In case the encoding
is not supported we assign to the document the encoding ISO_8859_1 (i.e.,
the encoding that refers to the Latin alphabets, consisting of 191 characters
from the Latin script).

- **pdf**: in this case, TC makes use of the Apache PDFBox\(^2\) library, an open
  source Java tool conceived for working with PDF documents and able to
  quickly and accurately extract text from a the PDF documents.

![Figure 6.3: Text area after loading.](image)

In both cases the file is opened and shown into the MOoD-TC text area (Figure
6.3) and its textual content is loaded line by line, building the resulting string \(S_d\).

\(^1\)https://code.google.com/p/juniversalchardet/
\(^2\)https://pdfbox.apache.org/
CHAPTER 6: TEXT CLASSIFIER

6.2 TEXT EXTRACTION

This part is devoted to extracting a Bag-of-words $BoW_d$ (a list of words) contained in document $d$, starting from its string representation $S_d$.

Figure 6.4: Text extraction module.

$$BoW_d = textExtraction(S_d)$$

**IMPLEMENTATION DETAILS.** In order to obtain the Bag-of-Words from the input string, we clean the input string $S_d$ (we substitute all the occurrences of multiple white spaces or non visible characters such as tab and newline with a single white space), then we split it by punctuation marks (e.g., comma, full stop, apostrophe), to separate the single words of $S_d$. Finally, we assign each single word to a list of String $BoW_d$ that is provided to next modules.

6.3 ONTOLOGY LOADING

TC allows the user to load an ontology written in OWL. The purpose of this module is to store the model $M_o$ of the ontology $o$ taken in input, and make a Bag-of-Words $BoW_d$ with the concepts of $o$.

$$M_o, BoW_d = ontologyLoading(File Path of o)$$

**IMPLEMENTATION DETAILS.** To manage the input ontology, we use the Jena tool. In Section 4.3 we introduced Jena, with its two main packages that we use
in our code; the following function consists in an in-memory creation of an empty ontology model:

```java
// Create an in-memory ontology model
private void createOntologyModel(){
    OntDocumentManager mgr = new OntDocumentManager();
    OntModelSpec s = new OntModelSpec(OntModelSpec.OWL_DL_MEM);
    s.setDocumentManager( mgr );
    this.ontModel = ModelFactory.createOntologyModel( s, null );
    //Create a new inputstream starting from ontology file path
    InputStream in = FileManager.get().open(ontologyFile);
    // read the ontology file and store it in the model
    ontModel.read( in, "" );
}
```

Once we create and store the ontology model $M_0$, we exploit Jena again in order to iterate over the ontology classes, starting from the roots, and make the corresponding $BoW_o$ with the class names.

```java
private List<String> extractOntologyConcepts(){
    ArrayList<String> bow = new ArrayList<String>();
    //Ask for an iterator to iterate over
```
Object *OntClass* provides several methods to manage everything about an ontology class, like property, subclasses, superclasses and so on. The original ontology is then rebuilt in our TC keeping the entire hierarchical structure, and shown in the ontology area of MOoD-TC (Figure 6.6).

```java
//the ontology classes starting from the root
ExtendedIterator it = ontModel.listHierarchyRootClasses();
while (it.hasNext()){ //In the model, classes ontology are represented as OntClass jena objects
    OntClass cl = (OntClass) it.next();
    //Add the class name to the bow
    bow.add(cl.getLocalName());
}
return bow;
}
```

Figure 6.6: MOoD-TC ontology area after loading.
Figure 6.6 shows that, after the ontology loading operation, the language of the ontology is automatically identified, in fact a label with the language name (English) appears above it; automatic language identification takes place also during the text loading phase, so we postpone its explanation to the next section.

6.4 LANGUAGE DETECTION

This module is devoted to recognizing the language $L_o$ of the ontology $o$ and also the language $L_d$ of the document $d$. This step is necessary because the modules coming after in the pipeline need to know the language of the resources to perform the multilingual classification. To this end, several different approaches can be employed. In the following, we refer to the identification of the language from a general Bag-of-Words ($BoW$), as both the document and the ontology are turned into a $BoW$ before executing this stage.

A naive approach could be to count the matches of the document’s words with the dictionaries of various languages. Obviously, this approach is computationally unfeasible. An approach that ensures a fast detection of the language is based on the use of Naive Bayes classifiers with character n-gram. TC adopts this approach.

![Diagram](image)

Figure 6.7: Language detection module.

$L_{BoW} = \text{detectLanguage}(\text{Bag-of-Words BoW})$
IMPLEMENTATION DETAILS. TC employs the Language-Detection library\(^3\) that is able to detect, with a precision greater than 99%, 53 languages making use of Naive Bayesian filters. In particular, TC analyzes the Bag-of-Words provided by the previous modules and, depending on its length, calls the language detector library using different profiles. To speed up the language detection, TC avoids to provide the complete text of the document to the language detector, given that, potentially, TC could be required to classify documents long tens or hundreds pages. From our experiments, we noticed that using the first 100 words of the text (e.g., about 500-800 characters) provides very good results in terms of both precision and performance.

```java
private Language languageDetection(ArrayList<String> bow) {
    Detector detector = DetectorFactory.create();
    //Pass to the detector the BoW to analyze
    detector.append(bow.toString());
    //Create a new Language object with the language detected
    return Language.valueOf(detector.detect().toUpperCase());
}
```

Note that the language returned by the `detect()` function, is in ISO 639-1\(^4\) standard format, in which the language is represented by a two letters code (e.g., English is `en`, Italian is `it` etc.). Finally we create a new Language object (i.e., a language enumeration) starting from the detected language; this object contains different information about the language such as the language name (`getName()`), or if the language reads right to left. After our system identifies the language of the Bag-of-Words (of \(o\) or \(d\)) \(L_{BoW}\), it must check that the \(L_{BoW}\) belongs to the set \(S\) of languages supported by TC; if the outcome is positive, the classification can proceed to the next step, otherwise an error message (`unsupported language`), is printed on the console output.

\(^3\)https://code.google.com/p/language-detection/

54
6.5 **PART-OF-SPEECH TAGGING**

Once the language of the document is correctly identified, we can proceed with the part-of-speech tagging of the text. The tagging step consists in determining the *POS* for each token in our *BoW* (i.e., the Bag-of-Words provided by the text extraction module), and storing it with other information required for the classification step, in a list of *TagObject* objects (*LoTO*). In this phase we exploit TreeTagger, in particular the multilingual aspect of this tool.

![Figure 6.8: Part-of-speech tagging module.](image)

LoTO = POStagging(Language $L_d$, Bag-of-Words $BoW_d$)

**IMPLEMENTATION DETAILS.** In this module we use TreeTagger for Java (*tt4j*) that is a Java wrapper around the popular TreeTagger package by Helmut Schmid. In order to load the correct TreeTagger file parameter, we use the information about the language identified in the previous step, in particular its name; for that reason, the TreeTagger model must be stored with this format: `[name of language]-utf8.par`.

---

5http://code.google.com/p/tt4j/
private void treeTaggerConfig(Language lang) {
    // Instantiation of wrapper to manage the POS tagging
    TreeTaggerWrapper<String> tt = new TreeTaggerWrapper<String>();
    // Load the TreeTagger model that corresponds to the
    // language identified "lang".
    tt.setModel(lang.getName() + "-utf8.par");
}

As we discussed in Section 4.2, the non-standard POS representation, different for each TreeTagger corpora, required an additional phase in order to handle an homogeneous configuration. In order to achieve this, we have created the configuration files (one for each language) where the POS labels of the original corpus are converted to standard POS labels, especially those required by BabelNet (i.e., noun n, adjective a, adverb r and verb v). Table 6.1 shows an example of this conversion of an English corpus.

For example, if we focus on adjectives, we can see that the English parameter file provides a good granularity to describe them (e.g., jjr means comparative adjective, jjs the superlative one), but since the other corpora do not offer these distinctions (e.g., the Italian on), and because of the BabelNet standard, we have to normalize them to simple adjectives. Note that not every original POS representation has a conversion; for example, in the English corpus we can find CD (cardinal number), DT (determiner), PP (personal pronoun) and other POS that we are not interested in classifying.

Once we prepared the configuration, we can execute the POS tagging operation.

\[\text{http://www.cis.uni-muenchen.de/schmid/tools/TreeTagger/data/Penn-Treebank-Tagset.pdf}\]
### Table 6.1: Conversion of POS representation - English parameter file.

<table>
<thead>
<tr>
<th>ORIGINAL</th>
<th>STANDARDIZED</th>
<th>POS</th>
</tr>
</thead>
<tbody>
<tr>
<td>fw</td>
<td>n</td>
<td>noun</td>
</tr>
<tr>
<td>nn</td>
<td>n</td>
<td>noun</td>
</tr>
<tr>
<td>nns</td>
<td>n</td>
<td>noun</td>
</tr>
<tr>
<td>jj</td>
<td>a</td>
<td>adjective</td>
</tr>
<tr>
<td>jjr</td>
<td>a</td>
<td>adjective</td>
</tr>
<tr>
<td>jjs</td>
<td>a</td>
<td>adjective</td>
</tr>
<tr>
<td>rb</td>
<td>r</td>
<td>adverb</td>
</tr>
<tr>
<td>rbr</td>
<td>r</td>
<td>adverb</td>
</tr>
<tr>
<td>rbs</td>
<td>r</td>
<td>adverb</td>
</tr>
<tr>
<td>vb</td>
<td>v</td>
<td>verb</td>
</tr>
<tr>
<td>vbd</td>
<td>v</td>
<td>verb</td>
</tr>
<tr>
<td>vbg</td>
<td>v</td>
<td>verb</td>
</tr>
<tr>
<td>vbn</td>
<td>v</td>
<td>verb</td>
</tr>
<tr>
<td>vbp</td>
<td>v</td>
<td>verb</td>
</tr>
<tr>
<td>vbz</td>
<td>v</td>
<td>verb</td>
</tr>
</tbody>
</table>
public ArrayList<TagObject> P0Stagging (ArrayList<String> bow) {
    final ArrayList<TagObject> tagObjs = new ArrayList<TagObject>();
    //Action to perform per word in the set
    tt.setHandler(new TokenHandler<String>() {
        public void tag(String token, String pos, String lemma) {
            //return the standard POS representation from our
            //HashMap of correspondences
            String val = corr.get(pos.toLowerCase());
            //Check if original POS representation has a correspondence
            if (val != null) {
                //Create a new tagObj depending on the POS found
                switch (val) {
                    case "n": tagObjs.add(new TagObject(token, lemma, POS.NOUN)); break;
                    case "a": tagObjs.add(new TagObject(token, lemma, POS.ADJECTIVE)); break;
                    case "r": tagObjs.add(new TagObject(token, lemma, POS.ADVERB)); break;
                    case "v": tagObjs.add(new TagObject(token, lemma, POS.VERB)); break;
                    default: break;
                }
            }
        }
    });
    //Process the Part-of-speech tagging
    tt.process(bow);
    return tagObjs
}

tt4j provides the process function, that requires a collection of strings as argument (i.e., the BoWd); we have to specify the action to do after the tagging
### Table 6.2: Output of process function.

<table>
<thead>
<tr>
<th>word</th>
<th>pos</th>
<th>lemma</th>
</tr>
</thead>
<tbody>
<tr>
<td>The</td>
<td>DT</td>
<td>the</td>
</tr>
<tr>
<td>room</td>
<td>NN</td>
<td>room</td>
</tr>
<tr>
<td>was</td>
<td>VBZ</td>
<td>be</td>
</tr>
<tr>
<td>beautiful</td>
<td>JJ</td>
<td>beautiful</td>
</tr>
</tbody>
</table>

### Table 6.3: Final output of the POS tagging phase - each row represents a `tagObj`.

<table>
<thead>
<tr>
<th>word</th>
<th>lemma</th>
<th>BabelNet POS</th>
</tr>
</thead>
<tbody>
<tr>
<td>room</td>
<td>room</td>
<td>POS.NOUN</td>
</tr>
<tr>
<td>was</td>
<td>be</td>
<td>POS.VERB</td>
</tr>
<tr>
<td>beautiful</td>
<td>beautiful</td>
<td>POS.ADJECTIVE</td>
</tr>
</tbody>
</table>

operation of a word in the collection. In the `tag` function, automatically called for each word by the `process` function, we check if the original extracted `POS` exists in our conversion file; in positive case we apply the conversion and we create a new `tagObject` that contains all the output information about the tagging (i.e., the original word represented by `token`, the lemma and the BabelNet `POS`), otherwise we ignore that particular token.

To summarize this section, we analyze step by step a small example. Let us consider this sentence:

"The room was beautiful."

The input parameter of our `POS` module is the English language (automatically identified), and the Bag-of-Words ("The", "room", "was", "beautiful"). Now each word of the collection is processed by the `process` function (output in Table 6.2).

Finally, for each tagged word we convert the `POS` to the standard format, and we initialize our list of `tagObj` objects (Table 6.3).

As we can expected, the word "the" does not appear in our final result, because
the DT original POS has no correspondences in our conversion file shown in Table 6.1. The lemma identification action is fundamental in order to reach our multilingual classification goal, because BabelNet is able to find and translate correctly only words in lemma form.

6.6 TEXT TRANSLATION AND CLASSIFICATION

The main goal of this module is to translate each word of BoWd into the language Lo used to describe the ontology o (Figure 6.9) and to return each translated word that was found in o, by exploring the in-memory ontology model Mo associated with o (Figure 6.10). These operations are performed together in a unique step because the implementation was easier and more efficient in this way, but formally, in our functional schema, they are two different steps. From the previous POS tagging phase, we have all the information we need to correctly use BabelNet, in particular the POS and the lemma associated with each word and saved into the TagObject object.

Figure 6.9: Text translation module.

Figure 6.10: Classification module.
$BoW_{d(Lo)} = \text{translateText}(BoW_d, L_d, L_o)$
$BoW_{Lo}$ is the Bag-of-Words containing, for each $w \in BoW_d$ in the document language $L_d$, the translation in the ontology language $L_o$.

$LoCO = \text{classification}(M_o, BoW_{Lo})$
$LoCO$ is a list of $ClassifierObject$, objects that contain information about the classification step.

**IMPLEMENTATION DETAILS.** In the sequel we first show the code of the classification step. A detailed explanation follows.

```java
public TCOOutput classification(ArrayList<TagObject> tagObj){
    // Text classifier standard output
    TCOOutput output = new TCOOutput(textLang, ontLang, null);
    // Classification standard output
    ArrayList<ClassifierObject> info = new ArrayList<ClassifierObject>();
    for (TagObject obj : tagObj) {
        found = false;
        // Retired each BabelSynset by text language, lemma and POS of the word
        for (BabelSynset syn : bn.getSynsets(textLang, obj.getLemmaWord(), obj.getPOS())){
            // For each BabelSense obtained from the translation of the Synset in the ontology language
            for (BabelSense sen : syn.getSenses(ontLang)){
                String sense = sen.getLemma().toLowerCase();
                if (!sense.contains("_")) {
                    // Start the iteration over the ontology classes
                    ExtendedIterator it = ontModel.listClasses();
                    while (it.hasNext()){
                        OntClass c = (OntClass) it.next();
                        String name = c.getLocalName();
        ```
ArrayList<String> names = null;
if (name != null)
    //Compound ontology classes
    names = new
        ArrayList<String>(Arrays.asList(
            name.split("__")))
    //Check if there is some correspondences
    if (names != null && names.contains(sense))
    {
        boolean contains = false;
        for (int i = 0; i < info.size() &&
             !contains; i++) {
            ClassifierObject o = info.get(i);
            //If exist a ClassifierObject for
            //the same lemma, add the new token
            //word to it.
            if(o.getLemmaWord().equals(
                obj.getLemmaWord()) &&
                o.getPos().equals(obj.getPOS())){
                o.addTextWord(obj.getTextWord());
                contains = true;
            }
        }
    }
    //Otherwise create a new ClassifierObject
    if(!contains) {
        List<String> textWords = new
            ArrayList<String>();
        textWords.add(obj.getTextWord());
        info.add(new
            ClassifierObject(textWords,
                obj.getLemmaWord(), sense,
                obj.getPOS(),
                getOntologyTree(c)));
        }
    found = true;
    break;
}
The classification function takes the list of TagObject objects (LoTO), resulting from the previous tagging phase as its input. For each $To \in LoTO$, all the synsets in the text language containing the lemma and the POS of the original word $w$ contained in $To$ are retrieved from BabelNet. To do this, we used the BabelNet function getSynsets.

```java
public java.util.List<BabelSynset> getSynsets(it.uniroma1.lcl.jlt.util.Language language,
        java.lang.String word,
        edu.mit.jwi.item.POS pos)
    throws java.io.IOException

Parameters:
    language - the language of the input word.
    word - the word whose senses are to be retrieved.
    pos - the PoS of the word.
```

 Obviously $w$ can appear in more than one synset. For instance, in case of an Italian text containing the word “pulitissima” (clean), the BabelNet function getSynsets is called with the parameters $L_d$=Language.IT, lemma = “pulito” and $POS$ = POS.Adjective, and returns a set of synsets $S$. Indeed, “pulito” in Italian
has different meanings\textsuperscript{7}, including for instance: (1) free from dirt or impurities; or having clean habits\textsuperscript{8}, (2) characterized by freedom from troubling thoughts (especially guilt)\textsuperscript{9}.

Once we obtain the synsets, we have to take all the senses associated with each synset in the language of our ontology. Given $L_o$ the target language used in the ontology, all the words associated with each synset $s \in S$ in the language $L_o$ are retrieved by means of the BabelNet function \textit{getSenses}.

\begin{verbatim}
public java.util.List<BabelSense> 
    getSenses(it.uniroma1.lcl.jlt.util.Language language)

Get the senses contained in this BabelSynset for an input language

Returns:
    the senses of this Babel synset in a specific language.
\end{verbatim}

In the case of the word $w$="pulito" and $L_o=$Language.EN, we obtain several translations including: clear, clean, neat, uncontaminated, orderly, elegantly, untarnished, untainted, unstained, stainless, unsullied (i.e., the lexicalizations in the various languages contained in the synset that express its concept or named entity).

At this point, for each translated sense, we check using Jena if it is present into the ontology model $M_o$ previously stored. Then the ontology is scanned concept by concept and, if a match is found between the sense and the concept, the search stopped. We know that an ontology concept can be compound (e.g., in a smartphone ontology we can find concepts like “battery-life”, “front-camera”, “external-storage” and so on). In order to correctly perform a match with this kind of concept, we implemented a first algorithm variant in which the compound concept is split by its words separator (i.e., underscore, hyphen, etc., ), and after that we treat its components (e.g., “battery” and “life”) as singular concept, with the searching scope limited to the same text sentence. A better algorithm was implemented by Luigi

\footnotesize
\begin{itemize}
    \item \textsuperscript{7}http://BabelNet.org/search?word=pulito&lang=IT
    \item \textsuperscript{8}http://BabelNet.org/synset?word=bn:00099776a&details=1&orig=pulito&lang=IT
    \item \textsuperscript{9}http://BabelNet.org/synset?word=bn:00099807a&details=1&orig=pulito&lang=IT
\end{itemize}

64
Carratino as project of *Intelligence System* course, but it is not actually integrated in TC; Appendix A shows an informal explanation and the pseudo-code of this approach. If a match is found, the first time we proceed with the creation of an object used to save the information about the classification, the *ClassifierObject*.

```java
class ClassifierObject {
    private List<String> textWords;
    private String lemmaWord;
    private String ontologyWord;
    private POS pos;
    private List<String> ontologyTree;
    // ...
    // Constructor, getter and setter
    // ...
}
```

*ClassifierObject* contains the following information:

- **textWords**: list of words found in the text (in the document language $L_d$) that are linked to the same concept of ontology, with the same $POS$.

- **lemmaWord**: the lemma of the word (in the text language $L_d$), which corresponds to the concept found in the ontology. Obviously all previous textWords found have this lemma as their basic form.

- **ontologyWord**: the concept found in the ontology (in the ontology language $L_o$).

- **POS**: the part-of-speech of this lemma.

- **ontologyTree**: list of labels that correspond to the ontology path from the root to the concept found.

*ClassifierObject* contains also the method `getNumberOfOccurrences()` that returns the number of occurrences of that concept found in the text, which is simply the length of the `textWords` list.

So at the first occurrence found, the *ClassifierObject* is created. If in the text we will find other words related to the lemma already found, with the same
POS, we modify the corresponding ClassifierObject inserting the new word in the corresponding textWords list and thereby increasing the number of occurrences found.

Let us make an example to clarify the algorithm; suppose we want to classify the following Italian phrase:

"La camera è molto bella, soprattutto per il bellissimo bagno."

As already mentioned, at this point we have several information for each word correctly tagged in the previous phase. Let us focus on the adjective bella (beautiful), and its superlative bellissimo, and let us assume that our ontology, whose concepts are expressed in English, contains the concept "beautiful". During the BabelNet search phase, among the possible translations of the word bella, there is also beautiful, that corresponds to the concept in our ontology. At this point the search of the word bella ends and, since it is the first time that we find this particular concept, the corresponding ClassifierObject, with the following information, is created:

- textWords = ["bella"]
- lemmaWord = "bello"
- ontologyWord = "beautiful"
- POS = ADJ (adjective)
- ontologyTree = ["positive-opinion","positive-accommodation","positive-room-features","beautiful"]

When the classifier reaches the word "bellissimo", it finds out that a ClassifierObject with the same lemma ("bello") and the same POS (adjective) already exists; the word is then added to the textWords list of the ClassifierObject identified. So the new ClassifierObject has the same fields as the previous one (since the information of classification of words "bella" and bellissima are the same), but the textWords list now contains also the new word:

textWords = ["bella","bellissima"]
The set of \textit{ClassifierObject}, created at each classification step, is stored in a list $R$ that will become the main component of the object for the standard output representation of our TC.

### 6.7 Text Classifier Result

The \textit{TCOutput} is the object that stores all the information about the text classification. In addition to the list of \textit{ClassifierObject} cited above, it also stores the information of the original text language $L_d$ and the language of the ontology $L_o$.

![TCOutput Object Diagram](image)

Figure 6.11: Text classifier output object.

This information is used either to generate a valid result output of the classification or as standard input for ADM, as we will see in Section 7.

```java
public class TCOutput {
    private Language textLang;
    private Language ontLang;
    private List<ClassifierObject> info;
    // ...
    // Constructor, getter and setter
    // ...
}
```

A sample of \textit{TCOutput} object is then shown in the MOoD-TC output area (Figure 6.12).
The information is represented in table form. For each correctly classified word of the text, in respect of the ontology, the MOoD-TC GUI displays its lemma (lemma word), its corresponding concept in the ontology (ontology word) and the number of occurrences of that word in the text (occurrence). Note that some pieces of information, previously saved in ClassifierObject, are not explicitly shown in the table area; in particular, the textWords list and the ontology path (ontologyTree) from that concept to the ontology root are not shown. In order to make the interface more interactive, we decided to provide this information as a result of a mouse click action from the user on the table row corresponding to that particular ClassifierObject (Figure 6.13).

In particular, the row click generates an event that consists in highlighting, in the text area, all the words contained in the textWords list that correspond to the selected ClassifierObject. In addition, the entire path stored in the ontologyTree list is used to automatically expand the various ontology nodes within the MOoD-TC ontology area, by obtaining an immediate visual result of the path from that concept to the root. Figure 6.13 shows how the beautiful ontology concept derives from 4 (occurrences) words within the text, in particular those highlighted in blue (bella, bellissimo, Bella, bello); moreover we know that beautiful, in this particular ontology context, is a positive room feature, that is a positive accommodation feature, which is in turn a positive feature.
Figure 6.13: Sample of classification result.
In Section 5, we have already mentioned that our MOoD-TC allows the user to integrate the TC with Application Domains Modules (ADM in the sequel), in order to obtain different functionalities. An ADM can be seen as an external component that, after the integration with the classifier, turns it into a more powerful tool that adds new functionalities to the basic ones offered by TC (Figure 7.1).

![Architecture of a generic module integration.](image)

The main idea is to provide specific methods in order to transform the input of TC (pre-processing) into another one, for example for removing negative forms turning the whole text into a positive form, and manage the standard output generated
by TC, that is the ClassifierObject object, to obtain specific results regarding a particular application domain (post-processing). Furthermore, the ADM can exploit the capabilities and features offered by the classifier, including the powerful aspect of the multilingualism, in different application domains such as, for example, sentiment analysis or opinion spam detection. Let us consider the spam detection domain and let us assume to work just with the TC, that is without an ad hoc external module; as valid input we could input to TC an opinion and a spam ontology, previously created starting from a set of concepts common in spam domain (e.g., "win", "prize", "click here" etc.). The output computed by TC would be a list of the words in the email that are included into the ontology, therefore the words marked as spam.

Figure 7.2: Example of integration between the TC and a module for managing spam detection domain; the result is a spam detector tool.

Assume that we want to add a particular module to transform the classifier into a spam detector (Figure 7.2), that is a system able to classify an opinion as spam or not. While in the pre-processing phase the email may remain unchanged (or just cleaned by keeping those parts of the text that we are actually interested in classifying), in the post-processing phase the module could do different analysis operations on the standard output of the classifier (i.e., the ClassifierObject associated with that classification) in order to return results that tell us if the opinion is actually spam; for example, we could estimate the number of words marked as spam in the opinion
with respect to the length of the email, or check their position inside the email (in the subject field rather than in the body) thus giving a different weight based on where the spam words are found.

Another example, implemented in this work and that we will analyze more in detail in Section 8, is to transform the classifier into a sentiment analyzer (i.e., a tool able to determine the polarity of a document). In this case, the pre-processing phase becomes very important in order to transform the text into a new one in which, for example, we handle the negation and the presence of superlatives, because they are critical aspect in the correct polarity assignment, as shown in Section 2.4. Once TC classifies the new document, it returns as result the final polarity of the entire document (positive or negative).

From these examples it should be clear what the purpose of our work is: providing a very flexible and modular framework that allows a designer or a programmer to exploit the basic multilingual functionalities of TC (which, although basic, are still a significant contribution of this research activity), building more sophisticated functionalities on top of them.

**IMPLEMENTATION DETAILS.** The ADM is an external component designed and implemented by a programmer which must contain two methods (pre-processing and post-processing). The TCModule, that treats a generic ADM, is implemented as a Java interface.

```java
public interface TCModule {
    public String preProcessing(String text);
    public ModuleOutput postProcessing(ClassifierOut c);
}
```

Therefore, the programmer interested in designing and implementing a specific module, creates a Java class that implements the TCModule interface, so that it inherits the two abstract methods that will be implemented by the programmer coherently with the module’s application domain. After defining the new class, the programmer must create a jar file containing it; this jar is, therefore, the ADM representation and it will be passed as input to TC. Moreover, a single jar file can
contain more than one class that extends TCModule; therefore, the programmer has the possibility to define an arbitrary number of ADM classes that will be all contained in the same jar.

In order to allow the program to extract information from external jar files created ad hoc by the programmer, to correctly load the module during the MOoD-TC execution, we used the reflection technique (Figure 7.3). In computer science, reflection is the ability of a computer program to examine and modify the structure and behavior (specifically the values, meta-data, properties and functions) of the program at runtime\footnote{http://en.wikipedia.org/wiki/Reflection_computer_programming}.

![Reflection in MOoD-TC to manage external modules.](image)

Figure 7.3: Reflection in MOoD-TC to manage external modules.

The steps of the module loading phase are:

1 **Jar selection**: from file selection menu, the user chooses which module (that is the jar file) s/he wants to load.

2 **Scan of the jar classes**: once the program obtains the path of the jar, it scans it in order to extract a list containing all the names of the classes within it; for this operation we used the Java package `java.util.jar`.

3 **Classes filtering**: from the list obtained in the previous step, the program checks which classes extend our TCModule interface, as they are the only
classes that can be managed; the valid classes are returned in a new list of Class objects.

4 **Classes instantiation and storage**: at this point, each class in the list is instantiated by the method newInstance() (provided by Class type objects) and inserted as an item of the drop-down menu situated in the module area of the MOoD-TC GUI.

5 **Module selection**: finally, the user can select the module s/he is going to use directly from the drop-down menu, which now contains the list of all classes present within the selected jar, that implement the TCModule interface (Figure 7.4).

![Figure 7.4: Sample of module selected by the user after the jar loading.](image)

Once the module has been selected, the object instantiated from the class that represents that module is saved within the program; at this point, when the classification process is executed, the selected module will change the standard behaviour of our TC thanks to its pre-processing and post-processing methods, turning it into a new tool inherent in the application domain of the created module.

### 7.1 Pre-processing

```java
public String preProcessing(String text);
```
The pre-processing method allows the programmer, who creates a new *Application Domain Module* that implements our interface, to create a new document, starting from the original one, that contains information useful for the classification step (Figure 7.5).

![Figure 7.5: General schema of the pre-processing module phase.](image)

The *preProcessing* function takes as input the path of the text file already selected, and therefore already stored within the program. Remember that an ADM can work only with the classifier, so the classification process can not start if text and ontology have not been selected yet. Therefore, the implemented function can change the text content, based on the needs of the application domain (we see a pre-processing example later in detail, talking about the sentiment analysis module), and returns the new text as a string; obviously, the function will return the same text as a string if the pre-processing phase is not required.

At this point, the new text is taken as input by the TC, which will execute its classification step, generating an output that will be the new input for the second part of the ADM, the post-processing.

### 7.2 Post-processing

```java
public ModuleOutput postProcessing(ClassifierOut c);
```

The post-processing step is the most interesting part of the ADM because it has the task of extrapolating the most significant results from the text, with respect to the application domain, starting from the TC output.
The postProcessing function takes as input a ClassifierOut object. Recall that ClassifierOut is the object that describes the standard output of the TC, that contains the text language, the ontology language and a list of ClassifierObject objects, which contain the most important information about the classification (see Section 6.7). From this information, the function will carry out an analysis on the data in order to obtain the required results; also in this case, in Section 8 we will show a detailed example of post-processing, always making reference to the sentiment analysis module implemented in our work.

As MOoD-TC must be able to show the output of the module, but each module implementation is different from the other, and not known a priori, it was necessary to create a standard object to store the different outputs, and display them on the GUI of the program; this standard object is ModuleOutput.

```java
public class ModuleOutput {
    Object[] columnNames;
    Object[][] data;
    String result;
    // ...
    // constructor, getters and setters
    // ...
}
```

As we have seen in Section 6.7, the GUI of the program models a tabular output, and contains also a label that can contain a String representing a textual result of the classification. The ModelOutput object contains exactly these fields, in particular an array columnNames, that is the list of the column names of the table, a two-dimensional array data which corresponds to all the output table rows, and the
result string to fill the label field of the GUI with relevant information (e.g., in the case of spam detection module, the string result could tell us whether the opinion was classified as spam or not).
In Section 5.2 we have seen a generic overview of how an ADM works; in this chapter we analyze the implementation of an ADM for the sentiment analysis domain in detail. This ADM is named SentiModule and exploits TC to provide ontology-driven sentiment analysis functionalities. We decide to realize in this work a module to manage the domain of sentiment analysis, the SentiModule; therefore, the idea is to transform our TC in a sentiment analyzer (Figure 8.1).

The purpose of this new tool is, starting from the output of our classifier, to tell us if the document that we are analyzing has positive or negative polarity (Figure 8.2).
Figure 8.2: The sentiment analyzer purpose: to extrapolate the polarity from a document.

In our work we present two different sentiment analysis approaches, which vary according to the type of ontology used:

- **Features ontology driven**: we start from a standard ontology in which we represent the features of our domain (for example, a smartphone ontology), and estimate the polarity of the document based on the polarity of the adjectives that refer to the ontology concepts appearing into the text.

- **Sentiment ontology driven**: we start from a sentiment ontology hierarchically organized as follows: this kind of ontology must have at least two direct children of the "Thing" root, which must be named *positive-opinion* and *negative-opinion*, which will represent concepts (typically adjectives) that refer to the considered domain. For example, in the case of a hotel ontology, it makes sense to store concepts such as dirt, economic, polite (in reference to the staff of the hotel) and so on. Subsequently, during the sentiment analysis step, the ontology will be active part of the classification, in order to influence the polarity computation.

While the pre-processing phase is the same for the both modules, they differ instead for the post-processing phase where the final computation of the document polarity occurs. In both modules, we use SentiWordNet to assign polarity to the identified tokens. We will explain later in detail these two variants because they are both contained into the *SentiModule* jar.
Moreover, since the TC offers multilingual support, we decided to take advantage of this feature. In particular SentiModule (and so the both variant) performs sentiment analysis for documents in 5 different languages: English, French, German, Italian and Spanish.

In the sequel we see in detail how pre-processing and the two different post-processing functions were implemented in this module to reach our goal.

8.1 Pre-processing

In Section 2.4 we have discussed some of the problems that should be addressed to correctly identify the polarity of a document. Among the most frequent problems, certainly the main one is the presence of negative clauses within a sentence, that reverse its polarity. For that reason, in the pre-processing phase, we focus on the management of negation, since it normally adds more noise to the classification process. Let us consider this example:

"The room was not clean."

If we not handle the negation in this sentence and we execute TC with the sentiment ontology driven module, the resulting polarity will be wrong. In fact, the polarity of the entire sentence will be positive, because clean (that has positive polarity) is the only text word that appears in our sentiment ontology; but we know that the result is wrong because the sentence has a negative polarity.

In Section 3 we have cited several works in which authors tried to address the problem of negation handling. Most solutions refer to English only, and to the management of its syntactic peculiarities. Since our purpose is to create a multi-language sentiment analyzer, it is therefore necessary to find a way to handle negation for different languages. With respect to the management of the negation in multilingual documents, in the current state of the art a better/optimal solution is not yet present, as the basic problem is that every language has its own grammatical features. The most commonly used solution, the same implemented with some changes in our module, consists in identifying the negation keywords within the sentence and, after that, changing the polarity of the word to which the negative particle refers. The algorithm for the identification and management of the negation,
implemented in the pre-processing method, consists of the following steps:

1 **Identification of negation keyword**: we search, inside the text, the negative particles (negation keyword) for a specific language.

2 **Identification of negation scope**: each negation keyword is assumed to have a scope of influence of the negation; we identified this scope focusing on the nearest adjective to the previous negative keyword identified.

3 **Change of scope polarity**: finally, we invert the polarity of the identified scope word, in particular we change the adjective with its antonym, and we delete the associated negative keyword.

Figure 8.3 shows an example of how the negation is handled in an English sentence, using the described algorithm.

Figure 8.3: Sample of negation handling algorithm.
IMPLEMENTATION DETAILS. The following code shows the algorithm implementation.

```java
@Reverse
public String preProcessing(String text) {
    //Extract token from the input text
    List<String> words = extrapolateTokens(text);
    boolean stop = false;
    String antonym = null;
    for (int i = 0; i < words.size(); i++) {
        //Check if the analyzed token is a negative keyword
        if (isNegativeKeyword(words.get(i))) {
            int negIndex = i;
            //Iterate until find an antonym, the end of the
            //document or a stop punctuation mark
            while (antonym == null && i < words.size() && !stop) {
                //Check tokens after the negative keyword
                String token = words.get(++i);
                //Stop if it is a punctuation mark
                if (isStopToken(token)) {
                    stop = true;
                    break;
                }
                //Otherwise tag the token
                TagObject t = tagToken(token);
                //If it is an adjective, take its antonym
                if (t.getPOS() == POS.ADJECTIVE) {
                    antonym = getAntonym(t.getLemma());
                }
            }
            //If the antonym exists, change the original word
            //with it and delete the associated negative keyword
            if (antonym != null) {
                words.remove(i);
                words.remove(negIndex);
                words.add(i, antonym);
            }
        }
    }
    return text;
}
```
The first step of the algorithm consists in the negation keywords identification. For this purpose, as already mentioned, it was necessary to keep a set of files, one for each language handled by our SentiModule, in which we stored the negation keywords for that particular language; for example, Table 8.1 shows the English negative keywords that we stored in our system.

<table>
<thead>
<tr>
<th>ENGLISH NEGATIVE KEYWORDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
</tr>
<tr>
<td>NO</td>
</tr>
<tr>
<td>ISN’T</td>
</tr>
<tr>
<td>AREN’T</td>
</tr>
<tr>
<td>WASN’T</td>
</tr>
<tr>
<td>WEREN’T</td>
</tr>
<tr>
<td>DIDN’T</td>
</tr>
<tr>
<td>DON’T</td>
</tr>
<tr>
<td>DOESN’T</td>
</tr>
</tbody>
</table>

Table 8.1: English negative keywords stored in SentiModule.

When the pre-processing function is called with the original text, we identify its language and we store the negative keywords associated with the corresponding file into a list of string. The actual pre-processing phase starts now.
As we did for the TC, the function takes in input from the token extrapolation from the text, but in this context we do not eliminate the punctuation because it is very important in order to identify the negation context; for each token stored in the list, we call the `isNegativeKeyword` function.

```java
public static boolean isNegativeKeyword(String word){
    return negativeKeywords.contains(word);
}
```

The `isNegativeKeyword` function checks if one given word is a negative keyword, that is, if it is present among the words stored in our `negativeKeywords` list. If the token is a negative keyword, we move to the second part of the algorithm, namely the identification of the negation scope, otherwise we check the next token.

2 For the second part, we use TreeTagger, in order to tag all the tokens that follow the negative keyword found, until the end of the sentence determined by punctuation marks such as period, semicolon, exclamation mark etc. We analyze the various tokens and we stop when we find one with `POS` equals to adjective; that token is a possible candidate as negation scope (i.e., the word for which it is necessary to perform the change of polarity).

3 The last step of the algorithm is the change of polarity. We face this step by exploiting BabelNet, in particular the `BabelPointer` feature. The idea is to navigate through the semantic relationships between the synsets and, in particular, the antonym relationship, in order to reverse the polarity of a synset.

```java
public String getAntonym (String word) throws IOException{
    BabelNet bn = BabelNet.getInstance();
    ArrayList<String> antonyms = new ArrayList<String>();
    //Assume only with adjective POS
    //Retrieve all word synsets
```
for (BabelSynset syn : bn.getSynsets(lang, word, POS.ADJECTIVE)) {
    // For each word synset, retrieve all the synsets that are in antonym relation with it
    List<BabelSynset> antonym = syn.getRelatedMap().get(BabelPointer.ANTONYM);
    // If exist at least one antonym synset
    if (antonym != null) {
        // Retrieve the first sense of the first antonym synset
        BabelSense antonymSense = antonym.get(0).getSenses(lang).get(0);
        // Return its lemma, that is the antonym of the input word
        return antonymSense.getLemma();
    }
    return null;
}

We select the various synsets associated with the lemma of the identified token. For each of them we retrieve its associated list of antonym synsets. To do this, we use the `getEdges` BabelNet function, that allows us to retrieve all the semantic relations available for that particular synset. By calling this function with the `BabelPointer.ANTONYM` pointer, we filter only the synset marked as opposed compared to the starting synset. From the retrieved list of antonym synsets, we extrapolate the first one, because it is considered more akin to the opposite synset starting, and we take the first sense of the corresponding senses list from it, always in according with the text language. The example in Figure 8.4 shows all the steps for the extrapolation of the antonym of the word "clean".

At the end of the process, if we have found an antonym, we replace the token identified by it, and we delete the token that represents the negative keyword. Instead, if the antonym is null, which means that the token has not antonym, we
Figure 8.4: Sample of "clean" antonym retrieve.

repeat the process with the next token, always equals to adjective, until the next stop punctuation marks. In the end we obtain our words list, from the source text, in which there are some words substituted by their antonyms. The last step is to rebuild the new text, from the list, and use it as input of our TC; to do this, we build a string containing all the concatenated words from the list. Once the text is properly prepared, we pass it as input to the TC in order to classify it and, on the obtained classification results, execute the post-processing step.

There are several considerations about this pre-processing phase. First of all the languages complexity, that makes the negation identification a non-trivial task. Our approach appears to be a good choice for the multilingual negation handling, but it works in quite simple contexts, where the negative clause precedes the word to
be negated. This is a common feature of the five considered European languages, but changes in other languages where the negation, for example, is represented by a combination of the negation keyword with the word to be negated. There are also many other complex contexts in which the negation is not in form of negative clauses, but it is intrinsic within the sentence, making its identification a very complex task to do in automatic way. Again, there are more subtle questions about the polarity computation; in fact, we assume to change the polarity of the word identified with that of its opposite, but the result is not always correct because, for example, not necessarily the polarity of "good" is the same of "not bad".

8.2 POST-PROCESSING

After the pre-processing phase, the TC stores the result of the classification into the \textit{TCOutput} object. In this post-processing phase we analyze the document w.r.t. an ontology that represents its domain, in order to extract information about its polarity.

We talked about two different variants of post-processing, which change in accordance with the type of ontology used. Obviously, given that the ontology used may be different, also the \textit{TCOutput} object will contain different results to be managed by corresponding post-processing variants. In both post-processing versions, the main output will be the polarity of the document analyzed with other useful information. We will see how, in the case of the sentiment ontology driven variant, we could improve the sentiment analysis results, exploiting the ontology structure.

\textbf{IMPLEMENTATION DETAILS.} We briefly present the common part of both versions, which differ by the central body of the function, that is the polarity extraction algorithm.

```java
@Override
public ModuleOutput postProcessing(TCOutput output) {
    Object columnNames[] = new Object[]{"Lemma word","Ontology word","Polarity","Root"};
```
The portion of code above deals with the ADM output, which will be the final result of the Sentiment Analysis process. In particular, for each ontology word included in the text, we are interested in storing its lemma in the language of the text, its corresponding concept in the ontology, its polarity and its path within the ontology. We have the major part of this information into the TCOutput object, returned by the TC classification. The polarity of each word, instead, will be computed in different ways depending on the post-processing variant used. Moreover, as label of the ModuleOutput object, we return the result of the analysis of the entire document polarity; in particular, the label will be "Positive document" if the resulting polarity is positive, "Negative document" otherwise.

**SentiWordNet** The tool that we use in both versions, in order to extract the polarity of a word, is SentiWordNet (Section 4.4); SentiWordNet assigns positive and negative polarity values to each WordNet synset. We used the class provided by the library available from the web site, the SentiWordNetClass, a java class that contains two methods to manage the SentiWordNet.txt file:

- **Class constructor**

```java
public SentiWordNetClass(String pathToSWN)
```
CHAPTER 8: SentiModule

<table>
<thead>
<tr>
<th>SYNSET ID</th>
<th>POSITIVE</th>
<th>NEGATIVE</th>
<th>POLARITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>101513619</td>
<td>0.25</td>
<td>0.25</td>
<td>0.0</td>
</tr>
<tr>
<td>01676517</td>
<td>0.75</td>
<td>0</td>
<td>0.75</td>
</tr>
<tr>
<td>00196449</td>
<td>0</td>
<td>0.625</td>
<td>-0.625</td>
</tr>
</tbody>
</table>

**Resulted polarity**: 0.12

Table 8.2: Sample of polarity calculation for the word "terrific"; the resulting polarity is the sum of the synsets polarities in which "terrific" appears.

The class constructor takes as input the SentiWordNet.txt path in order to extract from it all the saved words, to store them into a hashmap, with their respective polarities. The resulting polarity is computed as the sum of the various polarities that the word assumes into the different synsets to which it belongs. The polarity of a synset $s$ is computed as its positive polarity minus its negative one:

$$Polarity(s) = Pos(s) - Neg(s)$$

For example, in Section 4.4 we have seen that the word "terrific" assumes three different polarities in three different synsets; Table 8.2 shows the resulting polarity calculation for "terrific".

- **Extract method**

  ```java
define extract(String word, String pos)
```

  The second method, offered by the class, is the extract one, that retrieves the polarity of a given word with its relative POS from the hashmap, created by the constructor.

  $$0.12 = extract(terrific, POS.ADJECTIVE)$$

  The way in which this class handles the polarity computation is quite simple; in fact it is an unweighted polarities average, without a disambiguation process, which makes the Sentiment Analysis result often inaccurate or even wrong.
8.2.1 Features Ontology Driven

The first variant of our SentiModule is the common approach used by various sentiment analysis tools. The main idea is to identify in the text the various concepts, that are stored in a features ontology domain oriented, like our MySmart ontology, a features ontology that represents the smartphone domain (Figure 8.5).

For each identified concept, we search its associated adjective and we extract its polarity; finally, we compute the resulting polarity of the document as sum of the different identified adjective polarities.

Let us consider, for example, this review about a new smartphone:

"This smartphone is good: average battery, but the processor is amazing."

smartphone, battery and processor will be properly identified in the text because they are in our MySmart ontology; at this point we proceed to calculate their polarity, i.e., the polarity of their associated adjectives, respectively good, average and amazing. The resulted polarity of the document will be the sum of the various adjectives polarities (Figure 8.6).
Note that the use of the ontology gives us several information, useful for the analysis; for example, we can understand what are the specific lacks of a smartphone, just analyzing a set of reviews and finding that a particular feature is frequently marked with a negative polarity value.

**IMPLEMENTATION DETAILS.** The steps to implement the algorithm described above are the following:

1. Extrapolate a list of sentences from the pre-processed text. We assume that each adjective refers to a noun present in the same sentence, delimited by punctuation marks such as period, colon, semicolon etc.

2. For each extrapolated sentence, we search adjectives in it, after the Part-Of-Speech tagging phase on each token of the sentence.

3. For each identified adjective, we search the corresponded noun inside the same sentence, i.e., one token with POS equals to noun that is into our ontology.

4. We calculate the polarity of the adjective identified, using SentiWordNet; we should recall that, in order to correctly use SentiWordNet, the adjective must be in English, otherwise we translate it using BabelNet.

5. Finally, we calculate the average of all the identified polarities in order to calculate the resulted document polarity. If the result is greater or equal than
a threshold \( t \), the document will be labelled as positive, negative otherwise. We see in Section 9 how we choose the threshold that determines the resulted document polarity.

The following code is the algorithm implementation in the post-processing method of the features ontology driven approach.

```java
public ModuleOutput postProcessing(TCOutput output) {
    // ModuleOutput initialization
    // Sentences extrapolation from text after pre-processing phase
    ArrayList<String> sentences = getSentences(this.processedText);
    ArrayList<TagObject> tagObj = new ArrayList<TagObject>();
    // Sum of polarities
    double sum = 0.0, count = 0;
    // Number of identified couple (ontology concept, adjective associated)
    for (String sentence : sentences){
        // Execute the POS-Tagging for each token in each sentence
        tagObj = getTokens(sentence)
        for (int i = 0; i < tagObj.size(); i++){
            TagObject t = tagObj.get(i);
            // If exists an adjective, search its nearest ontology lemma
            if (t.getPOS() == POS.ADJECTIVE){
                String lemma = nearestLemma(tagObj, i);
                if (lemma != null){
                    // Perform polarity calculation of the adjective associated to the lemma
                    // (with eventually BabelNet translation)
                }
            }
        }
    }
}
```
In this variant we used only SentiWordNet to determine the polarity of the document. We have seen how the limits imposed by this tool make the polarity calculation often inaccurate. Although we know that some concepts are definitely positive or negative in a particular context, SentiWordNet assigns to them an average polarity that makes the result wrong in the context that we are analyzing. This error may be small (e.g., an adjective with a 0.8 polarity turns to 0.4, thus decreasing its weight in the final calculation), or large (positive adjectives takes negative polarity). Therefore, all this generated noise falsifies the results, as show in Section 9.2. For those reasons, we have tried to overcome these limits in the second version of SentiModule, with the sentiment ontology driven approach.

8.2.2 Sentiment Ontology Driven

In this approach, we use an ontology of sentiments, to influence the document polarity calculation. As we have already described previously, the structure of this kind of ontology is fundamental; in particular a sentiment ontology must have at least two direct children of the “Thing” root, which must be named positive-opinion and negative-opinion, that contain positive or negative concepts (usually adjectives) in that particular domain. Figure 8.7 shows an example of a sentiment ontology, in
the hotel domain.

The main idea of this approach is to force the results of the polarities calculation obtained from SentiWordNet, in accordance with our ontology. Unlike the features driven approach, the polarity calculation is executed directly on the ontology concepts, not on the associated adjectives. This approach avoids problems due to wrong adjective identification in the sentence. The polarity of a concept can also be fixed in accordance with its ontology root (positive or negative); in fact we suppose to believe more in the ontology than in SentiWordNet. Another strength of this approach is that in the polarity calculation we do not consider only the adjectives, but all those concepts that we know to be positive or negative in the considered domain; for example in SentiHotel, there are non-adjective concepts such as worship or conscience respectively positive and negative concepts that appear in hotel reviews.

**Implementation Details.** The TCOoutput object contains all the information necessary for the Sentiment Analysis (the list of ClassifierObjects, the ontology language and the text language), because we are interested in calculating only the polarity of the concepts of our sentiment ontology, correctly classified by TC. For each ClassifierObject we calculate the polarity of the concept ontology. If the ontology language is other than English, we translate the concept in English.
language because of SentiWordNet; the polarity calculation is oriented by the fixScore function. In the end we consider as resulting document polarity the average of all the identified polarities. Also in this approach, if the result is greater or equal than a threshold $t$, the document will be labelled as positive, negative otherwise.

The following code is the algorithm implementation in the post-processing method of the sentiment ontology driven approach.

```java
@Override
public ModuleOutput postProcessing(TCOutput output) {

    // ModuleOutput initialization

    int size = 0, positive = 0, negative = 0;
    double sum = 0.0, score = 0.0, tmp = 0.0;
    SentiWordNetClass sentiwordnet = new SentiWordNetClass(pathToSWN);
    ArrayList<ClassifierObject> info = (ArrayList<ClassifierObject>) output.getInfo();

    // Perform polarity calculation for each ClassifierObject
    for (int i = 0; i < info.size(); i++) {
        ClassifierObject c = info.get(i);
        ArrayList<String> tree = (ArrayList<String>) c.getOntologyTree();
        String root = tree.get(0);
        if (root.equals("positive-opinion")) positive++;
        else if (root.equals("negative-opinion")) negative++;

        String concept = c.getOntologyWord();
        // If the ontology language is not English, translate the ontology word to English
        if (!output.getOntLang().equals(Language.ENGLISH))
            concept = babelNetTranslation(concept);

        // Assign to the word the fixed SentiWordNet polarity and multiply its score by the word number
        // (This is a placeholder for the actual calculation)
    }

    // Calculate the average polarity
    for (int i = 0; i < info.size(); i++) {
        ClassifierObject c = info.get(i);
        ArrayList<String> tree = (ArrayList<String>) c.getOntologyTree();
        String root = tree.get(0);
        if (root.equals("positive-opinion")) positive++;
        else if (root.equals("negative-opinion")) negative++;

        String concept = c.getOntologyWord();
        // If the ontology language is not English, translate the ontology word to English
        if (!output.getOntLang().equals(Language.ENGLISH))
            concept = babelNetTranslation(concept);

        // Assign to the word the fixed SentiWordNet polarity and multiply its score by the word number
        // (This is a placeholder for the actual calculation)
    }

    // Calculate the average polarity
    double avgPolarity = sum / size;

    if (avgPolarity >= t) return new ModuleOutput(ModuleOutput.SUCCESS, "positive");
    else return new ModuleOutput(ModuleOutput.FAIL, "negative");
}
```
8.2 Post-processing

```java
// of occurrences inside the document
tmp = fixScore(Math.floor((sentiwordnet.extract(
    concept, getSentiPOS(c.getPos()))) * 100)/100, root) *
c.getNumberOfOcc();
size += c.getNumberOfOcc();
sum += tmp;
// Insert new output row
rowData[i] = new Object[]{c.getLemmaWord(),
    c.getOntologyWord(), tmp, root};
}
// Average calculation
score = sum/(positive+negative);
if (score >= THRESHOLD)
    String out = "POSITIVE REVIEW";
else
    String out = "NEGATIVE REVIEW";

// Return ModuleOutput object
}
```

The `fixScore` function changes the word polarity calculated with SentiWordNet, in accordance with its ontology root. We want to give more weight to the information associated with the ontology. If a concept has positive-opinion root, but SentiWordNet assigns it a negative polarity, then we reverse the polarity, changing the sign, because we build our ontology assuming that the concept is actually positive (similarly for negative concept). However, if the SentiWordNet polarity of a word is 0 (because the resulting polarity is actually 0, or because that word does not appear in SentiWordNet), we change it with a fixed value, with positive or negative sign based on its ontology root.

```java
public static double fixScore (double score, String root) {
    if (root.equals("positive-opinion")){
        if (score < 0)
            score = -score;
        else if (score == 0.0)
```
For example, *top* and *funny* are two concepts of our *SentiHotel* ontology for which the *fixedValue* function performs a polarity change, with a fixed value equals to 0.2 (Table 8.3).

<table>
<thead>
<tr>
<th>word</th>
<th>SentiWordNet polarity</th>
<th>Ontology root</th>
<th>New polarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>0</td>
<td>positive-opinion</td>
<td>0.2</td>
</tr>
<tr>
<td>funny</td>
<td>-0.15</td>
<td>positive-opinion</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Table 8.3: Output of process function.
CHAPTER 9

EXPERIMENTS

This section describes the research question, metrics, experimental objects, procedure and results of the empirical study conducted to evaluate the performance of MOoD-TC. We test our system with the sentiment analysis module in classifying smartphones reviews (features ontology driven variant) and hotels reviews (sentiment ontology driven variant) in five languages (i.e., English, Italian, Spanish, French and German).

9.1 RESEARCH QUESTION AND METRICS

Our study aims at answering the following research question:

RQ: Is MOoD-TC able to classify multilingual documents w.r.t. the opinion / sentiment they describe?

The goal of our research question is to evaluate the effectiveness of MOoD-TC in classifying documents written in different languages w.r.t. the opinion / sentiment they describe. The metrics used to answer RQ is the number of documents correctly classified over the total number of documents.

9.2 FEATURES ONTOLOGY DRIVEN EXPERIMENTS

In this section we show the results obtained with the first variant of SentiModule. In the following we will use MOoD-TC\textsubscript{Features} to refer to the use of MOoD-TC with the features ontology driven approach.
In order to perform our experiments, we used MySmart as features ontology to represent the smartphone domain, introduced in Section 8.6.

9.2.1 DOCUMENTS

We conducted our evaluation of MOoD-TCFeatures over a sample of multilingual smartphones reviews. In particular, we focused on classifying reviews in five different languages: English, Italian, Spanish, French and German. We randomly selected the reviews from Amazon. In particular, for English reviews we selected 100 reviews, 20 for each value of the overall score (from 1 to 5). For the other languages, we randomly selected 50 reviews for each language, 10 for each value of the overall score (from 1 to 5), resulting in a total of 200 reviews.

9.2.2 PROCEDURE

To answer our RQ we proceeded as follows:

- For each review, we executed MOoD-TCFeatures and recorded the classification w.r.t. the features ontology MySmart. In particular, we recorded the resulting polarity $P(r)$, computed as average of the adjectives polarities associated with the review words that belong to the ontology. Since the synset polarity assigned by SentiWordNet is between -1 and 1, the resulting polarity belongs to the same range.

- We classified each review as positive if $P(r) \geq Tr$, negative otherwise. $Tr$ is a threshold that in our case study was set to 0 (the average of the polarity range).

- For each review, we compared our classification (computed as shown in the previous step) with the overall score provided by the real user and recorded in the dataset together with the review. The classification is correct when: (1) we classified a review as positive and the user provided a score $\geq 3$, (2) we classified a review as negative and the user provided a score $< 3$. In the other cases the classification is wrong.

---

1http://www.amazon.com/
9.2.3 RESULTS

Table 9.1 reports the data used to answer RQ. For each dataset (i.e., set of reviews in a specific language) and for each overall score (i.e., the number [1,5] assigned by the users), it reports the number of correctly classified reviews and the corresponding percentage over the total number of reviews. In the last columns, we report aggregated results over all the five datasets.

<table>
<thead>
<tr>
<th>Overall Score</th>
<th>Reviews EN</th>
<th></th>
<th>Reviews IT</th>
<th></th>
<th>Reviews FR</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Correctly Classified</td>
<td>Total</td>
<td>Correctly Classified</td>
<td>Total</td>
<td>Correctly Classified</td>
<td>Total</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>75.0%</td>
<td>6</td>
<td>60.0%</td>
<td>7</td>
<td>70.0%</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>60.0%</td>
<td>7</td>
<td>70.0%</td>
<td>5</td>
<td>50.0%</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>55.0%</td>
<td>5</td>
<td>50.0%</td>
<td>4</td>
<td>40.0%</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>30.0%</td>
<td>3</td>
<td>30.0%</td>
<td>3</td>
<td>30.0%</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>40.0%</td>
<td>4</td>
<td>40.0%</td>
<td>3</td>
<td>30.0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Overall Score</th>
<th>Reviews ES</th>
<th></th>
<th>Reviews DE</th>
<th></th>
<th>All Reviews</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Correctly Classified</td>
<td>Total</td>
<td>Correctly Classified</td>
<td>Total</td>
<td>Correctly Classified</td>
<td>Total</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>80.0%</td>
<td>4</td>
<td>40.0%</td>
<td>40</td>
<td>66.7%</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>70.0%</td>
<td>5</td>
<td>50.0%</td>
<td>36</td>
<td>60.0%</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>40.0%</td>
<td>3</td>
<td>30.0%</td>
<td>27</td>
<td>45.0%</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>50.0%</td>
<td>3</td>
<td>30.0%</td>
<td>23</td>
<td>38.3%</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>70.0%</td>
<td>4</td>
<td>40.0%</td>
<td>26</td>
<td>43.3%</td>
</tr>
</tbody>
</table>

Table 9.1: $MOoD-TCFeatures$ results (threshold = 0).

Concerning the reviews with evaluation 5 (i.e., very good) or 4 (good), we can see that $MOoD-TCFeatures$ is not able to provide a good resulting classification. In particular, in case of overall score = 5 and considering all the languages employed in the five datasets, $MOoD-TCFeatures$ correctly classifies only the 66.7% of the reviews, with worst results in case DE. Similarly, $MOoD-TCFeatures$ classifies only the 60.0% of the reviews with overall score = 4.

The worst classification results concern the reviews with evaluation 1 (i.e., very
bad) or 2 (bad). Indeed, respectively only in the 43.3% and 38.3% of the cases it produces a correct results. From the data reported in Table 9.1, it is evident that the result of the classification is unbalanced, and tends to favour positive ratings.

The main cause of the poor results of this approach is due to the adjective identification that gives the polarity to the associated ontology concept. This identification is wrong most of the time, due to the different grammar rules that do not allow the use of a general multilingual approach to the problem, causing a lot of noise in the resulting document polarity computation.

Precisely because these first results were not satisfactory, we decided to use a different approach that could generate less noise in the classification, the sentiment ontology driven one.

9.3 SENTIMENT ONTOLOGY DRIVEN EXPERIMENTS

In this section we show the results obtained with the second variant of SentiModule. In the following we will use $MOoD-TC_{Sentiment}$ to refer to the use of MOoD-TC with the sentiment ontology driven approach.

To perform our experiments, we used SentiHotel as sentiment ontology to represent the hotel domain, introduced in Section 8.2.2.

9.3.1 DOCUMENTS

We conducted our evaluation of $MOoD-TC_{Sentiment}$ over a sample of multilingual TripAdvisor’s reviews. In particular, we focused on classifying reviews in five different languages: English, Italian, Spanish, French and German.

About English reviews, we chose Wang TripAdvisor Data Set\cite{49}; this Data Set is composed by more than 12000 Json files each of which contains about 10 TripAdvisor reviews with different information about them (e.g., review text, overall score, ID). From this dataset we randomly chosen 455 English reviews with an equilibrate distribution of different overall scores (i.e., we have a similar number of positive and negative reviews).

As regards the other languages (Italian, Spanish, French and German), we randomly selected 50 reviews for each language, 10 for each value of the overall score (from 1 to 5), resulting in a total of 200 reviews.
9.3.2 Procedure

To answer our RQ we proceeded as follows:

1. For each review, we classified $MOoD\cdot TC_{Features}$ and recorded the classification w.r.t. the sentiment ontology $SentiHotel$. In particular, we recorded the resulting polarity $P(r)$, computed as average of the fixed polarities of the ontology concepts found in the review. Since the synset polarity assigned by SentiWordNet is between -1 and 1, the resulting polarity belongs to the same range.

2. We classified each review as positive if $P(r) \geq Tr$, negative otherwise. $Tr$ is a threshold that in our case study can assume the following values: 0, 0.2.

3. For each review, we compared our classification (i.e., computed as shown in the previous step) with the overall score provided by the real user and recorded in the dataset together with the review. The classification is correct when: (1) we classified a review as positive and the user provided a score $\geq 3$, (2) we classified a review as negative and the user provided a score $< 3$. In the other cases the classification is wrong.

9.3.3 Results

Table 9.2 reports the data used to answer RQ. For each dataset (i.e., set of reviews in a specific language) and for each overall score (i.e., the number $[1,5]$ assigned by the users), it reports the number of correctly classified reviews and the corresponding percentage over the total number of reviews. In the last columns, we report aggregated results over all the five datasets.

Concerning the reviews with evaluation 5 (i.e., very good) or 4 (good), we can see that $MOoD\cdot TC_{Sentiment}$ is able to provide, most of the times, a correct classification. In particular, in case of overall score = 5 and considering all the languages employed in the five datasets, $MOoD\cdot TC_{Sentiment}$ correctly classifies the 94.2% of the reviews. In two cases, IT and DE the classification is very accurate; similarly, $MOoD\cdot TC_{Sentiment}$ correctly classifies the 90.1% of the reviews with overall score = 4.

Conversely, $MOoD\cdot TC_{Sentiment}$ is not able to classify correctly the reviews with evaluation 1 (i.e., very bad) or 2 (bad). Indeed, respectively only in the 67.2%
and 58.7% of the cases it obtains correct results. From the data reported in Table 9.2, it is evident that the result of the classification is unbalanced, and tends to favour positive ratings.

We reported also the classification returned for the reviews with overall score = 3. They express a judgement that obviously is neither positive nor negative. Thus, a binary classification (i.e., positive vs negative) cannot be used for classifying such kind of reviews. But, for such reviews, we expect \( MOoD-TC_{Sentiment} \) to behave as a classifier which assigns a review to one of the two classes (positive and negative) with a probability of 50%, while, by adopting the threshold \( Tr=0 \), this is not true (see the 82.7% reported in the table). Thus we searched for a threshold value that allowed us to obtain, for the overall score = 3, a results as close as possible to 50%. Such threshold value is 0.2.

Table 9.3 reports the results of the classification performed using \( Tr=0.2 \). Concerning the reviews with evaluation 5 (i.e., very good) or 1 (very bad), we can see
that $M OO D-TC_{Sentiment}$ is able to provide, most of the times, a correct classification. In particular, in case of overall score = 5 and considering all the languages employed in the five datasets, $M OO D-TC_{Sentiment}$ correctly classifies the 92.6% of the reviews. In two cases, IT and DE the classification is perfect; similarly, $M OO D-TC_{Sentiment}$ correctly classifies the 89.1% of the reviews with overall score = 1.

As expected, in cases of reviews that do not express a sharp judgement (i.e., overall score 4 and 2) the correctness of the classification performed by $M OO D-TC_{Sentiment}$ is slightly worse. In particular, in case of overall score = 4 and 2, and considering all the languages employed in the five datasets, $M OO D-TC_{Sentiment}$ correctly classifies respectively the 88.8% and 82.6% of the reviews.

Obviously the choice of the threshold is subjective but note that its value has been selected in order to balance the results on the reviews with overall score = 3 and not for achieving the best possible classification. As future work we plan to investigate
the reason behind the unbalanced classification obtained when using the standard threshold \( Tr = 0 \). For instance, this could depend on the kinds of positive/negative words used in the reviews or from the coverage of the positive/negative words by BabelNet. However, we plan to better validate the effectiveness of the approach implemented by \( MOoD-TC_{Sentiment} \) through a cross-validation (aka, leave-one-out) procedure. For instance, we will use a leave-one out cross validation with \( k = 5 \), where 5 is the number of original data sets. Thus, we will split the original datasets into four datasets used for training the threshold \( Tr \) and one dataset used for testing the effectiveness of \( MOoD-TC_{Sentiment} \) employing such threshold, with the testing dataset rotated so as to test \( MOoD-TC_{Sentiment} \) on each of the five available datasets. Obviously, to this end we will increase the number of reviews considered for the languages different from English.

To summarise, with respect to the research question RQ we can say that, using an appropriate threshold, \( MOoD-TC_{Sentiment} \) is able to classify correctly the majority of the reviews in all the five considered languages. The preliminary evaluation reported in this thesis shows the feasibility of the approach implemented by \( MOoD-TC_{Sentiment} \) even if further investigations are required to refined and fine-tune both the approach and the tool.


**CHAPTER 10**

**CONCLUSION AND FUTURE WORK**

In this work we have discussed the design, features and implementation details of MOoD-TC and our first experiments with it. The results are promising: the text classifier TC is able to classify documents in a significant set of languages, in particular the main European ones, with regards to multilingual ontology. The language does not have to be known a priori but is automatically identified by our system. Moreover, thanks to the system modularity, TC can be integrated with external modules (ADM), created to manage a particular application domain. To achieve this purpose, an ADM offers two processing methods (pre-processing and post-processing), in order to handle the TC input and output.

In particular we have presented Sentimodule, the Sentiment Analysis ADM, with its two variants: features ontology driven (MOoD-TCFeatures) and sentiment ontology driven (MOoD-TCSentiment).

One limitation of TC is the number of supported languages; indeed, TreeTagger currently owns corpora for only 19 languages. Thanks to the system modularity, however, if we need to classify a document in a different language, it is just necessary to create a new TreeTagger corpus for that particular language and include it into the system.

As regards Sentimodule, we made a first basic version that will be subsequently improved. This first version handling the negation problem relatively easily, because of the multilingual approach; in fact, we assume that the negation is represented by negative keywords, and the negation scope is the first next adjective founded in the analyzed sentence. Again, in MOoD-TCFeatures we assume that there are some specific grammar rules, inside the document, shared among the handled languages.
With this approach we obtained poor results, caused mainly by the difficulty in identifying the correct adjective that refers the ontology concept inside the text. In the sentiment ontology driven approach, instead, the results are more interesting, but not optimal. In our future work we plan to investigate the reason behind the unbalanced classification obtained when using the standard threshold $T_{\gamma}=0$. As we already stressed, our work is not aimed at solving all the well known open problems in multilingual text classification and sentiment analysis such as negation, irony \cite{...}, named entities, etc. but rather, it provides a flexible and modular framework ready for integrating, with limited effort, the results and algorithms addressing the above problems coming form the research community.
BIBLIOGRAPHY


APPENDIX A

COMPOUND WORDS MANAGEMENT

For the following algorithm we need as support 3 sets:

- **A finalized-class-set**: a set containing all the classes of the sentences already parsed.
- **An incomplete-class-set**: a set of all that classes matched only partially.
- **A complete-class-set**: a set of all that classes we found all the words of in the sentence.

The idea behind the algorithm is that the words that compound a class of the ontology are not in different sentences of the analyzed text. So if we try to match a class compound of multiple words, and we scan the text word by word, every time we encounter a semi-colon or a dot, we consider matched all the classes we have found all the words of (i.e., we insert all the elements of the complete-class-set into the finalized-class-set, and empty the incomplete-class-set and the complete-class-set).

During the scanning of the text if we find a word that match a class made by just a word, we save this class into a set containing the “complete” classes (the complete-class-set).

If we find a word that matches with a part of a compound class (e.g., we found “display” that matches with part of the class “display colors”), we search, into a set containing all the classes matched “partially” (the incomplete-class-set), a class that is completed by this word (e.g., we already have a partial class made of “display colors” with only “display” matched and then we found the word “colors”). If we
find it we remove that class from the incomplete-class-set and we put it signed as complete into the complete-class-set; if not it could be that we have a class into the ontology that is made by the same words of another class plus others (e.g., we have into the ontology the class made of the word “display”, and also the class made of the words “display colors”) and it could be that we have already parsed and inserted the smallest class into the complete-class-set. For this reason we search into the complete-class-set a complete class made of words that, together with the word we have just parsed, complete a compound class. If we find it, we remove it from the complete-class-set and insert it in the same set this time as the new compound class; if not it means we cannot complete anyone of the “partial” classes we have found till this point with this word. So we need to save the word we found as an “incomplete” class: if we find a class into the incomplete-class-set that would be “a bit more complete” with the new word (e.g., we already have a partial class made of “cpu quad core” with only “cpu” matched and my new word is “quad”), we sign it as matched with the addition of the new word; else we insert a new “partial” class made of the new word into the incomplete-set-class.

PSEUDOCODE:

```plaintext
FinalizedClassesList = empty List
CompleteClassesList = empty List
IncompleteClassesList = empty List

foreach(word in Text) {
  if (word is equal to ‘;’ or ‘.’) {
    insert all elements of CompleteClassesList into FinalizedClassesList;
    empty CompleteClassesList;
    empty IncompleteClassesList;
  } else {
    if (word match with a simpleClass of the Ontology) {
      insert that simpleClass into CompleteClassesList
    } else if (word match with a compound class of the Ontology) {
      if (word complete an incompleteCompoundClass contained
```
into IncompleteClassesList) {
    remove that incompleteCompoundClass from
    IncompleteClassesList;
    insert the completedCompoundClass made of
    (incompleteCompoundClass + word) into
    CompleteClassesList
} else if (one completeClass from CompleteClassesList
    + word) match with a compoundClass of the Ontology) {
    remove completeClass from CompleteClassesList;
    insert compoundClass made of (completeClass + word)
    into CompleteClassesList;
} else if (one incompleteClass from
    IncompleteClassesList + word) match with part of a
    compoundClass of the Ontology) {
    remove that incompleteClass from
    IncompleteClassesList;
    insert a new incompleteClass made of (the old
    incompleteClass + word) into
    IncompleteClassesList;
} else {
    insert a new incompleteClass made of (word) into
    IncompleteClassesList;
}