Watson Versus the World

André Ricardo Anselmo Félix

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Prof. Pável Pereira Calado

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Member of the Committee: Prof. Nuno João Neves Mamede

June 2018

Abstract

Ever since IBM came up with the idea of developing a system capable of being the best at *Jeopardy!*, back in 2004, Watson has been through many changes. After its victory against the best players ever on the show, in 2011, Watson has changed from a system built only to answer to clues in a certain format, to a system capable of answering almost any question, whether written or spoken. Watson now supports several different languages, unlike the original system built for English only. It can answer questions and analyze meta-data in a matter of seconds, providing insights about almost any field of expertise, like medicine, finance or food.

But, despite its success and popularity, no actual comparison between Watson and other Natural Language Processing (NLP) toolkits has been performed. Because of this, we propose a formal evaluation of Watson's capabilities, comparing its performance with that of another toolkit, over a set of predefined NLP tasks.

IBM provides access to Watson's Application Programming Interface (API) through an online platform called Bluemix, which was used to develop part of the solution, and addressed in more detail later. Insights over the existing NLP tools are presented. Details about the implementation of the chosen tasks are discussed, along with the evaluation methodology that was followed and the results that were obtained.

Keywords: IBM Watson, Natural Language Processing, Bluemix, Natural Language Processing Toolkits

Resumo

Desde que a IBM teve a ideia de desenvolver um sistema que seria capaz de ser o melhor no concurso *Jeopardy!*, em 2004, o sistema Watson já sofreu muitas alterações. Após ter ganho, em 2011, contra os melhores concorrentes que já estiveram no programa, o Watson mudou de um sistema desenvolvido apenas para responder a pistas de um certo formato, para um sistema capaz de responder a quase qualquer questão, quer escrita, quer falada. Neste momento o sistema Watson está disponível em várias línguas, ao contrário do sistema original que foi construído apenas para Inglês. Para além disto, o sistema consegue responder a perguntas e analisar metadados em poucos segundos, conseguindo transmitir aos seus utilizadores conhecimentos sobre quase qualquer área de conhecimento, como, por exemplo, medicina, finanças, ou cozinha.

Mas, apesar do seu sucesso e popularidade, não existe nenhuma comparação entre o sistema da IBM e outras ferramentas de Processamento de Língua Natural (PLN). Por esta razão, propomos uma avaliação formal das capacidades do sistema Watson, comparando o seu desempenho com o que é atingido utilizando outra ferramenta, sobre um conjunto pré-definido de tarefas de PLN.

A IBM dá acesso à Application Programming Interface (API) do Watson através de uma plataforma online chamada Bluemix, que foi usada para desenvolver parte da solução, e vai ser apresentada em mais detalhe mais à frente. Uma discussão sobre as ferramentas de PLN já existentes é apresentada. Detalhes sobre a implementação das tarefas escolhidas também vão ser apresentados, para além da metodologia de avaliação que foi seguida e os resultados que foram alcançados.

Palavras-chave: IBM Watson, Processamento de Língua Natural, Bluemix, Ferramentas de Processamento de Língua Natural

Acknowledgments

First, I would like to thank my parents for their constant sacrifices, love and support ever since I can remember. Despite their divergences in recent years, if it wasn't for both their efforts, this work, and my path on IST, would not exist.

I would also like to thank both my advisors, Prof. Luísa Coheur and Prof. Pável Calado, for always being available, for their guidance, feedback, and efforts to make this work possible. Since the moment I met Prof. Luísa to talk about her thesis proposals, I knew that she would be a great advisor, and I was right. Thank you, once again, for all the tips, corrections, and motivation to constantly do better.

A very, very big thank you to my girlfriend, Catarina Mendes. For all the late nights in her company, for all the times when I doubted myself and wanted to stop, but didn't let me, for all the laughs and distractions, for everything I don't have the space to write about. I could not thank her enough for her support, encouragement, love and companionship throughout this journey.

I must also thank all of my friends and colleagues I have met throughout the years on IST, for their support and help, for the late nights, the existential crises when things wouldn't go well, the fun times, all the many great memories from these 6 years, they all made this journey worthwhile, and inspired me to improve myself all this time. A very big thank you to them.

I would also like to express my gratitude to my friends outside of IST, specially to Carina Neves, for her constant encouragement and support, even if we couldn't be together many times during the writing of this thesis. Thank you for believing in me, even when I struggled to.

The journey that led here, and this thesis, would not have been possible without them. A very big thank you to all of them.

Contents

List of Figures

List of Tables

List of Acronyms

AI Artificial Intelligence

API Application Programming Interface

CoNLL Conference on Computational Natural Language Learning

CREOLE Collection of Reusable Objects for Language Engineering

CSV Comma-Separated Values

GATE General Architecture for Text Engineering

GMB Groningen Meaning Bank

HPSG Head-Driven Phrase Structure Grammar

HTML HyperText Markup Language

IDE Integrated Development Environment

IE Information Extraction

IR Information Retrieval

JSON JavaScript Object Notation

LUIS Language Understanding Intelligent Service

MALLET Machine Learning for LanguagE Toolkit

MARF Modular Audio Recognition Framework

ML Machine Learning

MRS Minimal Recursion Semantics

MT Machine Translation

MUC Message Understanding Conference

NER Named Entity Recognition

NLC Natural Language Classifier

NLP Natural Language Processing

NLTK Natural Language Toolkit

NLU Natural Language Understanding

PaaS Platform-as-Service

PoS Part of Speech

QA Question Answering

XML Extensible Mark-up Language

1

Introduction

1.1 Motivation

Many people know IBM[1]'s Artificial Intelligence (AI) system, Watson[2], due to its victory in the North American gameshow *Jeopardy!*[3], in 2011. But, since then, much work has been put into Watson, which is now a system capable of a great number of tasks other than answering to clues. Nowadays, Watson is a Question Answering (QA) system that can answer questions made using natural language (English or Portuguese for instance), or questions made using speech [12, 25].

Despite its potential to learn about almost anything, Watson still has as basis several Natural Language Processing (NLP) components, besides the learning components that allow it to infer knowledge over data sources. These NLP components can also be implemented in a variety of open-source NLP toolkits, and so, a performance comparison can be made between Watson, and other toolkits available for free to the community. Until now, to the best of our knowledge, no comparison of this kind was ever made. Therefore, we propose a comparative study between Watson's performance in some NLP tasks, and the performance of another commonly used NLP toolkit over the same tasks.

1.2 Goals

As it was mentioned in the previous section, no comparison was made between IBM's system and other NLP toolkits. Because of this, the main goal of this work is to evaluate Watson's capabilities, in the field of NLP. To do this, three NLP tasks will be implemented using Watson's API. These tasks will also be implemented using another open-source NLP toolkit that is used commonly by the community. The goal is to compare Watson's performance with the performance achieved by the other chosen toolkit. The three chosen tasks to implement are: **sentiment analysis**, **NER**, and the implementation of a **natural language interface for database**.

[1] `https://www.ibm.com/pt-pt/` (last visited on 02/05/2018)
[2] `https://www.ibm.com/watson/` (last visited on 02/05/2018)
[3] `https://www.jeopardy.com/` (last visited on 02/05/2018)

To use Watson's API, IBM provides an online platform called **Bluemix**[4] , which will be used to develop part of the solution and will be addressed in more detail in Chapter 2. Due to the constant growth and expansion of NLP, there is a large number of open-source tools that developers can use to perform tasks of different levels of complexity. Considering a wide range of factors, like functionality offered, usage by the community, easiness to use, and others (full analysis made in Chapter 2), the toolkit chosen to implement the three tasks mentioned above is the **Natural Language Toolkit (NLTK)**. Once these tasks are implemented with both NLTK and IBM's online platform, the performance of all tasks will be tested, and a formal evaluation will be conducted, considering a series of factors, which will be discussed throughout Chapters 3, 4 and 5. This will let us know to some extent if Watson performs better than NLTK on the tasks chosen in this work. All tests were performed on a computer with an Intel Core i7-7500U processor (2.7 GHz), with 8 Gb of RAM, using an internet connection with 35 Mbps of download and upload speed.

1.3 Contributions and results

The main contributions of our work include:

- Three sentiment analysis systems (one for NLTK, two for Bluemix) capable of labeling movie reviews or tweets as positive or negative;
- Two systems (one for each toolkit) that can perform NER over text, and can identify a wide range of named entities, such as person names, locations, events, organizations, among others;
- Two natural language interfaces for database (one for each toolkit), that can answer questions regarding the cinema domain;
- An extensive comparative study between NLTK and Bluemix, not only for each task that was proposed, but also having in mind the results achieved for all three tasks.

1.4 Document outline

The rest of this document is organized as follows: in Chapter 2, we provide a more detailed description of Bluemix, along with the most used open-source NLP tools, and an analysis of what they have to offer and their usage nowadays. In Chapters 3, 4 and 5, we present our solution: the implementation of the three tasks proposed in this chapter, using both Bluemix and NLTK, and their respective evaluation results. Finally, in Chapter 6 we summarize the highlights of the work developed and present future work possibilities.

[4] `https://www.ibm.com/watson/products-services/` (last visited on 02/05/2018)

2

Related Work

In this chapter, we provide a description of Bluemix[1]. We also present a review of the existing Natural Language Processing (NLP) toolkits that are most used worlwide for educational or commercial purposes. In Section 2.1, the criteria used to separate which toolkits are of greater importance in the context of this work, and which are not, are presented. In Section 2.2 we present Bluemix in greater detail, describing how can users access the services offered, its main components, with the main focus going towards the services that use Watson[2]'s Application Programming Interface (API). In Section 2.3, we present several NLP toolkits, from which only one will be chosen to implement the tasks proposed in Chapter 1.

2.1 Criteria used to choose which toolkits to analyze

As it was said in Chapter 1, there are many NLP toolkits to choose from. But, in this chapter, we want to analyze toolkits that are widely used by the community for research, educational or even commercial purposes. Also, the focus of the toolkits discussed in this chapter must be related to the tasks proposed. For instance, toolkits that perform Machine Translation (MT), or audio recognition, will not be discussed here. Finally, the toolkits discussed must be open-source. The goal of this work is to evaluate the performance of toolkits available to the public in general, and not the performance of paid software. Therefore, all toolkits that do not meet the criteria mentioned before, will not be considered. Nevertheless, we present a brief review of them in Appendix A.

2.2 IBM Bluemix

IBM[3] has an online platform called Bluemix that is an implementation of IBM's Open Cloud Architecture, based on an open-source Platform-as-Service (PaaS) called Cloud Foundry[4]. With this platform, software developers can develop, manage and deploy web applications without needing to do all the

[1] `https://www.ibm.com/watson/products-services/` (last visited on 02/05/2018)
[2] `https://www.ibm.com/watson/` (last visited on 02/05/2018)
[3] `https://www.ibm.com/pt-pt/` (last visited on 02/05/2018)
[4] `https://www.ibm.com/cloud/cloud-foundry` (last visited on 02/05/2018)

underlying setup or worry about the infrastructure. Of the many available services in Bluemix, we are more interested in the ones that use the API of Watson, since it will be used to develop part of the goals proposed in Chapter 1. Watson's API provides the following services:

- **Discovery** - this service includes a content analytics and a cognitive search engine that can be bonded to other applications, to seek trends and patterns in data that allow users to make better quality decisions. The user can also use a simple query language to eliminate potential unwanted results;

- **Knowledge Catalog** - with this service, users can catalog and store data in a secure fashion, defining access policies for different types of users. In addition to this, this service displays insights about the data stored, making it easier to retrieve useful information about it;

- **Knowledge Studio** - this service consists of a web application that enables developers and domain experts to teach Watson to understand certain linguistic nuances in unstructured text. With it users can build custom models that combine annotation and model training, evaluation, and deployment;

- **Language Translator** - applications connected to this service can translate different types of text, in several languages, to many other languages. It can also identify written text from a vast number of languages;

- **Machine Learning** - with this service, users can make use of Watson's machine learning capabilities to train models, and then use these models to predict outcomes. This can done through a REST API, which can be called from a wide array of programming languages;

- **Natural Language Classifier** - this service is a classifier for strings. It returns the best matching class for a string or sentence. The user can submit questions and this service returns the best matches. For this to work, the classifier is fed a set of strings that represent certain entities, and a set of correct classes for each training string. After training with these training sets, the classifier can accept new strings or questions, and returns matches for them with a given probability value for each match;

- **Natural Language Understanding** - with this service, users can analyze text to extract meta-data from it, such as keywords, semantic roles, relations, named entities, emotions, and perform sentiment analysis. This is achieved by using Watson Knowledge Studio[5], which can identify domain specific named entities and relations between these entities in unstructured text. This service supports several different languages, although some languages do not support all functionality. For instance, to perform sentiment analysis, the languages supported are English, French, Arabic, German, Italian, Portuguese, Russian, and Spanish. But, for semantic role labelling, only English and Spanish are supported[6];

- **Personality Insights** - identifies psychological traits in people by analyzing social media and transactional data. These traits may be behavioural, intentions or purchase decisions. For example, with this service users can analyze tweets and draw conclusions about that person's personality (if the person is open, introvert, altruistic) and their values;

[5] `https://www.ibm.com/us-en/marketplace/supervised-machine-learning#product-header-top` (last visited on 02/05/2018)

[6] `https://console.bluemix.net/docs/services/natural-language-understanding/language-support.html#language-support` (last visited on 02/05/2018)

- **Speech to Text** - as the name suggests, this service converts human speech into written words. It can be used in a wide variety of scenarios, such as transcription of meetings, voice control of systems, basically any system that needs words written, instead of their spoken form. This transformation is done combining information about language structure and grammar, with information about audio signals, to generate an accurate transformation of speech to text. There are some languages to choose from, and these are US and UK English, Spanish, Mandarin, Arabic, Japanese, Brazilian Portuguese and French;

- **Text to Speech** - does the opposite of the previous service. It processes natural language to generate the appropriate synthesized speech. But, much like on the Speech to Text service, there are several languages available to choose regarding the audio output. There are male and female voices to choose from in some languages, others only have a female voice available. The languages included are English, from the UK and the US, French, German, Italian, Spanish (North American and Castilian), Brazilian Portuguese and Japanese;

- **Tone Analyzer** - this service applies techniques of linguistic analysis to identify 3 types of tones when people communicate (in this case, written text): emotion, like joy or sadness, language styles (confident or analytical for example) and social propensities, such as openness or introversion. This can be done either at document or sentence level, using cognitive linguistic analysis;

- **Visual Recognition** - this service analyses images seeking different content such as faces and objects with the help of a classifier, much like what the Natural Language Classifier service does. The user can choose the already created default classifiers, or build a custom one. This allows applications that use this service to analyze images or videos to understand what is going on and identify patterns;

- **Watson Assistant** - this service is used to build applications with a natural language interface, such as chatbots and virtual agents, that can communicate with future users of the application through any device or channel. The service can be trained using a web application, provided by IBM. With this web app, users can define which entities and intents they want Watson to recognize, and also build conversation flows that guide the dialog between the chatbot and its future users;

- **Watson Studio** - with Watson Studio, users can choose the tools they need to analyze and visualize data, to cleanse and shape it, or to create, train, and deploy machine learning models. This service is compatible with the Knowledge Catalog service, allowing users to shares resources and access policies between the two services.

Of all the services reviewed, those that are of greater interest to this work are the Watson Assistant service, the Natural Language Understanding service and the Natural Language Classifier service. The first service will be used during the task of implementing a natural language interface for database, since it can be used to recognize intents and entities mentioned by users, and the other two can be used to perform Named Entity Recognition (NER) and sentiment analysis. In depth descriptions of all services provided by Bluemix can be accessed here[7].

2.3 NLP toolkits

Regarding the field of NLP, which is the focus of this work, there are several toolkits available for download online, most of them for free. These tools are developed in many different programming languages and consist of programs, frameworks or libraries to build applications in this field of expertise.

[7] `https://console.ng.bluemix.net/catalog/` (last visited on 02/05/2018)

2.3.1 NLP toolkits offered by big companies

Microsoft Azure

Much like IBM with Bluemix, Microsoft has an online platform that offers a wide array of services for users to build applications, or integrate them in existing software. The platform is called Microsoft Azure[8]. The list of services offered is quite long (full list available in the "Pricing" section[9]), but, the services that are more important in the context of this work are those of the **Artificial Intelligence (AI) + Cognitive Services** category. This category offers services in the fields of image processing, speech recognition and processing, knowledge related services, such as an intelligent recommendation service, and, more importantly for this this study, NLP related services[10]. In this category, there are six services available, which will now be described in greater detail:

- **Language Understanding Intelligent Service (LUIS)**[11] - with this service, users can integrate in their conversation applications (chatbots or dialog systems), a component (the LUIS service) that can extract intents and entities from the sentences input. This is very similar to the Watson Assistant service, available on Bluemix. The developer must first define the intents that are relevant to detect, depending on the context on which the application is meant to be used. For each intent defined, the user must input some sentence examples, so that LUIS can identify intents with a higher degree of certainty. The user must also define the entities that are to be detected by the service, providing some examples for each entity in the process. In the end, LUIS outputs, in JSON format, the information extracted from the input sentence (the intents and entities detected, along with a confidence score for each one). This service supports several languages, such as English (US and EN), French, Spanish, Italian, German, Chinese, Japanese, Brazilian Portuguese, and Korean;

- **Bing Spell Check**[12] - this service lets users perform spell checks over the text input. It is based on a spell checking algorithm created by Bing[13], which is trained over a large corpus of documents and web searches. The algorithm can detect a wide array of spelling mistakes, even if a more informal language is used. This service comes with two spell checking modes:

  - Proof: This mode has the highest mistake detection rate, but it only supports US English;
  - Spell: The only mode available for languages other than US English. This mode detects the majority of spelling mistakes, but misses some of the mistakes detected on Proof mode, such as words being repeated.

- **Web Language Model**[14] - based on N-grams and Markov models, with this service developers can perform four operations, regarding words in English from the US:

  - Break strings that contain no spaces, into words;
  - Calculate the $\log_{10}$ likelihood of a given sequence of words;
  - Calculate the $\log_{10}$ probability of a word occurring, given a sequence of previous words;

[8] `https://azure.microsoft.com/pt-pt/` (last visited on 02/05/2018)
[9] `https://azure.microsoft.com/pt-pt/pricing/` (last visited on 02/05/2018)
[10] `https://azure.microsoft.com/pt-pt/services/cognitive-services/` (last visited on 02/05/2018)
[11] `https://azure.microsoft.com/pt-pt/services/cognitive-services/language-understanding-intelligent-service/` (last visited on 02/05/2018)
[12] `https://azure.microsoft.com/pt-pt/services/cognitive-services/spell-check/` (last visited on 02/05/2018)
[13] `https://www.bing.com/` (last visited on 02/05/2018)
[14] `https://azure.microsoft.com/pt-pt/services/cognitive-services/web-language-model/` (last visited on 02/05/2018)

- Compute the words that are most likely to follow a given sequence of words.

- **Text Analytics**[15] - this service provides implementations of some useful NLP tasks, like sentiment analysis, key phrase extraction, topic and language detection. In sentiment analysis, the service outputs, for a text, a sentiment score between zero and one, being zero a very negative text, and one a very positive text. The key phrase extraction component extracts the most important words of sentences, their talking points. The languages supported by this service vary, according to the task the user wants to perform. For instance, for topic detection, only English is supported, but for language detection, there are 120 supported languages. Spanish and English are both supported in the tasks of sentiment analysis and key phrase extraction, but the former also supports Portuguese and French, and the latter additionally supports German and Japanese;

- **Translator Text**[16] - this a MT service that supports translation from and to more than fifty languages.

- **Linguistic Analysis**[17] - with this service, users can embed in their applications some other core NLP tasks, such as tokenization, sentence splitting, Part of Speech (PoS) tagging, and constituency parsing.

Google Cloud Natural Language API

Similarly to Microsoft (although with less services) Google has an online platform for the development of software in the field of NLP, called Google Cloud Natural Language API[18]. This platform offers services that allow users to perform some basic NLP tasks, such as tokenization, sentence splitting, PoS tagging, dependency parsing, sentiment analysis and NER. Several languages are supported by this platform, depending on the tasks the user wants to perform, such as English, French, Portuguese, German, Italian, Spanish, Japanese, Korean and Chinese. Also, there are APIs available for multiple programming languages, like Java, Python, Ruby, C#, Go[19] and Node.js.

2.3.2 Most popular open-source NLP toolkits

Natural Language Toolkit (NLTK)

NLTK[20] is a collection of open-source programs, modules and data sets written in Python, originally created in the Department of Computer and Information Science at the University of Pennsylvania. It can run on all platforms that support Python [4, 21]. Nowadays, NLTK has been used widely as a teaching aid, or as an individual learning tool for students and researchers in the field of NLP [4]. It comes with over 50 corpora and some other lexical resources, such as Wordnet[21], a database for relations between words. Also, it possesses modules to perform several core NLP tasks, such as tokenization, tagging, stemming, parsing, and many others, as it can be seen in Figure 2.1. Besides the modules described in Figure 2.1, there are others that are worth describing, because they can be important in the process of implementing NLP tasks:

[15] `https://azure.microsoft.com/pt-pt/services/cognitive-services/text-analytics/` (last visited on 02/05/2018)
[16] `https://azure.microsoft.com/pt-pt/services/cognitive-services/translator-text-api/` (last visited on 02/05/2018)
[17] `https://azure.microsoft.com/pt-pt/services/cognitive-services/linguistic-analysis-api/` (last visited on 02/05/2018)
[18] `https://cloud.google.com/natural-language/` (last visited on 02/05/2018)
[19] `https://golang.org/` (last visited on 02/05/2018)
[20] `http://www.nltk.org/` (last visited on 02/05/2018)
[21] `https://wordnet.princeton.edu/` (last visited on 02/05/2018)

Language processing task	NLTK modules	Functionality
Accessing corpora	nltk.corpus	standardized interfaces to corpora and lexicons
String processing	nltk.tokenize, nltk.stem	tokenizers, sentence tokenizers, stemmers
Collocation discovery	nltk.collocations	t-test, chi-squared, point-wise mutual information
Part-of-speech tagging	nltk.tag	n-gram, backoff, Brill, HMM, TnT
Classification	nltk.classify, nltk.cluster	decision tree, maximum entropy, naive Bayes, EM, k-means
Chunking	nltk.chunk	regular expression, n-gram, named-entity
Parsing	nltk.parse	chart, feature-based, unification, probabilistic, dependency
Semantic interpretation	nltk.sem, nltk.inference	lambda calculus, first-order logic, model checking
Evaluation metrics	nltk.metrics	precision, recall, agreement coefficients
Probability and estimation	nltk.probability	frequency distributions, smoothed probability distributions
Applications	nltk.app, nltk.chat	graphical concordancer, parsers, WordNet browser, chatbots
Linguistic fieldwork	nltk.toolbox	manipulate data in SIL Toolbox format

Figure 2.1: Tasks and corresponding modules in NLTK [20]

- **Visualization** - Provides a way for users to perform experiments, manipulate and view data structures in a graphical fashion. The structures that can be viewed include trees, chart parsers, mathematical functions, and finite state automata [4]. This allows users to have a clearer idea of the results of the tasks they perform;

- **Type checking** - This module explicitly checks if the arguments passed to functions are of the correct type. This is done for all basic data types and structures [4], and can save some debugging time regarding this specific type of bug. Since this is done explicitly, type checking can slow down the execution of tasks. But, this feature can be turned off, leading to a slight performance increase.

The developers of NLTK chose Python as the language of implementation for several reasons: because of its shallow learning curve, even inexperienced students and programmers can learn quickly how to implement simple NLP tasks [21]; Other main factors for the choice of this language include its clear syntax and semantics, the easiness to write structured programs in an object-oriented paradigm, and a short development/test cycle.

NLTK is composed of several modules, each one of them regarding an explicit NLP task, or data structure. A detailed explanation of all the modules and their functions, can be found on the tool's website[22]. With all the modules and vast array of functionality available, users can develop all sorts systems with this toolkit, by either using the available modules, extending them, or even creating new ones.

[22] `http://www.nltk.org/api/nltk.html` (last visited on 02/05/2018)

Stanford CoreNLP

Stanford CoreNLP[23] is a Java toolkit developed by the Natural Language Processing Group at Stanford University[24] . Originally meant to be used with Java, the toolkit has been adapted to be used with other programming languages, such as Python, Perl, Ruby, Scala, node.js, Clojure, and .NET. CoreNLP is one the most used NLP toolkits, being it for educational and research purposes, or for commercial and government applications [23]. It provides implementations of several core NLP tasks, which will be described in more detail shortly. These tasks include tokenization, PoS tagging, sentence splitting, NER, stemming, coreference resolution, syntactic parsing, gender identification, and sentiment analysis.

An annotation pipeline framework was adopted in this toolkit. Text in natural language is put into a so called Annotation object, which stores analysis information belonging to a text, along with the original text. Then, a sequence of analyses is performed using different annotator objects (all annotator objects implement the same interface, which allows different annotators to add information using the same methods), adding additional information to that object along the way. Finally, all the information gathered can be output as plain text, or in Extensible Mark-up Language (XML) format.

This toolkit can be used as an API, developing applications through API calls, or using the provided command line interface. The provided API is simple and easy to learn. This constitutes one of the big advantages of CoreNLP, in relation to bigger and more complex frameworks. It supports languages other than English, such as Chinese, German, French and Arabic, but English, as expected, has the most tools available to use out of all languages. Figure 2.2 shows, as of 2014, the different NLP tasks one can perform across all languages supported.

Annotator	Arabic	Chinese	English	French	German
Tokenize	✓	✓	✓	✓	✓
Sent. split	✓	✓	✓	✓	✓
Truecase			✓		
POS	✓	✓	✓	✓	✓
Lemma			✓		
Gender			✓		
NER		✓	✓		✓
RegexNER	✓	✓	✓	✓	✓
Parse	✓	✓	✓	✓	✓
Dep. Parse		✓	✓		
Sentiment			✓		
Coref.			✓		

Figure 2.2: Tasks and supported languages by Stanford CoreNLP [23]

Each task is embedded in an annotator, and some of these use models are trained from annotated corpora using machine learning, and others are rule based components. For English, these are the annotators that implement the most important functionality of this toolkit:

- **Tokenize** - transforms a text into a sequence of tokens, saving the character offsets of each token present in the original text;

[23] `https://stanfordnlp.github.io/CoreNLP/` (last visited on 02/05/2018)
[24] `https://nlp.stanford.edu/` (last visited on 02/05/2018)

- **ssplit** - takes as input a sequence of tokens, and separates it into sentences;
- **pos** - tags tokens with their most likely PoS tag, through the use of a maximum entropy model;
- **lemma** - lemmatizes the tokens provided as input;
- **gender** - adds information about the most likely gender of names;
- **ner** - recognizes named entities, such as persons, organizations, locations. It also recognizes numerical entities, like numbers, dates, monetary quantities, time, sets, and durations;
- **regexner** - comes with a rule-based named entity recognition system, based on Java regular expressions. It allows users to add new entity labels that are not annotated in corpora originally. The default annotator comes with regular expressions that can recognize religions, titles, nationalities and ideologies;
- **parse** - performs syntactic analysis of sentences, based on a probabilistic parser (Klein and Manning, 2003; de Marneffe et al., 2006). It outputs both dependencies and constituents;
- **sentiment** - performs sentiment analysis over sentences, using a deep learning model (Socher et al., 2013). A binary tree is produced for each sentence, and for each node of that tree, a sentiment score is assigned;
- **dcoref** - this annotator is used to perform mention detection, and coreference resolution. The coreference graph of a given text is stored in its Annotation object.

An additional detail is that all the annotators mentioned have some options, which the users can change. This can be done adding them to a Properties object. It is also possible for users to develop new annotators, if the user wants to perform a task not present in the original toolkit. The toolkit can be downloaded from here[25], where there are download links for the multiple languages supported.

OpenNLP

OpenNLP[26] is a free Java library, based on Machine Learning (ML), meant to process text written in natural language. Developed by the Apache Software Foundation[27], this toolkit supports several NLP tasks, like sentence segmentation, PoS tagging, tokenization, NER, parsing, chunking and coreference resolution[28]. This toolkit also includes maximum entropy models, perceptron based machine learning, some pre-built models for a wide array of languages (they can be downloaded here[29]), in addition to annotated text from which these models were derived from. OpenNLP is composed of several components, each one of them enabling the user to perform a certain NLP task, to train a model and test it. This can be done using Java API calls, or through a provided command line interface. Although tasks can vary greatly between them, the code to load the models needed by different tasks is quite similar. To do this users can make similar API calls, providing a model and an input.

Once this is done, the tool can be instantiated, and the task can be executed. Depending on the task being executed, the input and output formats can vary, but, most of the times, the input is a String or an array of it, and the output is an array of String. As it has been stated before, there are many NLP tasks that can be done with this toolkit, and now some of the components needed to perform these tasks will be described with a little more detail:

[25] `https://stanfordnlp.github.io/CoreNLP/` (last visited on 02/05/2018)
[26] `https://opennlp.apache.org/` (last visited on 02/05/2018)
[27] `https://www.apache.org/` (last visited on 02/05/2018)
[28] `https://opennlp.apache.org/docs/1.7.2/manual/opennlp.html` (last visited on 02/05/2018)
[29] `http://opennlp.sourceforge.net/models-1.5/` (last visited on 02/05/2018)

- **Sentence Detection** - detects punctuation characters that mark the end of a sentence, splitting the input text into sentences. Normally, this is done before tokenizing, but it is also possible to tokenize text first, and then split it into sentences. Most components expect text already split into sentences, which makes this a key component to use in almost every task;
- **Tokenizer** - sections input character sequences into tokens, which are, most of the time, numbers, words, or punctuation. OpenNLP offers three implementations of this tool:
  - Whitespace Tokenizer: identifies non-whitespace sequences as tokens;
  - Simple Tokenizer: identifies sequences of the same character class as tokens;
  - Learnable Tokenizer: uses a maximum entropy model to detect token boundaries based on probabilities.
- **Lemmatizer** - given a token and its PoS tag, it returns the lemma of the word that the token refers to. Without the PoS tag, a token could have multiple lemmas, depending on the context in which the word is used. So, to perform successful stemming, both the tag and token are needed;
- **Name Finder** - used to perform detection of named entities and numbers;
- **PoS tagging Tagger** - Assigns tokens their matching PoS tag. Due to the ambiguity present in texts, a token can have multiple PoS tagging tags. To tackle this problem, a probability model is used, to decide out of the possible tags that can be assigned, the out that is most suitable;
- **Chunker** - divides text into parts that are related syntactically (chunks), like verb or noun groups;
- **Parser** - given a sentence, parses the sentence and returns its parse tree in a printed form, not literally in the form of a tree, but in a sequence of constituent parts.

It is also important to mention that this toolkit can use corpora to train and test components, such as the Leipzig Corpora (a collection of corpora in different languages), the Portuguese corpora available here[30], and some data collections used on Conference on Computational Natural Language Learning (CoNLL). Finally, OpenNLP is available for download here[31].

2.3.3 Other open-source NLP toolkits

CogCompNLP

CogCompNLP[32] is a collection of Java libraries for several core NLP tasks, developed by the Cognitive Computation Group, of the University of Illinois [7]. With this toolkit, users can perform several NLP tasks, such as PoS tagging, chunking, tokenization, stemming, NER, constituency and dependency parsing, and semantic role labelling. The Core Utilities library provides data structure definitions and other useful tools for developing NLP applications or to run experiments. The complete list of data structures and usage examples is available here[33]. The Corpus Reader library has implementations of classes that can read corpora and transform them into the data structures defined on the Core Utilities library [7]. The Edison library performs feature extraction through a framework that uses the data structured from Core Utilities. The toolkit's source code can be cloned from Github[34].

[30] `http://www.linguateca.pt/` (last visited on 02/05/2018)
[31] `https://opennlp.apache.org/download.html` (last visited on 02/05/2018)
[32] `https://github.com/CogComp/cogcomp-nlp` (last visited on 02/05/2018)
[33] `https://github.com/CogComp/cogcomp-nlp/blob/master/core-utilities/README.md` (last visited on 02/05/2018)
[34] `https://github.com/CogComp/cogcomp-nlp` (last visited on 02/05/2018)

DELPH-IN

Short for Deep Linguistic Processing with Head-Driven Phrase Structure Grammar (HPSG), DELPH-IN[35] consists of a set of tools that are meant for deep linguistic processing of human language. It is the result of joint work from computational linguistics from institutions all over the world and is composed by several components of different complexities [36], such as deep grammars and deep parsers, for various natural languages and tools for shallower language processing [35]. The goal of this toolkit is to combine statistical and linguistic processing techniques to understand the meaning of texts and utterances. Also, this set of tools adopts two models of advanced formal linguistic analysis: Minimal Recursion Semantics (MRS) and HPSG, already mentioned previously. As with the other tools discussed before, it is open-source and free of charge. More detail about software, grammars and tree-banks, is available on the tool's website.

Freeling

Freeling[36] is an open source C++ library developed by researchers at the TALP Research Center, in Universitat Politecnica de Catalunya. This toolkit provides several language analysis tools, such as PoS tagging, semantic role labelling, parsing, morphological analysis, and many others. These tools can be used in many different languages such as Portuguese, Spanish, English, French, German, Italian and many others [27, 34]. Originally built in C++, this tool is available to the public in APIs which can be downloaded from its website. There are APIs in C++, the most complete of the ones available since it is the language used to develop FreeLing, but also in Java, Perl[37], Python, PHP and Ruby, which may not have all functionality available [34]. Developers can use the default linguistic tools available, like grammars and dictionaries for example, but these can be extended or adapted for specific scenarios, or even built from scratch for some new language. This can be done because the library is divided in many classes, each one capable of performing different types of analysis, so the user can select which classes to use or adapt and add new classes, with new functionality [27].

General Architecture for Text Engineering (GATE)

GATE[38] is an environment for graphical development, developed in the University of Shefield. With this environment users can build and deploy language engineering components to tackle various language processing tasks, reuse and reconfigure already existing components in new systems, and test and evaluate the systems developed [9, 10]. With this framework, users can develop systems with a wide range of functionalities, such as Information Extraction (IE), Information Retrieval (IR) or MT systems [10]. Developers have in this environment the ability to combine different tools and linguistic databases, and with this, launch several processes on the same text, or on different texts, and compare the similarities and differences between them, using, for example, different taggers or parsers across the different processes [10]. This framework is composed of three main components. The first one is a database that stores information about texts, its schema follows an object-oriented model of information about texts (called GATE Document Manager). The second main component, the GATE Graphical Interface, is a graphical interface that allows users to launch processing tools on data, and later visualize and evaluate the results. The third and last component is a set of wrappers that communicate with the interface and the database. These wrappers create a Collection of Reusable Objects for Language Engineering (CREOLE), and all systems developed in GATE rely on the use CREOLE objects. Developers can download GATE for free from the tool's website.

[35] `http://www.delph-in.net/wiki/index.php/Home` (last visited on 02/05/2018)
[36] `http://nlp.lsi.upc.edu/freeling/node/1` (last visited on 02/05/2018)
[37] `https://www.perl.org/` (last visited on 02/05/2018)
[38] `http://gate.ac.uk.` (last visited on 02/05/2018)

Gensim

Gensim[39] is a free open-source Python library, originally developed by Radim Řehůřek, that can be used to develop software in the fields of IR and NLP. With the help of Gensim, users can develop applications for document indexing, topic modelling and similarity retrieval with large corpora [28]. Since it is developed in Python, and relies only on Scipy and NumPy, it can be used on any operating system, so long as it has installed Python and both additional libraries that have been mentioned. Gensim has multicore implementations of several algorithms, such as word2vec, Hierarchical Dirichlet Process, LDA and LSA. These algorithms are very efficient because they use the scientific libraries, which possess optimized implementations for all scientific calculations. Also, it is important to note that LDA and LSA can be run either locally (one machine) or on a cluster of computers. Gensim can be downloaded from here[40].

LinguaStream

Developed at Université de Caen, LinguaStream is a free Integrated Development Environment (IDE) for users to develop applications in the field of NLP, and conduct experiments with [3]. This is done by progressive enrichment of digital documents, in which users add steps (or components) to a processing stream, and each step produces new output (annotations in the document to be analysed) that can be used in future steps of the stream, or even on another stream [2]. There are many steps from which the user can choose from (about fifty), contemplating different stages of linguistic analysis, such as lexical, syntactic, semantic or discourse analysis. Each step has some parameters that can be changed, allowing the user to change the behaviour of that component, and a set of input/output sockets that are used to connect components. Because of this, users can create long processing streams in a very simple fashion. Due to LinguaStream's graphical interface, users can create and test streams visually [2]. Using a Java API, templates and a macro-component system, users can add new components to be used in streams. The API also allows foreign applications to use components from this tool.

MontyLingua

MontyLingua[41] is a free NLP toolkit, designed for the English language, developed by Hugo Liu at Massachusetts Institute of Technology(MIT). The user passes text as input and it performs a semantic analysis of the text, outputting that analysis. It extracts key entities from sentences, such as the subject, verb, adjectives, objects, dates, places, amongst other relevant semantic information. This is useful for some fields, like IE, Question Answering (QA) and request processing. This toolkit can be used without any additional coding or data training. The user just has to input the text and it will output all the semantic information. Montylingua comes with a built in common sense knowledge database (called Open Mind Common Sense [30]), which allows it to discard any semantic interpretations that do not make sense. This toolkit has five major components relevant in the context of this analysis: MontyTokenizer splits sentences into tokens and resolves contractions (you're and you are, for instance) ; MontyTagger performs PoS tagging, using the common sense database to discard possibilities that do not make sense; MontyChuncker chuncks texts, using as basis regular expressions; MontyExtractor extracts subject, verb, object tuples and phrases from sentences; Finnaly, MontyLemmatiser, as the name sugests, lemmatizes words in sentences. MontyLingua can be downloaded from here[42].

[39] `https://radimrehurek.com/gensim/` (last visited on 02/05/2018)
[40] `https://pypi.python.org/pypi/gensim#downloads` (last visited on 02/05/2018)
[41] `http://alumni.media.mit.edu/~hugo/montylingua/` (last visited on 02/05/2018)
[42] `http://alumni.media.mit.edu/~hugo/montylingua/#download` (last visited on 02/05/2018)

SpaCy

Developed by Explosion AI[43], SpaCy[44] is an open-source Python library meant for advanced NLP tasks. It is written in Python and Cython[45], and can be used on any system that supports Python. It is said to offer the fastest syntactic parser available ever implemented, with its accuracy just 1% off the best accuracy ever achieved by other parsers. This was confirmed by two peer-reviewed papers in 2015 [6, 15]. Systems with such a high degree of accuracy normally are slower than SpaCy by some degrees of magnitude (20 times slower)[46]. Besides this, SpaCy offers more NLP related tasks, such as tokenization, PoS tagging, dependency parsing, NER, and sentence segmentation. Currently this library supports English, French and German, although work is being done to add support to other languages, like Chinese, Italian, Spanish, Portuguese, among others. As stated before and unlike the previous tools described, SpaCy is meant to be used for software production, and actual real-life situations, contrary to the research and educational purposes of most other tools. But, it is an open-source library, and can be installed through the command line, with the `pip` command.

2.4 Overview of the NLP toolkits discussed

In this section, we present an overview of all the toolkits discussed throughout this chapter, regarding four aspects: functionality offered, whether the toolkit is paid or not, the supported languages, and the date of its last update.

Toolkits	**Functionality offered**	**Paid**	**Supported languages**	**Date of last update**
Stanford CoreNLP	Several basic NLP tasks, like tokenization, stemming, sentence splitting, PoS tagging, NER, syntactic parsing, sentiment analysis, coreference resolution, and gender association regarding names	No	English, Chinese, Arabic, German, and French	February 2018
Natural Language Toolkit (NLTK)	Wide range of core NLP tasks, such as stemming, parsing, tokenization, sentiment analysis, simple chatbots, implementations of some evaluation metrics (precision, recall), and implementations of some ML algorithms	No	Several languages other than English, depending on the task being performed (clear supported language list not found)	September 2017
OpenNLP	NLP tasks such as tokenization, stemming, sentence segmentation, PoS tagging, coreference resolution, chunking, NER. Also includes implementations of several ML algorithms and models (maximum entropy models, perceptrons)	No	Languages supported depend on the task being executed, but English, German, Danish, Spanish, Portuguese, and Dutch are supported	December 2017

Table 2.1: Comparison between the several NLP toolkits discussed - part 1

[43] `https://explosion.ai/` (last visited on 02/05/2018)
[44] `https://spacy.io/` (last visited on 02/05/2018)
[45] `http://cython.org/` (last visited on 02/05/2018)
[46] `https://spacy.io/docs/api/` (last visited on 02/05/2018)

Toolkits	Functionality offered	Paid	Supported languages	Date of last update
CogCompNLP	Core NLP tasks: tokenization, chuncking, PoS tagging, NER, stemming, dependency parsing, semantic role labelling	No	English	February 2018
DELPH-IN	Tools for deep linguistic analysis of human language, includes parsing and generation of parse trees	No	English, Japanese, German, Spanish, Portuguese, Korean, Norwegian	(Not clear from website)
Freeling	PoS tagging, morphological analysis, NER, semantic role labelling, word sense disambiguation	No	English, Spanish, Italian, Portuguese, French, German, Russian, Catalan, among others	May 2016
GATE	Graphical environment for development of NLP tasks in the fields of Information Extraction (IE), Information Retrieval (IR), Machine Translation (MT), such as document comparison	No	English	June 2017
Gensim	Similarity retrieval with large corpora, document indexing, topic modelling. Implementations of LDA, LSA, word2vec, Hierarchical Dirichlet Process	No	English	March 2018
Google Cloud Natural Language API	Basic NLP tasks, like sentence splitting, tokenization, dependency parsing, PoS tagging, sentiment analysis, and NER	No	Portuguese, English, French, German, Italian, Spanish, Japanese, Chinese, and Korean	(Could not find that information)
LinguaStream	IDE for development of NLP tasks more related to semantics, such as lexical, semantic, and discourse analysis	No	English, French	(Could not find that information)
Microsoft Azure	Conversation applications, spell checking, sentiment analysis, topic and language detection, MT, calculate the probability of sequences of words occurring	No	Languages supported vary, depending on the services used. In topic detection, over 120 languages are supported, in MT over 50 languages are supported, but in all services, English is supported	(Could not find that information)
MontyLingua	Extraction of key entities from sentences, like the subject, verb, objects, dates, places. Also has components to perform PoS tagging, lemmatizing and tokenizing	No	English	August 2004
SpaCy	Advanced NLP tasks, but also some basic NLP tasks, such as syntactic parsing, PoS tagging, tokenization, sentence recognition, dependency parsing, and NER	No	English, German and French	April 2018

Table 2.2: Comparison between the several NLP toolkits discussed - part 2

2.5 Choice of NLP toolkit to use

In light of all the requirements mentioned in Section 2.1, some toolkits were immediately discarded, either because they are paid toolkits, or because their focus is on other specific fields, less related to the tasks proposed in Chapter 1. Regarding natural language interfaces for database, none of the toolkits analyzed in this chapter comes with a module designed for the development of such systems. NLTK is the only toolkit that comes with a chatbot module, but it is not usable to implement a natural language interface for database. Therefore, an important factor to take into account, is if the toolkit offers enough functionality to develop natural language interfaces for database. Another aspect that influences the decision, is how up to date are the toolkits. Software that is not updated in a long time (3 or 4 years) is not desirable. We want to use tools that are recent, that are being updated regularly, which means (most of the time) they are still in use by the community.

Therefore, looking at the tables in Section 2.4, many toolkits can be ruled out very quickly. Due to the fact that the date of its latest update was not found, DELPH-IN will not be used. MontyLingua has not been updated for more than ten years, and since we want to use recent software, this toolkit is also ruled out. Gensim does not offer all the functionality needed to implement the three tasks proposed, much like Freeling and LinguaStream, so these three are also put aside.

Popularity is also an important factor to take into account in our decision. After consulting several people in the field of NLP, and searching online for papers that use, or make reference to the toolkits discussed in this chapter, the toolkits that stand out the most are, **NLTK** and **Stanford CoreNLP**. Although Stanford CoreNLP does not have a specific module dedicated to natural language interfaces, which can be built from other existing modules, these two toolkits offer all the functionality needed to implement the three tasks proposed.

Since Microsft's platform does not offer a NER service, it will not be used in this study. OpenNLP does not offer a module dedicated to sentiment analysis, and because of that it will also be ruled out. Besides OpenNLP, only some of the less popular ones remain as a valid choice. Looking once again at the tables in Section 2.4, it can be seen that CogCompNLP does not offer a sentiment analysis module, which is also true for SpaCy. Looking at the date of its last update (April 2018), it can be seen that it is in constant development and improvement, and is currently used by the NLP community, although not as much as the other two toolkits chosen. This makes SpaCy also a viable option for the choice of which toolkit to use, although it does not possess a module dedicated to sentiment analysis. This leaves us with three remaining candidates that standout from the rest of the toolkits: **NLTK**, **Stanford CoreNLP** and **SpaCy**. Out of these three, the final choice ended up falling upon NLTK, due to its higher degree of popularity among researchers and students in this field, and also because it provides all the tools necessary to implement the three chosen tasks.

3

Sentiment Analysis

3.1 Overview

Generically, sentiment analysis consists of analyzing the polarity of texts [8]. In other words, the goal is to see if the texts transmit a more positive sentiment (a positive movie review, for instance), or a more negative one (a negative movie review). Because of this, this task can be very useful in plenty of scenarios. In this study, we will be performing sentiment analysis over texts that are reviews, or opinions, about movies, and over social media texts (tweets). It will output a sentiment score for the analyzed text, which can be positive, negative, or neutral (being 1 the most positive score, -1 the most negative, and 0 a neutral score, usually).

To implement this task in Bluemix, the **Natural Language Understanding (NLU)** and **Natural Language Classifier (NLC)** services will be used, leading to two separate sentiment analysis systems, each using one of Bluemix's services. The former offers sentiment analysis of texts coming from different sources, including plain text, HyperText Markup Language (HTML) source code, or an URL. The user can also choose from a wide array of languages[1], such as English, Portuguese, French, German, Italian, Russian, Spanish, and Arabic. To choose a specific language, the user must only change the language code passed to Bluemix, no additional training is needed. The latter, as the name suggests, is based on a classifier, meaning that training data had to be passed on to the service. NLTK comes with an already built model to perform sentiment analysis over texts in English, based on the VADER model [16]. By default, this toolkit only supports the English language, but it can be trained with datasets in other languages, if the user wants to analyze texts that are not written in English. But, in our implementation of this task, we will not be using the VADER model, as it will be seen further ahead.

Performing sentiment analysis using NLTK requires a classifier, and so, a Naive Bayes classifier was chosen. Additionally, four different types of word features were used (presented in Section 3.3), each with a different degree of complexity:

[1] `https://www.ibm.com/watson/developercloud/natural-language-understanding/api/v1/#sentiment` (last visited on 02/05/2018)

- **Baseline**;
- **Stopword removing**;
- **Most significant bigrams**;
- **Most significant bigrams considering only the most common words**.

3.2 Corpora Selection

We chose to use corpora that is already part of NLTK[2]'s vast pool of NLP related resources, and are specifically made for the task of sentiment analysis. Two different corpora were used for this task:

- **The movie review corpus** - A corpus with movie reviews, labeled "positive" or "negative". It consists of 1000 positive reviews and 1000 negative reviews;
- **The Twitter corpus** - A corpus that consists of a set of tweets taken from Twitter[3] interactions. Much like the previous corpus, each tweet is labeled "positive" or "negative". There are 5000 positive tweets and 5000 negative tweets.

These two corpora seemed appropriate to the task at hand, due to their domains being a common target of sentiment analysis systems. They were used both on Bluemix and on NLTK. In the two following subsections we will provide greater detail for the implementation of this task in each toolkit.

3.3 NLTK implementation

There are several examples of sentiment analysis programs online, so it was not hard to figure out what had to be done to tackle this task. The choice of corpora to use was already done, leaving us with the choice of which classifier to use. There were some options to choose from, such as a Naive Bayes, Maximum Entropy, or a Decision Tree classifier[4]. We chose to use the **Naive Bayes** classifier, due to its simplicity.

This classifier, along with all other classifiers that come with NLTK, accepts as input a set of features for training or classifying purposes. These consist of a dictionary with all the words or n-ngrams in the data sample to be passed as input. Using different word features to train the classifier, users can obtain varying levels of performance, depending on, for instance, whether all words were used, or only some of the most common words. In our case, several word feature generating functions were implemented:

- **Baseline** - as the name implies, this one is the most simple. It generates word features using all the words in each sample (only unigrams), with no additional filtering;
- **Stopword removal** - does the same as the previous function, but filters out stopwords. The list of stopwords for the English language is already provided by NLTK, but the words "no", "nor" and "not" were excluded from this set, since they can be significant for this particular task;
- **Most significant bigrams** - this function uses NLTK's built-in chi-square scoring function to score bigrams in the text provided as input. With the movie review corpus, only the 200 most significant bigrams for each review are considered. For the Twitter corpus, all bigrams are considered. Due to their small size, it does not make sense to discard any bigram from a tweet;

[2] `http://www.nltk.org` (last visited on 02/05/2018)
[3] `https://twitter.com/?lang=pt_pt` (last visited on 02/05/2018)
[4] `http://www.nltk.org/howto/classify.html` (last visited on 02/05/2018)

- **Most significant bigrams among the most common words** - In addition to what the previous function does, this function only considers the 10000 most common words used in the training set, for the computation of the 200 most significant bigrams across the training set, unlike the previous function, that returned the most significant bigrams for each review.

Since our goal with this task is to compare the performance of NLTK and Bluemix in the execution of sentiment analysis, we did not focus in performing very complex feature engineering. Instead, we focused on implementing simple, but effective features, which will allow us to perform this comparison. Implementing the first three functions was straightforward, despite the appearance of some problems regarding file encodings, which took some time to fix. The last function required some additional pre-processing of the training part of the corpus. This involved counting words and selecting the ones that were most common. After all the features are generated, the training features are passed to the classifier, along with the test features afterwords. The testing methodology adopted will be described in further detail in Section 3.5.

3.4 Bluemix implementation

Before getting into more detail, it is worth mentioning that IBM[5] does not specify which classifier is used in either the NLU service or the NLC service. Additionally, both services only classify one string at a time. Because of this, the execution time using these services is quite large, as it will be analyzed in further detail in Section 3.5.

Since the NLU service does not require training, implementing this task using this service was straightforward. The first step was to retrieve the raw text from both corpora, which is what the service accepts as input. This involved some manipulation of the Twitter corpus, due to the fact that it is stored in JavaScript Object Notation (JSON), and it is easier to work with it if is in a Comma-Separated Values (CSV) file. But this was bypassed using NLTK's built-in functions to manipulate corpora. After having all the input ready, we passed all the strings (individual reviews or tweets) to the service, and registered the top classes indicated in the response. Finally, after having analyzed all the input, we calculated all the metrics used to measure the system's performance.

Regarding the implementation of this task using the **NLC** service, it can be said that it was of a much higher degree of complexity, in comparison to the NLU service. There are some important aspects to note regarding this service:

- **Input length restrictions** - this service only accepts as input, either for training or classifying purposes, strings with a maximum of 1024 characters[6]. Since that on the movie corpus, only less than 15 reviews have under 1024 characters, this corpus cannot be used on this service;
- **Training data restrictions** - the training data must be passed to service in the form of a Comma-Separated Values (CSV) file, and with a large number of restrictions[7] to the format of the file.

Creating the training file according to these restrictions took some time, mainly because of the large size of the corpus used. Also, much like what happens with the NLU service regarding execution time, the classifier takes a quite a while to train (depending on the number of records in the CSV file) and to

[5] `https://www.ibm.com/pt-pt/` (last visited on 02/05/2018)

[6] `https://console.bluemix.net/docs/services/natural-language-classifier/using-your-data.html#data-preparation` (last visited on 06/05/2018)

[7] `https://console.bluemix.net/docs/services/natural-language-classifier/using-your-data.html#using-your-own-data` (last visited on 06/05/2018)

classify the corpus.

The execution flow consists of several steps. The first one is the same as with the NLU service: retrieve the raw text from the Twitter corpus, which was done with the help of some of NLTK's corpora manipulating functions. After that, the training CSV file is produced, according to all of the restrictions mentioned. Having produced the file, it is then passed on to the service, which in turn will train the classifier. Finally, after the classifier has passed the training stage, the remainder of the corpus is passed as input, one tweet at a time, and all the performance metrics all calculated when the classifier has no more tweets to classify.

3.5 Evaluation

3.5.1 Methodology

In order to evaluate the developed systems, we used an evaluation methodology that leverages, not only on the performance of the systems when performing the NLP tasks considered, but also a series of other factors related to ease of use and functionality. To measure the systems' performance, precision, recall, f-measure, and accuracy will be used. Precision measures the ratio of results with positive sentiment that are returned correctly, and is defined as:

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

TP is the number of true positives (the results returned correctly with positive sentiment), and FP is the number of false positives (the results that were identified as having positive sentiment, but not should have been). Recall measures the ratio of true positives identified, and is defined as:

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

FN is the number of false negatives (the results that were not identified as having positive sentiment, but should have been). The f-measure consists of the harmonic mean of precision and recall, and is defined as:

$$F\text{-}measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Finally, accuracy measures the percentage of results returned correctly, and is defined as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

The other factors that will be taken into account in our evaluation, are more related to the development stage of NLP tasks:

- **Difficulty installing the toolkit** - before any development starts, users must install on their machines the toolkit they will be using. If this is a time consuming and complex activity, using the toolkit in question is less appealing, so this factor must be taken into account in our evaluation methodology;
- **Documentation** - users, unless they already had previous experience with the tools, most likely do not know how the toolkits work, and what has to be done to perform a certain Natural Language Processing (NLP) task. In these cases, most times developers recur to the documentation

available. In this sense, another important factor when evaluating a toolkit, should be whether the available documentation is clear and easy to read, or not;

- **Community support** - when users encounter difficulties developing their application, besides the documentation available, they resort to the Internet to find a solution, visiting programming related websites (perhaps Stack Overflow[8] being the most famous). So, another import factor is what we call the existing community support, that is, if useful information about how to implement a specific task, using the toolkits chosen, is easy to find on the Web.

3.5.2 NLTK

In order to test NLTK's performance on this task, we used an n-fold cross validation strategy, splitting each corpus into `n` similar size pieces, with one piece being used to test the classifier, and the remaining pieces to train it. This is done `n` times, and, in the end, an average of all the results is made, being this the final result of this evaluation strategy. This is done to reduce result variability, by using different training sets on each of the `n` times. A total of 72 tests were executed, 36 to each corpus, combining each of the word feature functions mentioned in Section 3.3, with the execution of the cross validation explained in this section, using 2 to 10 folds. In this section we will only present the results for cross validation with 10 folds. With a higher number of folds, the results obtained are more reliable, being this the reason why we will only consider 10 fold cross validation in this section. But, the results for 2 to 9 folds can be found in Appendix B.

Corpus	**Features**	**Accuracy**	**Precision**	**Recall**	**F-measure**
Movies	Baseline	0.6770	0.7469	0.6809	0.6531
Movies	Stopword removal	0.6500	0.7385	0.6494	0.6108
Movies	Most significant bigrams	0.7425	0.7514	0.7403	0.7357
Movies	Most significant bigrams of most used words	0.8315	0.8317	0.8313	0.8306
Twitter	Baseline	0.7477	0.7503	0.7471	0.7466
Twitter	Stopword removal	0.7397	0.7408	0.7395	0.7391
Twitter	Most significant bigrams	0.7528	0.7530	0.7529	0.7526
Twitter	Most significant bigrams of most used words	0.7646	0.7648	0.7649	0.7645

Table 3.1: Test results achieved with NLTK

Looking at the results in Table 3.1, many observations can be made regarding NLTK's performance across both corpora. Regarding the movie corpus, one aspect that stands out is that the classifier always achieved better precision than recall, although the difference decreases as more complex features are used. This means that among the reviews which the classifier failed to classify correctly (false positives and false negatives), there was a larger number of false negatives than false positives. Ultimately, it can be said that the classifier has a tendency to classify more reviews as negative, but the ones it classifies as positive are more likely to be in fact positive. Regarding stopword removal, for the movie corpus, we can see that there is a slight drop across all measures, meaning that stopwords, in this case, add information, unlike most cases in which they can be removed and the classifier's performance improves.

In relation to the baseline features, we can see a significant increase in accuracy and recall when we use the features that consider only the 200 most significant bigrams from each movie review. With these features, the classifier produces less false positives, which in turn means that it can classify more negative reviews correctly. Because of this, the classifier's accuracy also increased almost 10%. Finally, for the movie corpus, the best performances are registered with the most complex features, that include

[8] `https://stackoverflow.com/` (last visited on 02/05/2018)

only the bigrams with the best score among the 10000 most common words in the training set. As it can be seen above, for any test, none of the measures drops below 0.8, which is a major improvement over the other features used. Having in mind all these results for the movie corpus, it can be easily concluded that the best performance is achieved using the best bigrams among the most common words in the training set. The results are more consistent, and the improvement over the other feature functions is significant.

Moving on to the Twitter corpus, first analysis of the results shows that the baseline results are much more consistent than they were with the movie review corpus. Besides the smaller variation between the several tests executed, for each test, all measures have almost equal values. Adding to all this, it is still important to note that, with this corpus, the classifier's baseline performance increased considerably. Regarding the stopword removing features, we can see that the results are very similar to the ones achieved using the baseline word features, with a small drop across all measures. This leads to the conclusion that, much like what happened with the movie corpus, stopwords add useful information to the classifier, and their removal comes with a small performance penalty.

With the most significant bigrams from each tweet, the results are very similar to the ones obtained using other features, with no significant changes to any of the measures across all tests. This due to the tweets' small size. Because of it, unlike what happened with the movie review corpus, no words or bigrams are filtered out due to their low score. Finally, using the most complex features, we can see a slight increase in the classifier's performance, unlike what happened with the movie review corpus, where there was a considerable boost of performance. Despite this fact, these are the word features that achieve the best results with the Twitter corpus.

3.5.3 Bluemix

Implementation using the NLU service

Starting with the NLU service, since it is a pre-trained service, the evaluation methodology adopted for this case was simply to pass the whole corpora to the service, for it to classify all the reviews and tweets present in them (100% of the corpus for testing). No pre-processing was done to any review or tweet (such as stopword removal, for instance), due to the fact that users must only pass plain text to the service, without any kind of change to the original text.

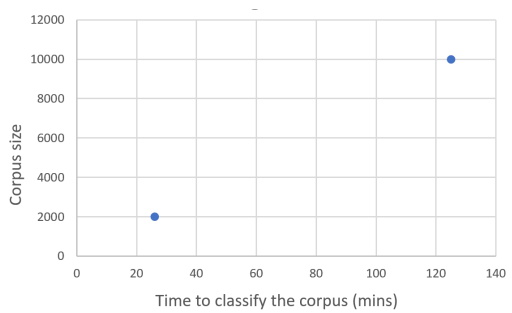


Figure 3.1: Relation between the corpus size and the time taken to classify it using the NLU service

As it was said before, both of Bluemix's services used to implement this task took a long time in classifying text, or training the classifier, in the case of the NLC service. Because of this, we show in Figure 3.1 the time that it took for the NLU service to classify both corpora. More specifically, for the movie review corpus, the service took 26 minutes to classify the 2000 reviews that make up the corpus,

whilst for the Twitter corpus, it took over 2 hours (125 minutes) to classify 10000 tweets. Not having in mind any additional test results, looking only at the time it takes to classify the corpora, we can see a clear **disadvantage** in relation to NLTK. With NLTK, each of the tests already presented in this section, took less than one minute to run, and each of them involved training and testing several times (the same number as the number of folds used for cross validation). Because of this very large gap in execution time, although it involves less code complexity, testing using the NLU service is much less practical than using NLTK, in which the user can obtain quick results.

Corpus	**Accuracy**	**Precision**	**Recall**	**F-measure**
Movie reviews	0.7885	0.9646	0.5990	0.7390
Twitter	0.5392	0.5369	0.5704	0.5531

Table 3.2: Test results using Bluemix's NLU service

Moving on to the service's actual performance in classifying the corpora, Table 3.2 shows the results obtained. Regarding the movie review corpus, we can see that in terms of accuracy, it performed better than NLTK's baseline and stopword removing features, and is similar to the accuracy obtained using the most significant bigrams features. In terms of precision, the NLU service achieved almost perfect results, meaning that it very rarely produces false positives (the reviews it identifies as positive are almost always in fact positive). But, with recall much lower at almost 0.6, it produces much more false negatives, meaning that almost 40% of the corpus's positive reviews are classified as negative.

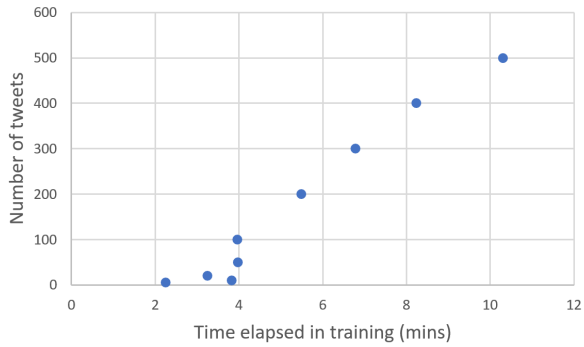
Whilst the results for the movie review corpus were more or less in pair with the results obtained using NLTK, for the Twitter corpus the results are significantly worse using the NLU service. With accuracy, precision and recall all lower than 0.6, the results are much worse than those achieved with the baseline word features in NLTK. Also, with the service needing 125 minutes to classify all the tweets, whereas in NLTK the classifier can be trained and then can classify test sets several times per minute, for the Twitter corpus NLTK is by far the better choice of toolkit to use.

Implementation using the NLC service

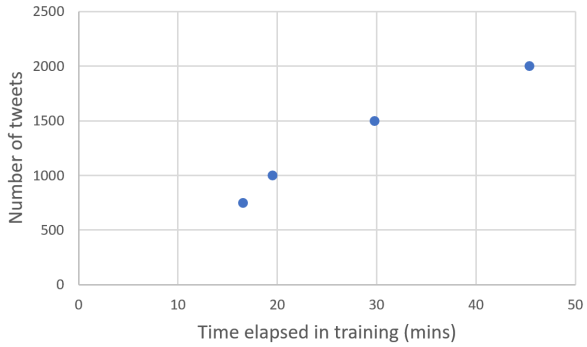
The other service that was used to implement this task was, as it was said before on this section, the **NLC** service. Unlike what happens with the NLU service, with this one the user must provide training data, so as to train the classifier. Because of this, and due to the large amount of time it takes to train, and then classify all the tweets (times in Figure 3.2), the testing methodology adopted for this service is somewhat different from the one adopted with NLTK. We chose to perform several tests, in each one choosing a predefined number of tweets from the corpus, at random, and using these random tweets to train the classifier. We chose to perform tests with 5, 10, 20, 50, 100, 200, 300, 400, 500, 750, 1000, 1500 and 2000 random tweets, used to train the classifier. We chose not to perform tests with more training tweets, due to the longer time they would take.

Looking at the times in Figure 3.2, we can see clearly a relation between the number of tweets used to train the classifier, and its training time. With the minimum training time being a little over 2 minutes, for only 5 samples (a very small number), it can be seen that the training time grows as the number of tweets grows, taking roughly 45 minutes to train when 2000 tweets were used, the maximum number used on testing. This is a recurring issue when using this service, as more people complain online about the time the service takes during training[9]. We can also see that the execution times are quite

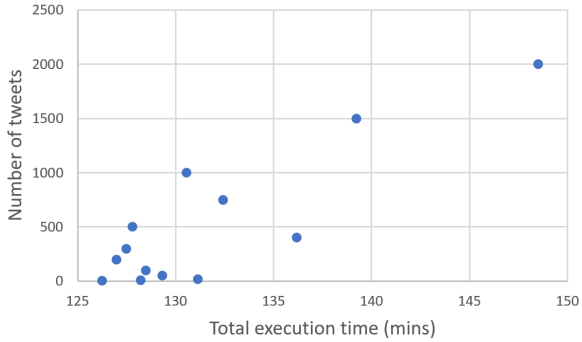
[9] `https://developer.ibm.com/answers/questions/253058/natural-language-classifier-training-time/`



(a) Number of tweets versus training time



(b) Number of tweets versus training time (cont.)



(c) Number of tweets versus total execution time

Figure 3.2: Number of tweets used for training and the consequences in terms of execution time

long, being similar to the ones obtained using the NLU service, ranging from a minimum of two hours and six minutes when 5 tweets were used for training, to a maximum of two hours and 28 minutes when 2000 tweets were used. Once again, looking only at the time it takes to train and test the service, NLTK has a clear advantage over the two Bluemix services used in this task.

Number of tweets used for training	**Accuracy**	**Precision**	**Recall**	**F-measure**
50 tweets	0.5628	0.5549	0.6387	0.5938
100 tweets	0.6142	0.5769	0.8533	0.6884
200 tweets	0.6756	0.6641	0.7078	0.6852
300 tweets	0.7112	0.6870	0.7811	0.7310
400 tweets	0.7192	0.7038	0.7639	0.7327
500 tweets	0.7195	0.7116	0.7437	0.7273
750 tweets	0.7371	0.7152	0.7936	0.7524
1000 tweets	0.7479	0.7597	0.7340	0.7467
1500 tweets	0.7302	0.7267	0.7443	0.7354
2000 tweets	0.7385	0.7719	0.6913	0.7294

Table 3.3: Test results using Bluemix's NLC service

Table 3.3 shows the results obtained using the NLC service. The tests with 5, 10 and 20 random tweets used for training were just to have an idea of what the classifier could achieve with little training data, which is why their results are not shown here. But, they can be consulted in Appendix B. Looking at the results, we can see that there is a general growth across all measures, as the number of tweets used for training increases, with the best results being achieved with 750 or more training tweets. These results are more or less similar to the ones achieved using the baseline word features in NLTK.

(last visited on 06/05/2018)

3.5.4 Conclusions

Having analyzed the results achieved using NLTK, the NLU and NLC services, there are several key aspects worth mentioning.

Execution time

Although this is not related to the actual performance of the classifiers, it is still an important factor in the decision of which tool is more adequate, since users want fast results, and not to wait hours to have some results. In this aspect, the clear advantage goes to **NLTK**. With it, users can train and test classifiers in a matter of seconds, whereas with Bluemix, this can take several hours, as we have shown in this section.

Performance

With all the test results presented in this section, we can say that better results are achieved using NLTK. Although it requires some more complexity in terms of the words features to be extracted from corpora, the results make up for this level of complexity. So, in terms of performance, NLTK exhibits better results than Bluemix.

Regarding the other factors mentioned in Section 3.5.1, which are less related to the performance and more to the development stage of the two systems, some conclusions can also be drawn.

Installation difficulty

As it was said previously, for this task the corpora used are both included in NLTK's pool of resources, meaning that, for the NLTK implementation, the only thing that was necessary to install was the toolkit itself and the resources that it includes. NLTK is very simple to install, with users being able to do it through the command line, or by downloading and executing the binary file found in the toolkit's website. As for the remaining resources, they can be installed resorting to an API call, or, once again, via the command line. Overall, it is a very fast and simple process.

Regarding the Bluemix implementation, it was necessary to install the Watson Developer Cloud API for Python, and Python's `sklearn` library, for testing purposes. Both of these libraries can be installed easily using the command line. As such, we can say the installation of all the toolkits and libraries for this task is quite simple and fast.

Documentation

On NLTK's website[10], developers can find a large amount of information regarding the API, and several use case examples, meaning there is vast documentation for anyone to consult. Since this an open-source toolkit, users can even see specific code snippets to understand what a certain function does, or how a class is defined and its attributes. But, such extensive documentation can become cumbersome for developers due to its sheer size, making it more difficult to find the necessary information quickly. In this case, the information available was able to, most times, steer us in the right direction when there were doubts, since it provided us with all we needed to know.

Regarding Bluemix, both the NLC and NLU services have extensive documentation, along with several code examples that illustrate how to make use of their API, and the usage restrictions that have to

[10] `https://www.nltk.org/` (last visited on 02/05/2018)

be respected in certain scenarios. In spite of its depth, some details are not discussed (some of the error messages the service outputs, for instance), making developers have to search somewhere else for solutions, or learn from their own experience (as it happened in our case). Having all of this in mind, we can say that NLTK's documentation, in this case, was more enlightening and provided more detail than the documentation of the NLC and NLU services.

Community support

In this aspect, there is a big gap between NLTK and Bluemix, regardless of the task being implemented. The former is a widely used open-source toolkit, very popular amongst researchers and students, whilst Bluemix is an online platform (non open-source) that requires its users to log in and, depending on what services they use, provide payment methods. Because of this, when searching online for sentiment analysis examples and doubts for NLTK, many results were returned, showing that it is indeed a very popular toolkit. On the other hand, for Bluemix, very few results were returned for this particular task, giving NLTK a clear advantage in this field.

Final remarks

Having all these factors in mind, we can say that **NLTK** is a better choice than Bluemix for the implementation of a sentiment analysis system. In all factors considered, NLTK exhibits better results than Bluemix, making it, in our opinion, the better choice of toolkit to implement the task of sentiment analysis.

3.6 Implementation using other toolkits

Although we did not implement this task (and the other two) with any other toolkit, in this section we will discuss what would be necessary to implement a sentiment analysis system using both Stanford CoreNLP and SpaCy. These were the two other toolkits that were the main candidates to use to implement the chosen tasks, other than NLTK. Because of this, part of this work should be dedicated to these toolkits. This will also be done in the end of Chapters 4 and 5 for Named Entity Recognition (NER) and the natural language interface for database, respectively.

3.6.1 Stanford CoreNLP

Much like what happened with NLTK, with Stanford CoreNLP, developers can perform sentiment analysis in two different ways:

- Using the `sentiment` annotator that is included with Stanford CoreNLP, as it was mentioned in Section 2.3.2;
- Using the **Stanford Classifier**, which allows users to train a classifier with their own training data, including data related to sentiment analysis.

If users do not have a training corpus, or do want the additional complexity of having to train a model, the first option is the most appropriate. Using the `sentiment` annotator, it is very simple to perform this task. The first step is to create a pipeline object (if developers are using Java), which is the basis of any task performed with Stanford CoreNLP. Next, users must choose which annotators they want to use, through a `Properties` object. For this particular task, as it was already said, the `sentiment` annotator is the only one necessary. After this is done, for each sentence or text that users want analyzed, they must create an `Annotation` object that contains the said chunk of text. Once these objects are

created, they must be passed to the pipeline, one at a time.

The results of the analysis made by the pipeline are stored in the `Annotation` object created earlier, and can be be accessed by developers easily through the API. One aspect worth mentioning is that, with this annotator, sentiment analysis is performed at sentence level. There is no overall sentiment score for when there are multiple sentences in the text passed to the pipeline. This can be an obstacle when users want the sentiment of a chunk text with several sentences, like what happened in the corpora we used for this task, especially in the movie review corpus. One solution to this obstacle is to make an average of the sentiment score of all the sentences in text, and make that the overall sentiment score. But this may not be very reliable in some cases, since the average may not translate the overall sentiment.

On the other hand, if users wish to perform sentiment analysis over a more specific domain, they can use their own data and train the **Stanford Classifier** with it. This classifier consists of a Java implementation of a maximum entropy classifier[11]. To train the classifier, and eventually test it, users must create a text file with all the training data. For this particular task, the file should have, for each line, the sentiment attributed to a text (positive, negative, or neutral) and the text associated to that sentiment, separated by a tab character. Once this is done, users must also create a properties file (a text file), which will include all the properties and options users want the classifier to consider. Additionally, in this file users must specify the paths to train and test files, in the case that testing is performed. Finally, users can train and test the classifier, using either the API or the command line, although it is simpler to do it via the terminal, where users only need to run one command. After running this command, the test results will be printed on the terminal, showing the usual metrics (accuracy, precision, recall and f-measure). More details about this process can be seen in Stanford CorenNLP's wiki[12].

3.6.2 SpaCy

Regarding the task of sentiment analysis on SpaCy, after some research, we came to the conclusion that it does not have any model or class dedicated to this task. To our knowledge, users must use other libraries besides SpaCy to be able to perform this task. In the toolkit's website, there is a link[13] to an example of sentiment analysis performed with the aid of Python's `Keras` library. This example is quite long and complex, but it shows users how it could be implemented using both SpaCy and `Keras`.

But, it is also worth mentioning that the only thing that SpaCy does in this example, is splitting the text into sentences. This is a very small part of this task, and it goes to show that SpaCy, regarding the task of sentiment analysis, has very few resources to help users implement this task.

[11] `https://nlp.stanford.edu/software/classifier.shtml` (last visited on 02/05/2018)
[12] `https://nlp.stanford.edu/wiki/Software/Classifier/Sentiment` (last visited on 02/05/2018)
[13] `https://github.com/explosion/spaCy/blob/master/examples/deep_learning_keras.py` (last visited on 02/05/2018)

4

Named Entity Recognition

4.1 Overview

As it is stated in the Glossary, NER consists in assigning pre-defined categories to words in a text. Depending on the software used to perform this assignment, and even the context of the text in question, there are many categories that can be attributed, some of the most common being: location, person, organization, time, quantity, or monetary value. Because of this, a particular algorithm can perform very well in a certain domains, but very poorly in others, depending on the datasets used to train the system. Normally, each dataset refers to a specific domain, such as news articles, web pages, or social media text.

In Bluemix[1], the **Natural Language Understanding** service is the one that offers this task. This service supports NER in a wide array of languages, such as English, Portuguese, French, German, Italian, Russian, Spanish, and Swedish[2]. It also recognizes a very long list of named entities[3], including the most common ones, mentioned before. The NLC service could also be used for this task, but in the end we chose to use the NLU service, due to the fact that it would just take too much time to process the corpus, as it will be explained later. In NLTK[4], there is a pre-trained model for the English language, which detects entities of three types: **person**, **organization**, and **GPE** (geo-political entity)[5]. But it is possible for users to train their systems with other datasets (which is what happened in our case), in English, or in other languages, which makes possible performing this task over a variety of languages.

To implement this task using NLTK, we used the provided **CRF Tagger**[6] to choose the most likely tag to be assigned to each token in a sentence. As what happened with the previous task, multiple feature

[1] `https://www.ibm.com/cloud-computing/bluemix/pt` (last visited on 02/05/2018)
[2] `https://console.bluemix.net/docs/services/natural-language-understanding/language-support.html#language-support` (last visited on 02/05/2018)
[3] `https://console.bluemix.net/docs/services/natural-language-understanding/entity-types.html#entity-types-and-subtypes` (last visited on 02/05/2018)
[4] `http://www.nltk.org/` (last visited on 02/05/2018)
[5] `http://www.nltk.org/book/ch07.html` (last visited on 02/05/2018)
[6] `https://www.nltk.org/_modules/nltk/tag/crf.html` (last visited on 03/05/2018)

functions were implemented, each generating features for the words it considers. These features will be presented in more detail in Section 4.3. The main difference across these functions is the number of words they consider at each time:

- Features that consider only the current word being analyzed (baseline);
- Features that consider, in addition to the current word, the previous and next word;
- Features that consider, in addition to the current word, the two previous and two next words;

4.2 Corpus selection

Unlike what was mentioned in Section 3.2, in which we used resources already included in NLTK, in this case we chose not to do so, due to the lack of a tagged English corpus for NER. Because of this, we had to search other resources available publicly, eventually choosing to use the **Groningen Meaning Bank (GMB) corpus**[7]. Developed by the University of Groningen, this corpus is composed of over 60000 tagged sentences, in which the majority of them are annotated by humans. Each of the words that compose the corpus has a NER tag associated to it, that can belong to one of the following categories:

- **art** - artifact. Examples: Good Morning America, Association of Tennis Professionals, Chrysler, Dodge, Twitter, Fox News;
- **eve** - event. Examples: Australian Open, World War II, Olympic Games, New Year, Indian Republic Day, World Cup;
- **geo** - geographical entity. Examples: Lion, United States, South Korea, Jerusalem, London, China, Europe;
- **gpe** - geopolitical entity. Examples: Ukrainian, Pakistani, American, Cuban, German, Mexican, French, Iraqi, Danish;
- **nat** - natural event. Examples: Hurricane Katrina, HIV, AIDS, Tropical Storm Rita, H5N1;
- **org** - organization. Examples: NATO, United Nations, Central Election Commission, FIFA, Central Bank of West African States, International Monetary Fund;
- **per** - person. Examples: Ron Paul, President Bush, Marc Anthony, Mr. Siniora, Kofi Annan;
- **tim** - time. Examples: Sunday, 2001, Decemeber 26, later, Monday evening, 2006-2008;
- **O** - none of the above (the word belongs to none of the categories mentioned).

Besides these main categories, each word is tagged with a subcategory, depending on its main category. But, we chose not to include these subcategories in our study, due the additional degree of complexity it involved, and also because some of them were somewhat confusing and unnecessary.

Figure 4.1 shows part of one the many data files that comes with this corpus. As it can be seen from the figure, each line is dedicated to one word only, and each word has several annotations associated, which in term are separated by tab characters. In addition to all this, the end of a sentence is marked by two new line characters, unlike what happens for the words, which are separated by one new line character. Each of the lines from the figure can seem confusing, but actually only a few annotations matter,

[7] `http://gmb.let.rug.nl/` (last visited on 02/05/2018)

in our case. The first element of each line is the word in question. The third element is the resulting of stemming the word, and finally, the fourth element is the word's category and subcategory.

```
Egyptian    JJ  egypt   gpe-nam 0   []  of  Organization    N/N lam(v1,lam(v2,merge(b1:drs([b1:[]:x1]
police  NNS police  O   1   []  O   Organization    N   lam(v1,b1:drs([],[b1:[1002]:pred(v1,police,n,
have    VBP have    O   0   []  O   O   (S[dcl]\NP)/(S[pt]\NP)  lam(v1,lam(v2,lam(v3,app(app(v1,v2),l
arrested    VBN arrest  O   1   [Theme,Agent]   O   O   (S[pt]\NP)/NP   lam(v1,lam(v2,lam(v3,app(v2,l
at  IN  at  O   0   []  O   O   (NP/NP)/(S[asup]\NP)    lam(v1,lam(v2,lam(v3,app(app(v1,v2),lam(v4,me
least   JJS least   O   1   [Patient]   O   O   S[asup]\NP  lam(v1,lam(v2,app(v1,lam(v3,merge(b1:drs(
16  CD  16  O   0   []  O   O   N/N lam(v1,lam(v2,merge(b1:drs([],[b1:[1007]:card(v2,16,eq)]),app(v1,
members NNS member  O   1   []  O   Human   N   lam(v1,b1:drs([],[b1:[1008]:pred(v1,member,n,'1')]))
```

Figure 4.1: Part of a data file from the GMB corpus

As what happened with the implementation of sentiment analysis both on NLTK and Bluemix, this corpus was used in both toolkits. In the two following subsections, we will describe in greater detail the implementation of this task in these two toolkits.

4.3 NLTK implementation

It was not as easy to find examples of NER performing programs online, as it was with sentiment analysis, and because of this, deeper research and experimentation was required to implement it. Also, because the corpus used was not included in NLTK, additional work had to be done to extract the information that mattered from the corpus (the three annotations mentioned in Section 4.2).

To implement this task we used NLTK's **CRF Tagger**, which is based on the implementation of a Conditional Random Field model in Python's `pycrfsuite` library, to choose the most probable tag for each token in a sentence. This tagger takes as input features generated by a feature generation function, which will be used to choose the most likely tag for each word. The feature generator must be implemented by the user, and it affects the tagger's performance (richer features will likely increase performance). To have a clearer idea about this difference in performance, we chose to implement three different feature generation functions, with different degrees of complexity:

- **Current word (baseline)** - this is the simplest of the three functions. It only considers only the current word being passed to the function, without any information about any of the previous words that have been passed, or upcoming words. Regarding the features being extracted from each word, this function returns the word being analyzed, its stemmed form and Part of Speech (PoS) tag, along with a series of truth values that indicate:
  - If the word contains a dash;
  - If the word contains a dot;
  - If the word contains only capital letters;
  - If the word is capitalized (only the first letter is a capital letter);
  - If the word only contains letters.
- **Previous and next words** - this function generates the same features as the previous one, except it takes into account the previous and the next word, besides the current word. As so, it generates the same features as the previous function, but for these three words;
- **Two previous and next words** - As the name suggests, it does the same as the two previous functions, with the addition it has access to a greater history (two previous words instead of one) and also can access the two next words (instead of just one).

Some research was necessary, to have a clearer idea of what features should be extracted from the corpus. Because of this, the implementation of these feature functions was not straightforward, but from the moment we knew which features were the most appropriate, and how to pass them to the classifier, they were quickly implemented. Some additional manipulation of the corpus was necessary, not only to extract the necessary annotations, as it was said before, but also to generate the actual named entities in it, which only contained tagged words, not chunks.

4.4 Bluemix implementation

Looking once again at the list of services in Bluemix's "Watson" section and after analyzing what each of them offers, we are only left with one viable option to implement this task using Bluemix: the **NLU service**. With the NLC service, we could train a classifier to recognize the category of some named entities, but, during the classification stage, we would have to pass each chunk individually to the classifier. If a sentence was passed to the classifier, it would output only one category for the sentence as a whole, which is not what is desired for this task. This would not be what we want for this task, and because of this, the NLC service was not used in this task.

The NLU service was already used to perform sentiment analysis, but since it can extract several types of meta-data from plain text, including named entities, it fits exactly what is required for this task. Also, since users can pass as input large chunks of text in each request, with almost no penalty regarding the time the service takes to return a response, the execution time will be much smaller than what it would be if the NLC service was used. Since the NLU service was already used to implement sentiment analysis, and it does not require any training, the execution flow is similar to what was done in that task. The first step, as it was with sentiment analysis, is to retrieve the raw text from the corpus, which is what is passed as argument to the NLU service. All that had to be done was to retrieve the first annotation (the original word) from each line of the GMB's data files, and generate the sentences as the files were read.

Having the text ready, the next step was to divide the raw text into chunks of 5 sentences each (this was the chosen size because this service only recognizes a maximum of 250 entities per request[8]) and pass each chunk to the service, for it to recognize the named entities in each chunk. As the results were returned by the service, the categories identified had to be "translated" to the categories present in the corpus. This was the most difficult part to implement, which required a significant amount of time analyzing what categories were returned by Bluemix, and which ones could be "renamed" to match the categories in the GMB corpus. For it to be possible to evaluate and compare the results achieved with NLTK, this step had to be done, and it will be described in greater detail in Section 4.5.1. Finally, having recorded all the entities recognized and their respective categories, accuracy, precision and recall were calculated, the same way they were calculated for NLTK.

4.5 Evaluation

4.5.1 Methodology

To evaluate NLTK's performance in this task, we will use the same evaluation strategy as the one that was used for sentiment analysis. An n-fold cross validation strategy was used, splitting the corpus into `n` similar size chunks, using one chunk to test the classifier, and the remaining chunks to train it. As

[8] `https://www.ibm.com/watson/developercloud/natural-language-understanding/api/v1/#entities` (last visited on 06/05/2018)

it was said in Section 3.5.2, this is done n times, and after this is done, an average of all the results is made, being this the final test result. For this particular task, a total of 27 tests were performed, 9 for each feature generation function, varying the number of folds used for the cross validation strategy, from 2 to 10 folds.

Although the cross validation strategy adopted for this is similar to what was used before, the actual formulas to calculate precision, accuracy and recall, are different from the "regular" formulas. Since this is not a binary classification task, there is no standard way to calculate these metrics. After some online research, we chose to follow one methodology which takes into account, not only the boundaries of the named entities recognized (if the text is the same as the original named entity, then the boundaries are correct), but also if the categories assigned to the named entities are the correct ones. This is the scoring system used in Message Understanding Conference (MUC) events [26]. Using this scoring system, NER systems are evaluated on two axes:

- **Correct type** - a correct type is recorded when the system assigns the correct type (category) to a named entity, as long as there is an overlap between the named entity recognized by the classifier, and the original named entity;
- **Correct textual boundary** - a correct boundary is recorded when the boundaries of the named entity recognized by the system and the original named entity are the same, regardless of their types.

For both these axes, three measures must be kept:

- **COR** - the number of correct answers returned by the system. This number increases independently for each axis when a correct type, or a correct textual boundary is recorded;
- **ACT** - the number of answers returned by the system. This consists in the number of textual boundaries plus the number of types that were returned by the system (one type and one boundary per each named entity recognized);
- **POS** - the number of textual boundaries plus the number of types in the training corpus.

Having all these measures in mind, **precision** can be defined as:

$$Precision = \frac{COR}{ACT}$$

That is, the total number of correct types and textual boundaries recorded, divided by the number of answers returned by the classifier. Moving on to **recall**, this measure can be defined as:

$$Recall = \frac{COR}{POS}$$

Which is the total number of correct types and textual boundaries recorded, divided by the number of named entities present in the corpus. The f-measure is calculated the same way it was defined in Section 3.5.1. The formula adopted for accuracy had in mind the definition of accuracy in the binary classification scenario. In that case, reminding the definition made in Section 3.5.1, accuracy can simply be expressed as the number of true positives plus true negatives (the number of data samples which were given the correct category), divided by the number of data samples. Adapting this concept for this particular task, **accuracy** can be defined as:

$$Accuracy = \frac{COR(TEXT) \quad and \quad COR(TYPE)}{ACT}$$

The nominator can be seen as the number of named entities guessed by the system that have recorded both correct texts and types (the textual boundaries and category are both correct). This is, effectively, the number of fully correct entities, both on the actual text boundaries, or its type. Finally, the denominator is the number of named entities recognized by the system. This seems an appropriate approximation to the definition used for binary classification, and it will be definition used for this particular task. Another important aspect to mention is that for the NLU service, the categories, or types of named entities that this service recognizes are different from the ones present in the GMB corpus. Because of this, a "translation" had to be made, transforming the categories recognized by the service, to the ones present in the corpus. This was a very time consuming task, which involved a detailed analysis of what named entities were identified by the NLU service, and comparing them to the original ones, present in the corpus. Some categories allowed for a direct translation, since only the category's name changed (the named entities recognized were, theoretically, the same). The full category list, along with their GMB equivalents, can be found in Appendix C.

For some of the categories on that list, it was very clear that would not have any possible translation. This is the case for JobTitle, Crime, Drug, MusicGroup, Sport, HashTag, TwitterHandle, Vehicle and TelevisionShow. On the other hand, for some other categories, a more in depth analysis had to be made, because at first it was not very clear if they could be "translated" to one of GMB's categories. This happened for the **Quantity** category (returned by the NLU service) and the **gpe** and **tim** categories present in the corpus. The Quantity category encapsulates all types of quantities, such as numbers, time, sums of money, and thus it could not translated to any category present in the corpus. The most similar one is the tim category, but this only contains time related entities, so it also had to be put aside. As for the gpe category, it features only words that refer to a person's nationality, such as "portuguese", "french", or "english", and there is no category for this kind of word in the NLU service, so this category could not be used. Because of all these incompatibilities, these categories had to be put aside, and so, the remaining categories were used to calculate the evaluation results.

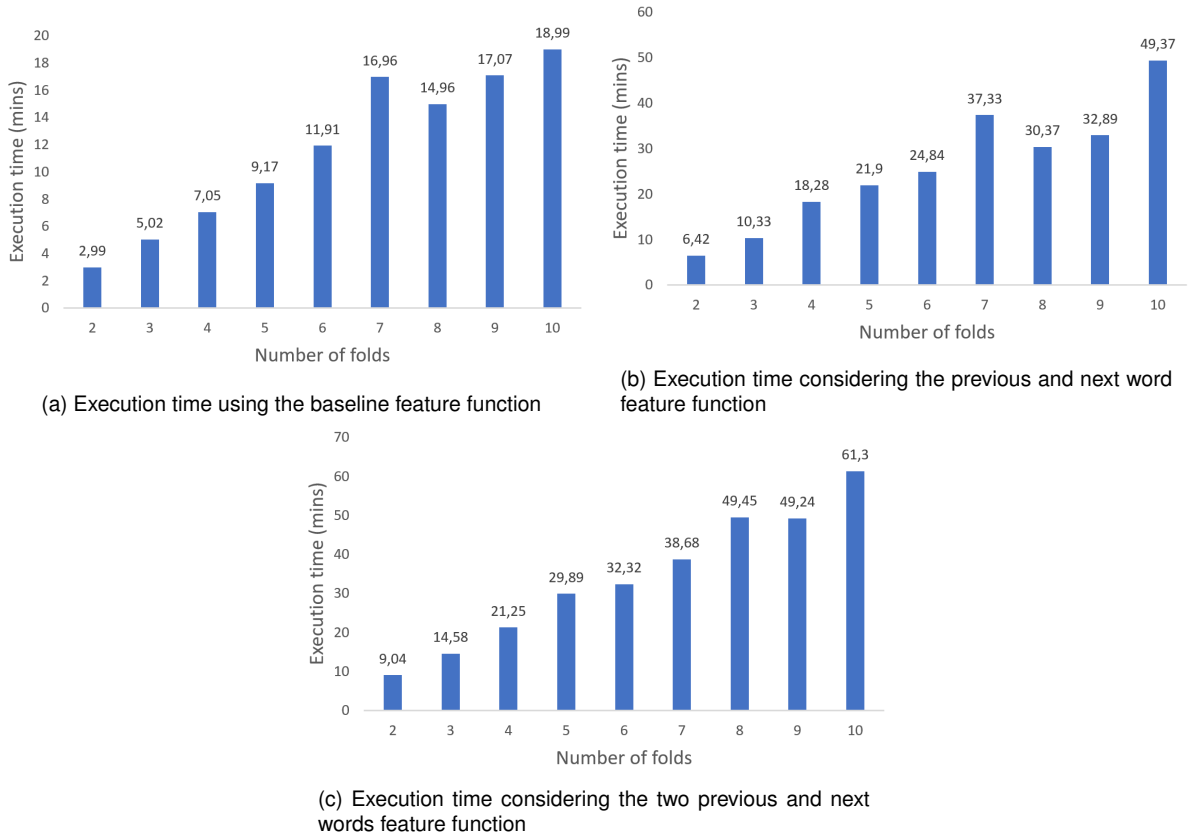
4.5.2 NLTK

Unlike what happened with sentiment analysis using NLTK, in which the execution time for each test were always under a minute, for this task, the execution times were much larger. Because of this, it makes sense to show here the execution times obtained for all tests, to later compare with the results obtained using Bluemix.

As it can be seen from Figure 4.2, the execution time increases as the number of folds increases, regardless of the feature generation function used. This is due to the additional training that has to be performed for higher numbers of folds. Another aspect to note is that the execution times increase as more features are generated (considering more words at a time generates a greater number of features), ranging from almost 3 minutes using the baseline function with 2 fold cross validation, to approximately 61 minutes using the feature function that generates the most features, which is a very significant difference.

Features	**Accuracy**	**Precision**	**Recall**	**F-measure**
Baseline	0.7985	0.8709	0.8682	0.8695
Previous and next word	0.8206	0.8859	0.8788	0.8824
Two previous and next words	0.8237	0.8880	0.8806	0.8843

Table 4.1: Test results achieved with the GMB corpus and NLTK



(a) Execution time using the baseline feature function

(b) Execution time considering the previous and next word feature function

(c) Execution time considering the two previous and next words feature function

Figure 4.2: Fold number versus execution time in NLTK

Moving on the test results regarding performance, Table 4.1 shows the results of the tests executed using the three feature functions presented earlier, with 10 fold cross validation. For 2 to 9 folds, the results can be consulted in Appendix B. Using the baseline features, in average, almost 80% of the named entities recognized by the classifier have the correct type and text boundaries, a very positive result. Another aspect to note is that the values for precision and recall are very similar, with 87,09% for precision, and 86,82% for recall, both very high values. This indicates that the number of correctly identified texts and types is very similar to the number of texts and types present in the corpus. Also, the marginally higher precision value indicates that the system makes a slightly smaller number of guesses than the number of texts and types in the corpus.

As for the function that considers the features regarding not only the current word, but also the previous and next word, the results achieved with this function are very similar to the ones with the baseline features. The main differences are on accuracy, which increased approximately almost 2,5%, and on precision, which registered an increase of 1,5%. The smallest difference is in recall, which showed an increase of 1%. These small differences indicate that generating features for the previous and next word does not provide much useful information for the classifier. Finally, for the last feature function, the results are not very different to the ones achieved with the previous two. It can be seen that there is a very small increase in accuracy (0,3%) in relation to the results obtained with the previous and next word features. Even smaller increases are registered for precision in recall, meaning that the additional history and next word provide almost no additional information to the classifier.

Comparing the results obtained using the three feature generation functions, it can be seen that the best results were achieved when considering the current word, along the two previous and two next

words to that one. It registered the best values for all measures except precision, although the difference to the function that generates features for the previous, current, and next word, is very small.

4.5.3 Bluemix

Before moving on to the test results, it is important to mention that, feeding the service chunks of 5 sentences in each request (due to the fact that the NLU service only recognizes 250 named entities per request), the total **execution time** was, approximately, 2 hours and 38 minutes. Much like what happened with sentiment analysis using Bluemix, this is a very long time, and, even not having in mind the performance obtained, this is always a downside, having to wait such a long period of time for the results.

Table 4.2 shows the results obtained using Bluemix's NLU service. Comparing these results with the best ones obtained in NLTK, it can be seen that there is a significant decrease in accuracy (of almost 20%), meaning that NLTK manages to correctly identify both the text and type much more consistently.

Accuracy	**Precision**	**Recall**	**F-measure**
0.6397	0.7294	0.7999	0.7630

Table 4.2: Test results using Bluemix's NLU service

Regarding precision, there was also a decrease of approximately 16%, in average, to the results achieved in NLTK. This means that the ratio of texts and types identified correctly by NLTK service is much higher than what was achieved with NLU. Recall registered a less significant drop, of roughly 9%, in relation to the best results achieved with NLTK.

4.5.4 Conclusions

Having shown the results achieved both for NLTK and Bluemix, there are, once again, some important aspects to mention.

Execution time

Having presented execution times for both NLTK and the NLU service, a very clear difference can be seen, with much better execution times being achieved using NLTK. The best results obtained using this toolkit, were with the feature function which considered, not only the current word, but also the two previous and two next words. Looking at the execution times obtained with that function, it can be seen that they range between approximately six minutes and thirty seconds, for 2 fold cross validation, and 61 minutes, for 10 fold cross validation. On the other hand, as it was said earlier in this section, the execution time obtained using Bluemix's service was 2 hours and 38 minutes. Considering that the NLU service is already trained, and that NLTK's classifier is trained several times, depending on the number of folds, there is a clear advantage for **NLTK**, in terms of execution time.

Performance

Comparing the results obtained using both toolkits, it can be seen that there is a clear advantage for NLTK, across all measures. But, As it was said earlier, despite having these precise values for each measure, since some categories could not be used in this study, due to incompatibilities in "translation", these results may not be most reliable to measure the difference in performance between the two systems. But, considering the categories that were not excluded, NLTK shows an advantage over Bluemix in performance.

Documentation

In this case, what was said in Section 3.5.4 stills holds for this task, since the NLU service was used to implement NER with Bluemix. Both NLTK and all of Bluemix's services (not only the NLU service), have detailed and extensive documentation to help developers use the APIs. But, as it was mentioned earlier, sometimes the documentation provided by IBM just does not cover all the details, giving NLTK a slight advantage in this field.

Community support

As what happens with documentation, with community support, the scenario is the same as it was in Section 3.5.4. Due to NLTK's high popularity among students and researchers in the field of NLP, the amount of information that can be found online by developers is much greater for NLTK than it is for any of Bluemix's services.

Final remarks

Having in mind these key points, it can be said that both toolkits have their advantages and drawbacks, with the best choice falling upon NLTK, to implement a NER system. The NLU service can identify a big amount of different categories, and does not require any training, but with its limit of 250 named entities per request, users cannot pass big chunks of text in each request, running the risk of the service's response not identifying all the named entities present in the text. So, as the users must pass relatively small chunks of text each time, this leads to larger execution times, which can be a very important factor if the user is dealing with a very large text. Also, due to the vast array of entities that the NLU service recognizes, depending on the domain of the text being analyzed, some categories may deem useless for the task in question, and this involves additional work to filter out these unwanted categories. On the other hand, with NLTK, users can train classifier to recognize the categories they specifically want to, provided they have a tagged NER corpus with these categories, which can be difficult to come by. In addition to this, the users must train the classifier, which is clearly not as easy as using the NLU service, in which the user must only input the plain text it wants analyzed.

So, if users want quick and very positive results, are comfortable with training a classifier, and have the necessary tagged corpus, using NLTK is the most viable option. But, if the user does not possess a tagged corpus for training, and is not sure as to which categories of entities he wants the system to recognize, or if he simply wants a tool which does not involve a lot of code complexity, using Bluemix seems a preferable option, despite the rather considerable difference in performance.

4.6 Implementation using other toolkits

As it was done in Section 3.6, in this section we will discuss how this task could be implemented using both Stanford CoreNLP and Spacy.

4.6.1 Stanford CoreNLP

To implement a NER system in Stanford CoreNLP, developers can use the existing `ner` annotator, much like the `sentiment` annotator, used to perform sentiment analysis. But, this annotator only recognizes a reduced number of entity types, such as person names, organizations, locations, and numerical entities. Because of this, and due to the fact that for this task, we used the GMB corpus to train and test our system, this annotator could not be used in our specific case. Instead, we need to be able to train

the classifier to recognize the categories present in the corpus.

To do this, Stanford CoreNLP provides its users with the `CRFClassifier` class (for Java). With this class, users can train a classifier to recognize named entities of any particular domain, provided they have the necessary training corpus. The training data should be in a text file, in tab separated columns, with words in one column and its respective label on the other column. In addition to the training file, much like what happened with sentiment analysis, users must provide a properties file. This text file must contain the path to the training file, the indication to which column refers to the named entity category, and which refers to the actual word, and as many features[9] as the users find relevant for the classifier to consider.

In our case, for the GMB corpus, the main difficulty would reside in transferring, or transforming the corpus into a text file with properties that were mentioned earlier. Once the training file is complete, along with the properties file, users can train the classifier. As what happened with sentiment analysis, users can train their classifiers through Stanford CoreNLP's API, or via the command line. Using the command line is much simpler, because it only requires one command to train and serialize the classifier. The only input needed is the properties file, which specifies all the information the classifier needs to train itself. After this is done, the classifier can be loaded at any time, and it will be ready to accept new sentences or texts to perform NER. In this case, testing cannot be done through the command line, but it would not be very different from what we have done for NLTK. Users would have to pass the test corpus to the classifier, and calculate the metrics manually, much like what happened in our implementation of NER systems.

4.6.2 SpaCy

Similarly to Stanford's toolkit, to implement NER with SpaCy, developers can use the provided model that is already trained. This model manages to identify a wide array of named entities, such as person names, companies, organizations, dates, percentages, among many other (the full list available on SpaCy's website[10]). But, since the GMB corpus was used, and the categories recognized by SpaCy are not completely identical to the ones present in the corpus, SpaCy's default model could not be used in our case. Once again, we would have to train a classifier or model to recognize the categories present in the corpus.

In SpaCy, this can be done by updating the existing NER model to recognize new categories, or edit already existing categories. To do this, users must first load which model they want to edit. They can edit the already existing or model, or create a new, blank model, which would be the most suitable for our case, since there would not be any category overlaps between the ones in the corpus and the ones already present in SpaCy. New categories can be added with the `nlp.add_label` function, passing the name of the new category. Next, using the `nlp.update` function, users can update the model loaded earlier by providing this function with text and its respective annotations. Finally, once all new categories and examples are provided, users must save their model using the `nlp.to_disk` function. After this, the model can be loaded at any time and it will be ready to identify named entities in text. Regarding testing, it would have to be done similarly to what was done with NLTK. Users would have to pass the sentences in the test corpus to the model and calculate all metrics afterwards.

[9] `https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/ie/NERFeatureFactory.html` (last visited on 02/05/2018)

[10] `https://spacy.io/usage/linguistic-features#entity-types` (last visited on 02/05/2018)

5

Natural Language Interface for Database

5.1 Overview

The last task, and the most difficult one, is the implementation of a natural language interface for database. With this kind of system, future users can pose questions made in natural language, and the system will query its underlying database for the answer [14]. More concretely, in our study, the questions will be made through written text. These systems can be seen as a variation of a Question Answering (QA) system. The main difference resides in the fact that a QA system will use unstructured information sources to provide an answer (plain text, or websites), whilst a natural language interface for database only uses structured information sources (databases, as the name suggests).

Normally, a system with this kind of functionality only answers questions regarding a specific domain, with all the knowledge being stored in a database. The usual architecture for this type of interface is composed of three main components [14, 31]:

- **Intent and Entity extraction** - In this stage, the question submitted by the user is analyzed, to extract the most important features: the intents expressed by the user, and the entities referred;
- **Disambiguation** (optional) - If there are multiple entities with the same name as the ones referred in the question, this step can be performed, for the user to specify which entity is the right one;
- **Mapping to query language** - In this step, several changes are made to the original question, transforming it first to a pseudo query language, which then can be easily translated into other existing query languages. The final query is passed on to the database afterwards.

Figure 5.1 shows the high level architecture we have adopted in the natural language interfaces that were implemented. As it was said before, the first step is to detect what is the user's intent, and the named entities he mentions in the question. Next, a logical form is created, which encapsulates the semantics of the user's question, and can be easily mapped to a query. Finally, after the query is generated, it is executed by database and the results returned are showed to the user.

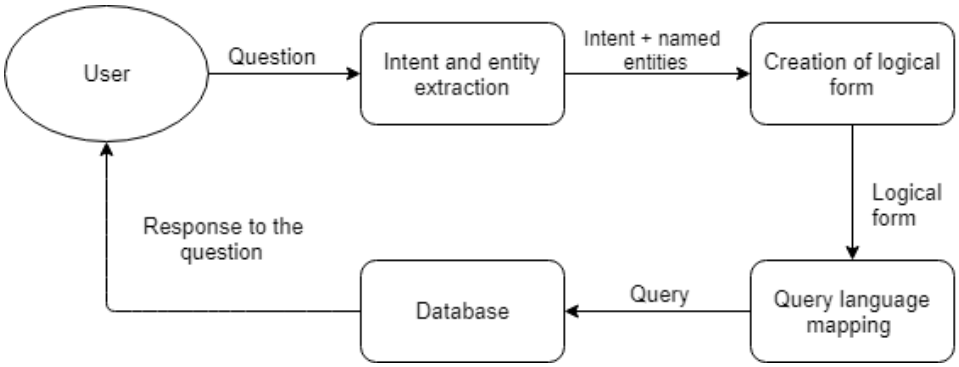


Figure 5.1: Adopted architecture for the systems developed

Before starting any kind of code implementation, the first step taken was to decide which domain to use. After some online research, we came to the conclusion that the **cinema** domain is one of the most common domains used, with a wide array of datasets available publicly online. Because of this, we chose to use the TMDB 5000 Movie Dataset[1], available on Kaggle[2]. As the name suggests, this dataset comprises information about approximately 5000 movies. This information was contained in two Comma-Separated Values (CSV) files. From these two files, a database was created, to be later used by the natural language interface to provide answers to the users' queries. Once the database was created, the information contained in the CSV files was retrieved and used to populate the database, leaving it ready to be used.

To implement this task on Bluemix, and on NLTK, the existing functionality already offered must be extended. Bluemix provides the **Watson Assistant** service, which can be used to build applications future users can interact with, using natural language. This service focuses more on defining and detecting intents and entities in questions, and designing the conversation flow. All that is related to transforming the question into a usable query, and perform the actual query, must be done outside of Bluemix. This also happens with NLTK, which possesses a chat module, as it was said previously, but it is very rudimentary. It cannot be used to implement a natural language interface for database, meaning that the system must be done from scratch.

5.2 Dataset

As it was said in the previous subsection, the chosen dataset contains information about cinema, such as information about movies, their cast, the people behind the production of each movie, among other facts. Seeing that it contains information regarding different fields, the dataset is divided into two CSV files:

- **Movies file** - this file contains information about the movies themselves (it does not include information about the people that participated on the movie). Each line contains data from one movie, with a total of 20 fields:
  - Budget (in US dollars);
  - Genres (Action, Adventure, for example). Since each movie can have multiple genres, this field is in JavaScript Object Notation (JSON)[3] format, for it to be possible to have several genres on a single field;

[1] `https://www.kaggle.com/tmdb/tmdb-movie-metadata/` (last visited on 02/05/2018)
[2] `https://www.kaggle.com/` (last visited on 02/05/2018)
[3] `https://www.json.org/` (last visited on 02/05/2018)

- Homepage: the movie's website;
- Id: unique identifier for each movie (not very relevant in our case, since these identifiers will be different from the ones used on the final database);
- Original language: the code of the most used language in the movie (e.g., en, pt, fr);
- Original title: the title of the movie;
- Overview: a short text that briefly describes the movie's plot;
- Popularity: a positive real number that expresses how popular a movie is. Since it is not clear how this value is calculated or obtained, it will not be used on our database;
- Production companies. Since there can be many companies responsible for a single movie, like what happens with the Genres field, this field is in JSON format;
- Production countries: the countries in which the movie was shot and/or produced. As the previous field, it is also in JSON format;
- Release date;
- Revenue (in US dollars);
- Runtime (in minutes);
- Spoken languages: all the languages that can be heard in the movie (also in JSON);
- Status: indicates whether the movie is released, if is being shot, or it is only rumoured (this field this not seem very relevant, and so it will not be a part of the final database);
- Tagline: short sentence, similar to a slogan;
- Title: equal to the original title;
- Vote average: a real number ranging from 0 to 10 indicating the average of the votes given to the movie;
- Vote count: the number of votes used to calculate the average.

• **Credits file** - this file contains information only about the people that take part in the production of each movie. Much like the previous CSV file, each line contains data for one movie, but this time, each line only has four fields:

  - Id: same as in the movies files (will also not be used in the database);
  - Title: the movie's title;
  - Cast: contains information about the actors cast for the movie, and what character they played (in JSON, due to the large number of roles in each movie);
  - Crew: contains information about all the other people that did not play a part in the movie, but had another role in its making (editing, sound, writing, among others). This field is also in JSON.

As it can be seen from the variety of fields present in both files, a natural language interface with this data as its knowledge source is able to answer a variety of different questions, proving itself to be a fitting dataset for the cinema domain. But, despite its quality, the data still has to be migrated to a database, for the interface to be able to answer the users' questions. This was the next step taken.

5.3 Database creation

Having chosen the dataset to be used, the next step was to put all the data in a database. Without this database, the natural language interface will not be able to answer any questions, since CSV files are not appropriate to perform queries efficiently. As such, the first step was to make a high level design of how the data should be organized across all tables.

5.3.1 Conceptual schema

There was already previous work done by a former student of IST, Ana Guimarães, regarding a natural language interface for database for the cinema domain. Although the database used in Ana's work [14] contained different information than our dataset, we could see that there were some overlaps in some cases, and so we decided to use some ideas of her design in ours, just to facilitate this step, not having in mind all the factors that affect query performance, which is not the main goal of this task (although query efficiency must be critical for bigger knowledge bases, but this was not the case). After analyzing all the data available in our dataset, and using some of the ideas present on Ana's thesis, a final schema was reached.

As it can be seen from Figure 5.2, the final database is composed of 12 tables, with the `movies` table being the one that contains most information on the movies CSV file. Since some of the information on this file is in JSON, some additional tables were needed for these fields. This is the case on the `spoken_languages`, `keywords`, `genres`, `production_countries` and `production_companies` tables. All these tables have foreign keys to the `movies` table. This is what makes joins between these tables possible. Besides these tables, both the `acting` and `activity` tables have foreign keys to the `movies` table. These two tables also serve as connection between the `persons`, `characters` and `activity_codes` tables. In addition to all these tables, there is still the `jobs` table, which contains all the different jobs that can be found on the original dataset. The `activity_codes` organizes jobs by department, and, besides being the link between many tables, the `acting` table contains one additional field, which is the cast order associated to a particular actor/character pair. The names of the fields in all tables are quite self explanatory.

With this structure, the database can be queried to answer a wide range of questions, not only about a particular movie, but also about any person involved in the making of any movie in the database. For instance, using the tables `persons`, `jobs`, `activity_codes`, `acting` and `movies`, the system can answer the question "Who acted with Julia Roberts in Pretty Woman?" with the following query:

```
SELECT person_name FROM persons, jobs, activity_codes, acting, movies WHERE
jobs.job_name = "Actor" AND persons.person_name NOT IN ("Julia Roberts") AND
jobs.job_id = activity_codes.job_id AND activity_codes.act_id = acting.activity_id
AND persons.person_id = acting.person_id AND acting.movie_id = movies.movie_id
AND movies.title = "Pretty Woman"
```

The system will also be able to answer less complex questions, which leads to simpler and shorter queries. To answer the question "What is the runtime of Titanic?", only one table is needed (the `movies` table), which generates a simple query:

```
SELECT runtime FROM movies WHERE movies.original_title = "Titanic"
```

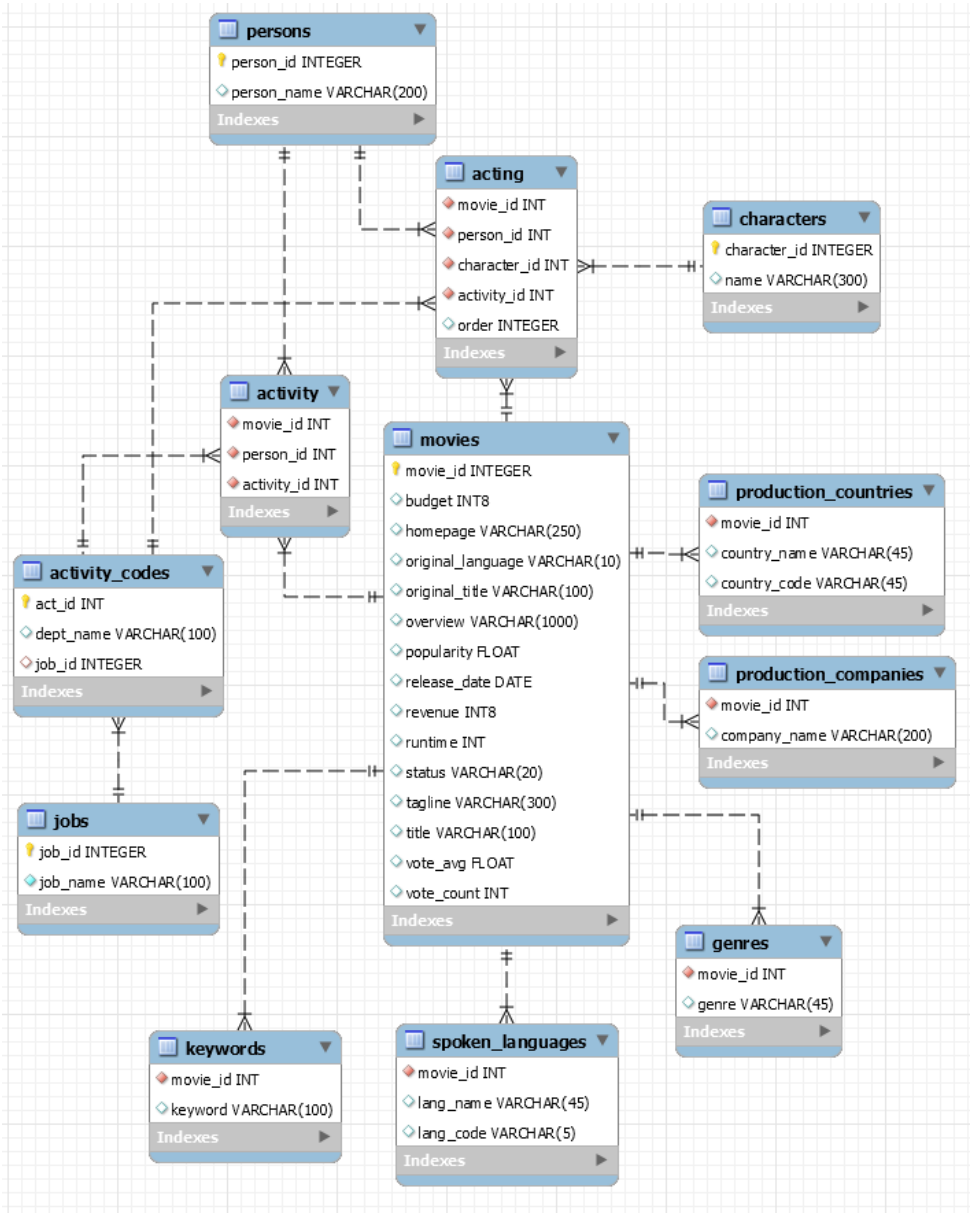


Figure 5.2: EER diagram of the cinema database

These are only some examples of the questions that users will be able to ask our natural language interface. As it will be seen in further detail on upcoming sections, with this data, a wide range of questions can be asked, so much so that our interface does not cover all types of questions, since that would be very time consuming to implement, and completeness is not the main goal set for this task.

5.3.2 Database creation

Having defined the conceptual schema, the next step was to create all the 12 tables and populate them, resulting in the final database. To create the model (which contains the database), MySQL Workbench[4] was used. This tool allows the quick creation of databases due to its graphic interface, which simplifies the process of creating tables and relationships between them. With MySQL WorkBench, users do not need to execute any SQL scripts in order to create tables, it performs this task automatically, adapting the script to the fields and options the user chooses during the creation step. Due to this

[4] `https://www.mysql.com/products/workbench/` (last visited on 02/05/2018)

simplicity, the creation of the schema defined earlier was relatively quick to perform.

5.3.3 Table population

After creating all the tables and defining the appropriate primary and foreign keys, the next and final step was to populate the tables with all the data from the two CSV files. With MySQL WorkBench, although it creates scripts for table creation automatically, users can also write SQL scripts themselves, and execute them. Because of this, it is possible for users to populate tables through this tool. But, in our case, since the data is in CSV files and MySQL WorkBench cannot fetch the data from these files, this process could only be done one line at a time. Due to the sheer size of both files, it is clear that this was an unfeasible approach, it could not be done manually.

The most efficient and evident solution for this, was to write a program that processes and extracts all the data from the CSV files, and then inserts all the data into the appropriate tables. Since only Python was used in the implementation of all systems until this point, we chose to use it once again. To access and alter the database created earlier (stored locally), Python's MySQLdb[5] library was used. With this library, after establishing a connection to the database, users can query and alter their databases by calling functions in the library, and passing to them the queries or scripts they want to execute.

The first thing to do was to process and extract the data present in the CSV files. The most difficult to implement was the processing of the fields that were kept in JSON. Having extracted the data from one line on both CSV files, this data had to be inserted into the correct tables. This step also involved overtaking some obstacles, due to the repeated names of people and jobs, which were certain to happen with a dataset of this size. The same actor can enter several movies, and, for instance, there are many directors in the dataset (same job, but different people doing it). To prevent these repetitions to happen in the database, we had to implement a history of the people and jobs already inserted into the database.

5.4 Corpora

Having created the database that will be used later, the final step needed before starting to implement the system is to create two corpora: one for training and one for testing. For our domain, we created a training corpus that consists of 208 questions, which cover almost all the different kinds of information present in the database.

As it can be seen from Figure 5.3, we included questions that address several fields, such as release dates, run-times, specific movie roles, profits generated by a certain movie, among other things, totalling 40 different question types. Questions that include information or entities not present in the database where not included in the corpus. In addition to this corpus, a test corpus was compiled. This corpus contains questions that were not created by us. Instead, we asked people to write questions about the cinema domain, filtered the ones that the database could not answer, and used the remaining ones for the test corpus. This corpus is much smaller than the training corpus, having a total of only 40 questions.

5.5 The Voting Model

There are many viable approaches to build a natural language interface, with the main component always being a **semantic parser**. These parsers take as input an utterance in natural language, and output a logical form that can be easily translated or transformed into a query, which in turn can be

[5] `http://mysql-python.sourceforge.net/MySQLdb.html` (last visited on 02/05/2018)

```
When was Mad Max 2 released?
How much profit did The Matrix Reloaded generate?
Who was the director of Braveheart?
What is the latest movie in which Adam Sandler enters?
What is the oldest movie produced?
Which movie was most recently released?
The movie The Avengers has how many characters?
What was the least profitable movie ever?
What is the longest movie ever made?
What's the number of spoken languages in American Sniper?
Meryl Streep has participated in how many movies?
Who entered the movie The Matrix?
What role did George Clooney play in Gravity?
How long does San Andreas last?
```

Figure 5.3: Sample of the training corpus

executed to provide an answer for the user's question. As such, the main differences between natural language interfaces lay in how the semantic parser is implemented.

We chose to use the so-called **Voting Model** (Bhagat et al., 2005) to implement a semantic parser in NLTK. The premise of this model is that, using statistical learning strategies, it achieves satisfactory results with very little training data. This is a very positive aspect, since many times it will be difficult to find sufficient training data for many domains. Besides this, and the fact that the model itself is not very complex, as it will be shown in this section, and simple to understand, led us to choose it over other models. In the following subsections, we will describe this model in greater detail.

5.5.1 Data representation

One the key concepts of this model is the concept of **frame**. It is what the parser outputs after parsing user input, and consists of a set of slot-value pairs [1], with each pair being called a "meaning element" or "frame element". Using our chosen domain (cinema), the parsing of the question "How many companies produced Quantum of Solace?", would have the following output:

```
TARGET=production_company AUX=count MOVIE=Quantum of Solace
```

In this case, the frame is composed of three frame elements, with each element having a different type, totalling three different frame elements, and their respective values:

- Type: `TARGET`, value: `production_company`;
- Type: `AUX`, value: `count`;
- Type: `MOVIE`, value: `Quantum of Solace`.

We chose to name frame element types only using capital letters, to be easier to distinguish them from the actual values. Another important aspect to note is that entities of the same type (movies titles or actor names, for instance), can all be grouped into the same category (Movie or Actor categories). With this categorization, it is simpler to represent meaning, by replacing entities that are not important to the semantics of the sentence as a whole, with their corresponding categories, which provide all the information that is necessary. But, the replaced values must be stored, for later substitution in the right places, making it possible to transform the resulting logical form into an executable query. For

example, for the same question ("How many companies produced Quantum of Solace?"), the frame element `MOVIE=Quantum of Solace` would become `MOVIE=MOVIE` and the movie title `Quantum of Solace` would be stored to later return to its original place, when the final frame is produced.

5.5.2 Training data

The training data for this model is a corpus made of sentences, with manually annotated frames for each sentence. In our case, we used the training corpus already created that was presented in Section 5.4. However, we still had to manually annotate each sentence with its respective frame.

```
What actors entered Titanic?
TARGET=actor_name;MOVIE=Titanic

Which characters were there on Goldfinger?
TARGET=character_name;MOVIE=Goldfinger

What are the languages spoken in Ocean's Eleven?
TARGET=spoken_language;MOVIE=Ocean's Eleven

Who was the director of The Great Gatsby?
TARGET=person_name;ACTION=directs;MOVIE=The Great Gatsby

Who starred in Batman Begins?
TARGET=actor_name;MOVIE=Batman Begins

What actor played the part of Jake Sully in Avatar?
TARGET=actor_name;ACTION=play_part;CHARACTER=Jake Sully;MOVIE=Avatar

Who entered the movie The Matrix?
TARGET=actor_name;MOVIE=The Matrix
```

Figure 5.4: Sample of the corpus for the cinema domain

Figure 5.4 shows examples of some tagged sentences that can be used (and were later used) to train a semantic parser on the cinema domain. As it can be seen, there are several different types of frame elements and values. Every question must have a `TARGET` frame element. This is the element that indicates to what is the user's question related to. The other frame elements provide additional information about what the user wants to know, such as, for instance, what movie, actor or director the user is referring to, if the user wants a numeric answer (a number or quantity) or a list of values (normally strings), among other things. A list with detailed explanations for each frame element type, and their respective values, can be found in Appendix D.

5.5.3 Training the parser

As stated before, this model uses statistical learning methods to decide which frame elements will be part of the final frame. More specifically, conditional probability models are used to calculate the probability of producing a slot-value pair f as output, given that the parser has seen a certain word or n-gram W in the input sentence [1]. With:

- $C(f_i \mid w_j)$ being the number of times the frame element f_i is seen as output in corpus, given that the sentence contains the word w_j;
- $C(f_i \mid w_{j-1}w_j)$ being the number of times the frame element f_i is seen as output in corpus, given that the sentence contains the bigram $w_{j-1}w_j$;

- $C(f_i \mid w_{j-2}w_{j-1}w_j)$ being the number of times the frame element f_i is seen as output in corpus, given that the sentence contains the trigram $w_{j-2}w_{j-1}w_j$.

The Voting Model calculates the following probabilities, for unigrams, bigrams, and trigrams, respectively:

$$P(f_i \mid w_j) = \frac{C(f_i \mid w_j)}{\sum_{k=1}^{n} C(f_k \mid w_j)} \tag{5.1}$$

$$P(f_i \mid w_{j-1}w_j) = \frac{C(f_i \mid w_{j-1}w_j)}{\sum_{k=1}^{n} C(f_k \mid w_{j-1}w_j)} \tag{5.2}$$

$$P(f_i \mid w_{j-2}w_{j-1}w_j) = \frac{C(f_i \mid w_{j-2}w_{j-1}w_j)}{\sum_{k=1}^{n} C(f_k \mid w_{j-2}w_{j-1}w_j)} \tag{5.3}$$

As it can be seen from the equations above, the model calculates the probabilities of correspondences between frame elements and n-grams. It is important to note that, other than immediate n-gram adjacency, the Voting Model does not record any other type of dependency between words in sentences, which makes this model very simple and easy to understand. These calculations are done for every frame element present in the training corpus, and all the values must be stored, to be later consulted. This concludes the **training step** of this model. Now the parser has all the information necessary to begin processing user input.

5.5.4 Sentence parsing

This is the final step, in which the parser uses the probabilities calculated earlier during training, to return the most likely candidates for the meaning elements in the final frame. The process for generating the frame for a given input sentence, is composed of two stages:

- Find the most suitable candidates for each word, bigram or trigram in the input sentence;
- Filter the set of candidates. The candidates that remain after filtering will be a part of the frame.

During the first of these two stages, given an input sentence, the model generates the set of all frame elements that correspond with each unigram, bigram or trigram in the sentence. These are the candidates that will later be filtered. To get to this set of candidates, the first step is to replace all the words or n-grams in the input sentence, that belong to a specific category, by the name of that category (for the cinema domain, “Avatar” would be replaced by “Movie”), and store the value that was replaced. After this is done, the second step consists in assigning a **weight** to each different frame element existing in the training corpus. The weight of frame element f_i can be calculated using the following expression:

$$Wt(f_i) = \sum_{j=1}^{n} P(f_i \mid w_j) + \sum_{j=2}^{n} P(f_i \mid w_{j-1}w_j) + \sum_{j=3}^{n} P(f_i \mid w_{j-2}w_{j-1}w_j) \tag{5.4}$$

With n being the number of words in the input sentence, and w_j, w_{j-1} and w_{j-2} words of the input sentence. The probability values above were already calculated and stored during training, as it was stated before. Because of this, the calculation is easy to perform, only involving the fetching of the correct values already stored (for words or n-grams not present in the corpus, the probability equals zero). This produces a value which measures how well, or how likely it is for a specific frame element to be part of the frame returned by the parser.

Having calculated all the weights, the model advances to its final stage, which consists of selecting from the list of candidates, the ones that are most likely to be the correct meaning elements. This selection is done gradually in three stages:

- **First stage** - in this stage, the selection is based on a heuristic observed in many domains. It was consistently observed that two frame elements with the same attribute almost never occur in the same sentence (e.g. `TARGET = actor_name` and `TARGET = orginal_title`). Using as basis this heuristic, what is done in this stage is, for each different attribute from the candidates, choose only the frame element with highest weight;
- **Second stage** - as in the previous stage, the selection is also based on a heuristic observed across several domains. It was observed that two frame elements with repeated values rarely occur in the same sentence (`ACTOR = Steven Spielberg` and `DIRECTOR = Steven Spielberg`). So, in this stage, for each different value from the candidates, only the candidate with the highest weight is chosen;
- **Final stage** - this stage consists mainly in applying a cutoff to the remaining candidates. It is difficult to reach the right cutoff, with it sometimes excluding more candidates than it should, and other times less. In Section 5.6.6 we will explain the cutoff adopted for our particular case.

After all the selection steps are done, the final output is returned by the parser: the frame with the frame elements most likely to be correct.

5.6 Natural language interface implementation using the Voting Model and NLTK

To implement a natural language interface with NLTK, using the Voting Model as its semantic parser, several steps had to taken taken. We will discuss these steps in greater detail.

5.6.1 Creation of named entity lists and question retrieval

Since NLTK does not have any NER system trained to recognize movies titles and actor, director or character names, the first step was to create lists with these names and titles, to be used later on the processing of sentences, being them sentences from the corpus or written by users. The names and titles are retrieved from the database using Python's `MySQLdb` library.

Having retrieved all the necessary entity names from the database, the next step was to store all the questions and their respective frames from the corpus. Both the questions and frames were stored in lists, conserving the order they appear on the corpus. This way it was easy to retrieve the frame associated to a certain sentence, or vice versa, since they share the same index in both lists. But, the questions and frames still had to suffer some changes, in order to be in the appropriate format for the training stage. Without these changes, the sentences would present a higher degree of variation, even in case where the semantics were similar. These steps serve to make these sentences more similar, increasing the model's overall performance.

5.6.2 Named entity recognition and replacement

The first change consisted of identifying and replacing entities with their respective categories (replacing "Brad Pitt" with "ACTOR", for instance). This was done on both questions and frames, with four categories of entities: **movie titles**, **actors**, **directors** and **characters**. This is a crucial step in

this model, since this is what allows the model to capture the semantics of sentences, regardless of what entities are mentioned. This is done resorting to the lists of entities extracted previously from the database.

In addition to these lists, a string matching algorithm must be used, to make sure the entities are recognized and replaced correctly. We chose to use the **Aho–Corasick** algorithm (Aho and Corasick, 1975), which returns all the occurrences of strings contained in a dictionary, on the input text. Python's `ahocorasick` library provides an implementation of this algorithm, and so, this library was used on this step. Since there are four categories of entities, in our case, four dictionaries had to be created (one for each entity category). Seeing that the Aho-Corasick algorithm returns all the occurrences in each dictionary, some of these occurrences correspond to substrings of words, which are not the correct matches. Because of this, additional filtering of the matches returned by the algorithm had to be made, and, for each dictionary, only the longest occurrence is kept. Additionally, the string corresponding to this occurrence in the sentence, must be delimited by white space or punctuation, either on the beginning or the end. If this does not happen for the longest match, the second longest is checked to see if this conditions are true, and so on. Having identified the most suitable matches returned by the Aho-Corasick algorithm, the entities are substituted by their names, on both the questions and frames in the corpus, and the original values are stored.

To illustrate this process, consider the question "What actors entered Toy Story 3?". For this sentence, the Aho-Corasick algorithm returns matches for the movies and characters dictionaries. More specifically, the matches for movies are "Toy Story" and "Toy Story 3", and for the characters there is a single match, "Story". Next, we only keep the longest match for each dictionary, leaving only "Toy Story 3" and "Story". Finally, starting with the longest of the remaining matches, we check if it is delimited by white space or punctuation. This is true for "Toy Story 3", thus it will be replaced by "MOVIE" in the sentence and its respective frame, and the value "Toy Story 3" will be stored to be later put on its original place. This leaves only the 'Story" match, but, since 'Toy Story 3" was be replaced by "MOVIE", this match is no longer present in the sentence, meaning no further changes to the sentence will be done.

5.6.3 Stopword removal

Having replaced the entities with the respective categories, the next step is to remove stopwords from all the questions in the corpus. With the removal of these words, no information is lost regarding the meaning of the sentence. Additionally, it contributes to smaller and simpler sentences. To perform this task we used NLTK's English stopword set, and removed from the sentences any word contained in this set.

5.6.4 Stemming

The last change to be made to the sentences in the corpus was to stem the remaining words, minus the categories that replaced the named entities detected earlier. This allows for greater uniformity across sentences that use similar words, but with different prefixes or suffixes, or verbs in different tenses. Stemming can also be performed with the help of NLTK. In our case, we chose one of the already implemented stemmers, the Snowball Stemmer.

5.6.5 Probability calculation

Having made the three changes presented before, the corpus is now stored in its final format, ready to be used to train the model. The next step is to calculate all the probabilities presented in Section 5.5.3,

for all the unigrams, bigrams and trigrams present in the corpus. Thus, it is necessary to store these n-grams for it to be possible to perform these calculations. To get the n-grams mentioned, we used NLTK's `ngrams` function, which, given a list of words as input, returns a list with the desired n-grams (for any n given by the user). With the n-grams extracted from the sentences in the corpus, the probability values are calculated and stored in dictionaries (one for each type of n-gram). This provides a way to easily access each of these values, by assigning an unique identifier to each value.

5.6.6 User input processing

After calculating all the probabilities, the system is ready to receive user input. To implement this part of the model, the same steps had to be taken when processing the corpus, namely:

- Replacing named entities with their respective categories. As when the corpus was processed, this was done using the lists of entities extracted from the database, and the `ahocorasick` library;
- Stopword removal, using NLTK's English stopword set;
- Stemming, using NLTK's Snowball Stemmer.

Once this is done, the weights mentioned in Section 5.5.4 are calculated, creating a list of frame element candidates, ordered by descending weight. Next, the candidates are filtered, following the stages presented in Section 5.5.4. Regarding the final stage of filtering, after some experimentation with the data and some input examples, we managed to create a heuristic that calculates the cutoff dynamically, depending on the list of candidates prior to this step. The cutoff value is equal to the average weight of all the candidates in the candidate list, up to this point. Only the candidates with weights higher than the average calculated are kept, the others are discarded. After trying different approaches, this heuristic displayed the most consistent results, and because of this, it was the one chosen to calculate the cutoff.

5.6.7 Query generation and execution

Having the final set of most likely frame elements, the next step consists of converting the frame into the correct query. In our case, this was done through a simple mapping of each different set of frame elements, to the corresponding query. For instance, frames of the form (corresponding to questions like "How many actors entered the movie X?"):

```
TARGET=actor_name;AUX=count;MOVIE=MOVIE
```

Can be mapped to the following query:

```
  SELECT COUNT(*) FROM
(SELECT person_name FROM persons, jobs, activity_codes, acting, movies
WHERE jobs.job_name = "Actor" AND jobs.job_id = activity_codes.job_id
AND activity_codes.act_id = acting.activity_id
AND persons.person_id = acting.person_id
AND acting.movie_id = movies.movie_id AND movies.title = %s) newtable
```

The query is missing only one argument, which is, in this case, the title of the movie. With this mapping, we only need to replace the `%s` in the query with the corresponding movie title. This happens for all the mapped queries, they only lack one or two necessary arguments, which can be easily put

in place. This mapping was done for all the combinations of frame elements that can be seen in the corpus. Since the corpus contains 208 questions, which, although it is not a very big number, adds up to a considerable number of frame element combinations. This made the task of mapping frames to queries somewhat time consuming and repetitive, but not very complex.

With larger corpora, this could become a problem, due to the large number of frames to map to queries, but, it is the simplest approach, not involving any training and classification processes. Since the main goal with this task was to have a functioning natural language interface, and try to keep it as simple as possible, some aspects could be improved, such as this stage of conversion of frames into queries. But, this was not the main objective, and because of that, we chose to do this direct mapping of frames to queries. After converting the frame to the proper query, the only thing left to do is execute the query, resorting to the `MySQLdb` library, and show the results to the user.

5.7 Natural language interface using the Voting Model and Bluemix

Unlike what the title of the previous section states, in this section we will not discuss the implementation of a natural language interface using the Voting Model and Bluemix, since it was not implemented. This was not possible because Bluemix does not offer enough functionality for this to be possible, as it will be explained in this section. Instead, what will be discussed in this section has to do with what would have to be changed for it to be possible to implement using Bluemix.

Having in mind what was presented in Section 5.6, one of the things that could be changed, is the way named entities are identified. As it was stated on that section, this was made possible by using lists with the named entities to be matched, along with the Aho-Corasick string matching algorithm. Bluemix, besides the Watson Assistant service (covered in Section 5.8), provides the NLU service which, among other things, recognizes named entities in text. As so, this is one of the changes that would have to be made to implement the Voting Model using Bluemix. Other aspects that could be changed have to do with tasks implemented using NLTK, that include:

- N-gram retrieval from sentences;
- Stopword removal;
- Stemming;

Unfortunately, Bluemix does not provide any service capable of performing the tasks listed above. Because of this, it is not possible to implement a natural language interface using only Bluemix's NLU service. Some of NLTK's functionality must be used to performed the three tasks mentioned. But, it is still possible to use the NLU service to perform NER. This service recognizes entities from a wide array of domains, and as such, it is likely that it can recognize cinema related entities effectively.

5.7.1 Testing the NLU service on the cinema domain

To understand if this change was indeed possible for our domain, we decided to test the NLU service on the cinema domain. This test consisted of passing all the sentences in the corpus created for NLTK's implementation of the Voting Model, and check what entities and respective categories were returned by the service. To measure the service's performance, the same evaluation strategy was used when evaluating NER using NLTK and the NLU service. A detailed explanation of this evaluation strategy can be seen in Section 4.5.1.

Accuracy	Precision	Recall	F-measure
0.8594	0.9141	0.2120	0.3441

Table 5.1: Test results using the NLU service to perform NER on the cinema domain

Looking at the results in Table 5.1, the values that stand out are the **accuracy** and **precision** values. With an accuracy of nearly 86%, this means that for the entities the NLU service recognizes, on 86% of them both the entity type and text boundaries are identified correctly. An even greater value is achieved in precision, with 91%. This means that only 9% of all entities detected by Bluemix were assigned a wrong type, or the text boundaries were not correct. However, there is a significant drop in recall, reaching only 21%, approximately. This means that the number entity types plus the number of correctly identified text boundaries is only a fifth of the total number of entity types and texts (2 * the number of named entities, one type and text for each named entity). Because of this, it can be concluded that there are many entities the service does not identify throughout the corpus (nearly 80% are not identified), but, the ones it manages to identify, are almost always identified correctly, both on the category and the text boundaries.

5.7.2 Conclusions

Having in mind the results obtained with Bluemix's NLU service, it can be said that it would not be viable to replace the NER method adopted in Section 5.6.2. The service misses too many entities for it be a part of a functioning natural language interface for the cinema domain, despite the high accuracy and precision shown during evaluation. If the service had shown much higher recall, then it could be possible for it to perform NER, instead of using entity lists and a string matching algorithm.

As these results are for the specific domain of cinema, no other conclusions can be taken from this test. Maybe on other domains the service could achieve a much higher recall percentage, whilst maintaining precision and accuracy. This would indicate that, not only the service was able to identify entity types and text boundaries correctly in a consistent manner, but also for almost all entities present in the corpus. If this would happen for another domain, then the NLU service could be used to perform NER, replacing the approach presented in Section 5.6.2.

5.8 Interface implementation using the Watson Assistant service

As it was mentioned in Section 2.2, one the services that could be helpful for this task is the Watson Assistant[6] service, which allows developers to specify different types of entities and intents in user input, and also construct dialog flow. All of this can be done through a web application, as it can be seen in Figure 5.5, or through the service's API, available in several programming languages[7].

Having in the mind the logic of the Watson Assistant service, even without knowing all the details that would be necessary for the implementation of a working natural language interface, it is clear what the first two steps would be:

- **Create intents** - intents express the main goal that the user wishes to achieve with their input question or sentence. Each intent must have some sentence examples, at least 4 or 5 (the bigger the number, the better), for the service to be able to recognize the user's intents more easily;

[6] `https://console.bluemix.net/catalog/services/watson-assistant-formerly-conversation?taxonomyNavigation=watson` (last visited on 02/05/2018)

[7] `https://www.ibm.com/watson/developercloud/assistant/api/v1/curl.html?curl` (last visited on 02/05/2018)

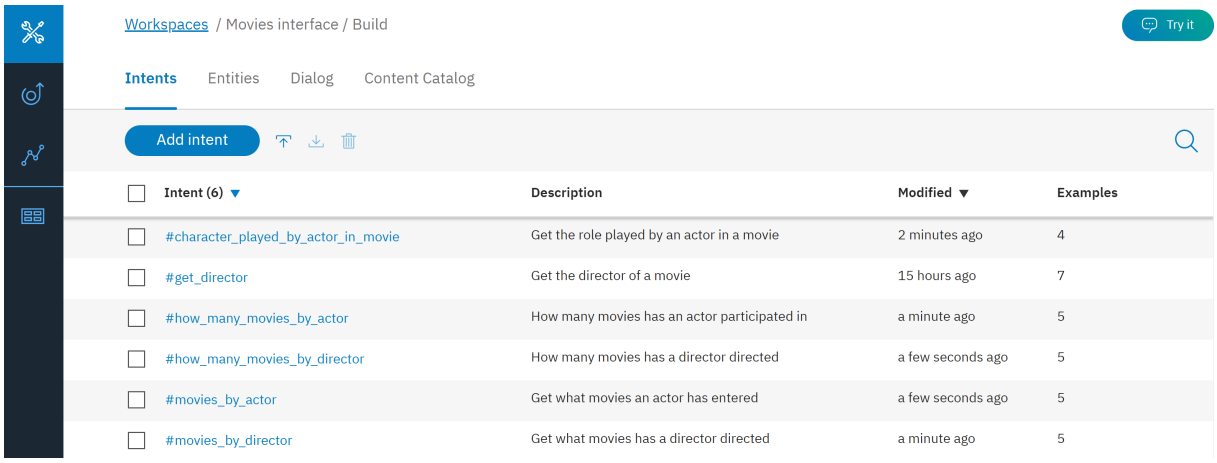


Figure 5.5: Interface of Watson Assistant's web app

- **Add entity types** - the next step is expressing which types of entities the service should be able to recognize in user input. For each type of entity, much like what happens with intents, some examples must be provided (again, the more the better).

There is still the "Dialog" tab on the web application, as it can be seen from Figure 5.5, that allows developers to edit the conversation flow, creating conversation nodes, defining what the system should answer in each node, and expressing conditions for the conversation to go to a specific node (if a specific intent is detected, for instance). But, this aspect of the service is not relevant for our task, since there will be no significant conversation flow, in our case. We do not want a predefined response when the service recognizes a specific intent. Our main interest in this service, is the ability to correctly identify the user's intents, and what entities he mentions.

Within the "Content Catalog" tab, users can find some predefined intents, trained on common questions or domains that users may refer to (banking, bot control, costumer care, eCommerce, general conversation topics, insurance, telephony services, utility services). Once again, this portion of service is not of great interest for our task, since none of these topics are related to the cinema domain. Additional details about the creation of intents and entities will now be presented.

5.8.1 Intent creation

Regarding the specific domain of cinema, we decided to create some intents that apply to the questions used on the training corpus mentioned in Section 5.4. As it can be seen from Figure 5.5, just to exemplify some of the questions that can be asked in this domain, these are some of the intents that were defined, out of a total of 40 intents:

- `character_played_by_actor_in_movie` - this intent addresses questions in which the user wants to know the role played by an actor on a specific movie. An example of sentence used to train this intent is: "What was the role taken by Brad Pitt in the movie Ocean's Eleven?";

- `get_director` - intent for questions in which the user wants to know the director of a movie. Question example: "Who directed The Matrix?";

- `how_many_movies_by_actor` - this intent relates to questions where users want to know in how many movies has an actor entered, until now. One of the questions used to train this intent was: "In how many movies can I see Keanu Reeves?";

- `how_many_movies_by_director` - this intent is very similar to the last one, with directors instead of actors. One of the examples used for training was: "How many movies has George Lucas directed?";
- `movies_by_actor` - relates to questions where users want to know in what movies has an actor entered. Question example: "Tom Cruise has played a part in what movies?";
- `movies_by_director` - very similar to the previous intent, with the difference it is applied to directors. An example used for training is: "Tell me the titles of all the movies directed by Sam Mendes".

These are just some of the many intents that were defined. Figure 5.6 shows the examples used to train the `movies_by_actor` intent. The remaining intents were created fairly quickly and with ease. All that had to be done was to name each intent as it was created, and then provide sentence examples for that intent. For all the intents defined, a minimum of 5 examples were given for each one, reaching a maximum of 7 on one of the 40 intents defined, the `get_director` intent.

Developers must try to make what users want to express with each intent very clear, through the examples used to train each intent. If each intent is clear and unique, the chance of recognizing the correct intents in user input increases. Because of this, we tried to provide as much variety as possible in the examples used, to increase the chances of the service recognizing the user's intent correctly.

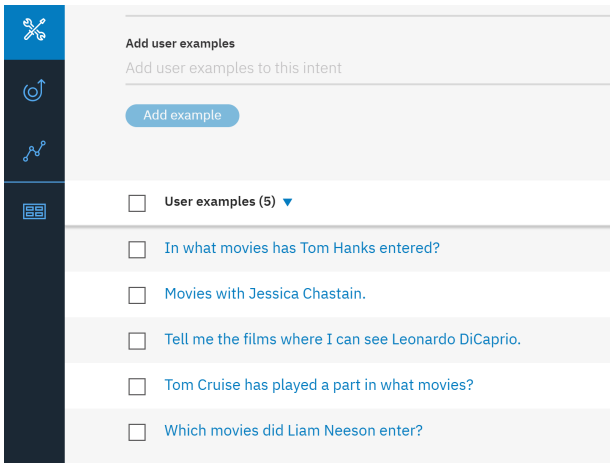


Figure 5.6: Examples used to train the `movies_by_actor` intent

5.8.2 Addition of entities

As what happened with intents, although to a smaller extent, there are several entity types that the service could be trained to recognize. In our case, there were four types of entities that needed to be recognized: **Movie**, **Actor**, **Director** and **Character**. For the natural language interface to function properly, a large amount of examples had to be provided for each of these categories, since the service does not recognize any entity of these types if the developer does not provide examples. These examples, in our case, much like what happens with intents, can be provided through the online platform the service provides, but, due to the sheer number of examples, this approach would take too much time. The solution was to use the service's API, which allows developers to define entity examples much faster. Using the API, a total of 100 000 examples were passed on to the service (the maximum amount of entity examples the service allows), all of them coming from the cinema database created earlier. These examples consisted only of the named entities present in the database, and were passed to the service

in lists. The number of examples present in the database was bigger than 100 000, meaning that some examples were not passed on. We chose not to use all the characters present in the database, due to the fact that, in our case, there are less intents involving characters than actors, directors or film titles.

This step is crucial to the success of a natural language interface using this service, since without a great number of examples, the service will likely not be able to recognize the entities mentioned in questions, and thus not be able to answer most of them.

5.8.3 Receiving and processing user input

Having defined all the necessary intents and entities for the service to recognize, it should be ready to receive user input, stating on its response which entities were recognized, and what is the user's intent. With this information, we have all that is necessary to generate the correct query to answer the user's question. This is a simple step (but time consuming, depending on how many intents were stated), since, if designed correctly, each intent should address a very specific type of question. This way it is possible to answer any question regarding a particular intent with an unique query. Having this in mind, the necessary steps to create a functional natural language interface were the following:

- Map each intent to a unique query, lacking only the arguments needed to execute it (movie title or actor name, for instance);
- Wait for user input;
- After the user types his question, pass it to the Watson Assistant service, through the available API;
- Check the service's response and store the intent and entities detected;
- Fetch the query mapped to the intent detected earlier, and fill in the missing values. This is done through the entities stored in the previous step;
- Execute the query, and finally, show the results to the user.

The first step has to be done once, before the system receives any user input, as this is a crucial step for it to be able to answer any question. The remaining steps are part of a continuous loop, which waits for user input, and in the end returns the query results to the user. One important aspect to note is that this process relies heavily on the service's correct identification of the user's intent, and the entities he mentions on the question. If the service returns an incorrect intent, then the system will not be able to execute any query, or it will execute some query, but not the one it was supposed to. Additionally, if the service does not identify all the entities, or does it with flaws, once again the correct query may not be executed, or it will be executed with a wrong argument, leading to incorrect responses.

5.8.4 Conclusions

Having in mind all that was said in this section, it can be said that it is indeed possible to implement a natural language interface for database, for the cinema domain, using as knowledge base the database presented in Section 5.3, and Bluemix's Watson Assistant service. With it, developers do not need to focus on implementing a semantic parser, which is a much more complex and difficult task than implementing what was presented throughout this section. By providing the service with concise and clear intents, which can be easily distinguished between one another, and a complete list of entities for it to recognize, the service outputs all the necessary information to generate the correct query, in most cases.

The first experiments with the service showed us that, most of the time, it identified the user's intents correctly. In addition to this, it was able to identify correctly most entities mentioned in sentences (if these entities are in the list of examples passed to the service). A more thorough evaluation methodology will be presented later, along with the results achieved. But, these preliminary results are a good indicator of the service's success. In spite of this, with big amounts of intents and entities, as it happens in our case, some problems arise. In the cinema domain, it is possible for a person to be an actor and director. If a user references this person in a question, the service will return the name of this person, but belonging to two types of entity: actor and director, in this case. This is a problem that would arise, not only using this service, but with any other approach to build a natural language interface. In this specific case, this problem is solved due to the fact that queries expect entities of a certain type as arguments, depending on the intent that was identified, and as such, the right type of entity is always chosen in these cases.

Another problem that arises is when there are multiple entities of the same type returned by the service. In this case, there is no way of picking the right one without asking the user which one is correct, which is not done in our case. Due to this, when this problem happens, incorrect results will most likely be returned by the interface. Finally, another problem that could come up is with the definition of overlapping intents, that use very similar sentences as examples. We do not know how the system would behave if this happens, but, most likely, it will not be able to recognize user intents correctly as often.

5.9 Evaluation

5.9.1 Methodology

To evaluate the two systems developed, a slightly different strategy was employed, than the ones used for the two previous tasks. The strategy adopted was to ask the questions present in the test corpus to each interface, and check the logical expression (composed of a variable number of frame elements) that was created, not letting the systems transform this logical expression into a query and execute it. This final step was not done because the conversion of each logical expression was based on a mapping to an SQL expression. Instead of having each question answered, what matters most in this case is to analyze the logical expressions returned by both systems, and not the actual answers to the questions.

The measures used to evaluate the responses are somewhat different than the ones used with the two previous tasks, with two of them (precision and recall) being the same as the ones used by the authors of the Voting Model to evaluate their system [1]. The measures considered are the following:

- **Precision** - consists in the percentage of frame elements returned by the interface that were correctly filled, out of the total number of frame elements returned by the interface;
- **Intent precision** - slightly different from the previous measure, consists in the percentage of correct intents identified by the interface (with each sentence having only one intent). On the interface developed with Bluemix, an intent consists of only one frame element per sentence, and it captures the type of question being asked and its semantics. The remaining frame elements are dedicated to named entities mentioned in the question. On the other hand, on NLTK's interface, due to the use of the Voting Model, an intent can be made up of a variable number of frame elements, with a minimum of one, as it can be seen in Appendix D.
- **Recall** - consists in the percentage of correctly filled frame elements, out of the number of frame

elements present in the test corpus. As such, this measure indicates the percentage of frame elements present in the test corpus that the interface was able to correctly fill in;

- **Number of correct sentences** - it is the number of questions in which the interface's response contained all the frame elements in the test corpus, for that particular sentence. This does not take into account additional frame elements returned by the interfaces, since these can be discarded during the transformation of the logical expression into a query. Thus, this number indicates how many questions could be effectively answered by the system, since all the necessary frame elements are filled correctly.

5.9.2 NLTK and the Voting Model

Unlike what happened during the evaluation of both NER systems developed, in this case the execution times were very small. It took less than 2 seconds for the interface developed with NLTK to train and process the 40 questions present in the test corpus. This is due to the relatively small size of the training corpus (208 questions) and the low number of test questions. Table 5.2 shows the results achieved after the interface implemented with NLTK processed the 40 questions contained in the test corpus. The number of frame elements returned is the total number of frame elements the system output for the 40 questions in the test corpus. On the other hand, the test corpus, after being manually annotated, displayed a total of 113 frame elements.

Number of frame elements returned	129
Number of frame elements in the test corpus	113
Number of correct frame elements returned	83
Number of correct intents returned	31
Number of correct sentences	23
Precision	0.6434
Intent precision	0.7750
Recall	0.7345

Table 5.2: Test results of the natural language interface implemented with NLTK

As it can be seen, the interface output a total of 129 frame elements, whilst the original number of frame elements is 113. This means that the system generated 16 more meaning elements than what was supposed to. Out of the 129 frame elements returned by the interface, 83 of them were filled in correctly, leading to a precision value of approximately 64,5%. With a small training corpus like the one used for this task, this is a very positive result. And, taking into account only the number of frame elements present in the corpus, the value is even higher, reaching 73,45% for recall. This means that only approximately one quarter of the test corpus was not guessed correctly by the interface.

Another interesting fact is that the system managed to identify the correct intent for 31 of the 40 questions in the corpus, leading to a precision of 77,5% on the task of intent identification. Despite this percentage, the system only managed to output all the necessary correct frame elements in 23 out of the 40 questions. This indicates that the interface had more difficulties in identifying named entities correctly, than the users' intents. This may have happened due to the limited number of training examples, or simply because the named entities mentioned in some sentences were not stored in our database, meaning they would be impossible to identify regardless of the amount of training examples provided. This happened for instance, for the question "In the film Tommy, who played The Acid Queen?", which is part of the test corpus. Unfortunately, there is no information on our database about the movie "Tommy", and so, it is impossible for the interface to provide an answer for this question.

5.9.3 Bluemix

Regarding the total execution time (excluding training, which is done internally as the service receives training examples), it was greater than what was achieved with the previous system. In this case, it took approximately 24 seconds for the service to output all the intents and entities present in the 40 questions. Although this is not a very long period of time, it is still a considerable difference from what was achieved using NLTK and the Voting Model. But, this gap in execution time was expected, since the Watson Assistant is an online service, whilst with the Voting Model all the information and processing is done locally, leading to much faster results. With faster internet connections, faster times would likely be achieved, although they would not be as fast the ones achieved with NLTK and the Voting Model.

Number of frame elements returned	110
Number of frame elements in the test corpus	77
Number of correct frame elements returned	67
Number of correct intents returned	35
Number of correct sentences	30
Precision	0.6091
Intent precision	0.8750
Recall	0.8701

Table 5.3: Test results of the natural language interface implemented with Bluemix

Moving on to the evaluation results, Table 5.3 shows the results obtained for the interface based on Bluemix's Watson Assistant service. Having a first look at the results and comparing them with the ones obtained with NLTK, what stands out the most is a slight drop in precision, but also a significant increase in both intent precision and the total percentage of correct frame elements. Having a closer look, we can see that service output a total of 110 frame elements (including both intents and entities), a number slightly lower than what the Voting Model returned. This difference happens due to the fact that with the Watson Assistant service there is only one frame element dedicated the sentence's intent, as opposed to the Voting Model where there can be multiple frame elements per sentence dedicated to its intent. Also, the number of correct frame elements was lower in this case, with only 67, leading to a precision of almost 61%. This is a similar value to the one obtained with the Voting Model, although somewhat lower, but still a positive result.

Despite its lower precision, the Watson Assistant service managed to identify the sentences' intent more frequently, with 35 out of the 40 sentences having the correct intent attributed to them. This led to an 87,5% of precision, on the intent level. This was a 10% increase in comparison with the Voting Model, which is a significant boost. One factor that could lead to this improvement is that, as it was mentioned earlier, each sentence has only one frame element dedicated to its intent, unlike what happens in The Voting Model. As such, with Bluemix, the system only needs to output one correct frame element to correctly identify the intent, whist with the Voting Model there can be several frame elements dedicated to an intent. Additionally, in the Voting Model, each frame element consists in a type/value pair, meaning that there are two "slots" to fill in, increasing the complexity of this task.

The elevated number of correct intents reflects itself on the number of correct sentences, increasing from 23 using NLTK and the Voting Model, to 30 using the Watson Assistant service. This also lead to an increase in recall, as the system managed to output correctly 87% of the frame elements present in the test corpus.

5.9.4 Conclusions

Having in mind the results presented in the two previous subsections, we can draw conclusions about some key aspects.

Execution time

Although this is not as crucial as with the two previous tasks, due to the small size of the corpus used for testing, it is still always an important factor to take into account. As it was expected, the Voting Model and NLTK achieved better results, with an execution time of roughly 1,5 seconds, which also includes the model's training stage. For the same 40 sentences, Bluemix managed to process them in 24 seconds, which grants the Voting Model and NLTK a clear advantage over the Watson Assistant service.

Performance

Comparing the values obtained for both interfaces, a slight advantage in performance can be seen when the Watson Assistant service is used. With this service, a sizable increase in intent precision, number of correct sentences, number of correct intents, and recall, can be seen. Despite the small drop in precision, the remaining measures presented better values for Bluemix. As such, weighing all these measures, we can conclude that the Watson Assistant service possesses an advantage, in terms of performance, on the Voting Model alongside NLTK.

Robustness of the solution

This aspect is not related to neither of the already mentioned facts, but it is still an important factor to take into account in this particular task, since it is the most complex. Upon implementing the two interfaces, we can say with certainty that the one that uses the Watson Assistant service is the least complex, in terms of code. Since all training is done by the service itself, all that we needed to do was pass it the entities it should recognize, and define manually the intents via the service's web application.

But, with this simplicity comes a disadvantage in relation to the Voting Model. As it was said previously in this section, only one frame element is dedicated to a sentence's intent when using the Watson Assistant service, whereas with the Voting Model, there can be multiple attribute/value pairs (frame elements) dedicated to the intent. This means that, with Bluemix, no granularity can be achieved in terms of intents. Any variation in what the user wants answered, even if the "main" intent is the same (only the first movie an actor has entered, instead of the full list, for instance), means that a new intent must be created. Considering this example, for NLTK, if the user wants, for example, the full list of movies where Angelina Jolie has entered, the following frame will be generated:

```
TARGET=original_title;ACTOR=Angelina Jolie
```

But, if the user only wants the first movie she has entered, then frame becomes:

```
TARGET=original_title;AUX=first;ACTOR=Angelina Jolie
```

As it is shown by these two examples, with NLTK we can define "sub-intents" (the `AUX` frame element, in this case), which provide the extra granularity and do not alter the "main" intent (the the `TARGET` frame element). On the other hand, for the Watson Assistant service, there are two separate intents. For the full list of movies, the intent would be `get_movies_by_actor`, whereas for the only the first movie, it

would be `get_first_movie_by_actor`.

Although this works when there is a relatively small number of intents, as this number increases, this solution is not appropriate. This makes the interface lack in robustness and become "hard coded", with developers having to define a new intent every time they want to introduce variations in existing intents. The Voting Model, in this aspect, provides more solidity, since it allows for the definition of any type of frame elements developers want to, meaning that there can be several degrees of granularity in an intent. As what happened with our interface, there can be several types of frame elements that make up an intent. In our case, there was a "central" frame element (the `TARGET`) that indicates what should be extracted from the database. The additional frame element types are "sub-intents" that provide the extra granularity for some queries. With this granularity, the model gains robustness because it can use existing frame element types for small intent variations, only being necessary to add or change the "sub-intent" frame elements.

Since the Watson Assistant service always returns only one intent per sentence, there is no way to implement this granularity using this service. The only thing that could be done to tackle this problem would be to train classifiers (outside of Bluemix) to identify the "sub-intents", and let the Watson Assistant recognize only the main intent. Adding all up, we can say that the Voting Model along with NLTK provides additional robustness, having the advantage over Bluemix in this aspect.

Documentation

Much like happened with the two previous tasks, the situation is the same for this case. Both the Watson Assistant service and NLTK have extensive documentation, but in truth, NLTK's documentation was not consulted for this task, since it involved only simple NLP tasks. Regarding Bluemix's Watson Assistant service, as what happened with the NLC and NLU services, lacked some detail in certain areas (maximum number of entity examples, for instance), which were only clarified after some experiences with the code.

Community support

Due to NLTK's high popularity in the NLP community, as it was said before, online searches always provide more information about NLTK related doubts and problems than with any specific service that Bluemix offers.

Final remarks

Having in mind all the factors presented before, we can say that each solution as its advantages and disadvantages. On one side, Bluemix provides the Watson Assistant service, that allows users to quickly define intents and provide entity examples, with no training required on the developer's end. This contributes to a reduced level of code complexity and thus, a simpler solution. But, with simplicity comes a lack of robustness for future additional intents, or larger and more complex domains. This would require the definition of a large number of different intents, which would unfeasible at some point.

On the other hand, NLTK provides this additional level of complexity and solidity, which can become useful for large training corpora. This leads to an increased code complexity, involving training the model, something which developers did not have to worry with Bluemix. In terms of testing results, as we showed before, Bluemix has an edge over the Voting Model, achieving a higher intent accuracy and

a greater number of questions that could have been answered.

Because of this, there is no clear choice between the Watson Assistant service, and NLTK and the Voting Model. For developers who want a simple and easy to implement solution, the Watson Assistant service is the best way to go. For small domains, with a reduced number of training examples, Bluemix is also the most appropriate choice. But, if a more robust solution is necessary, maybe it is not the most correct choice. Also, using Bluemix, with more training examples, the likelihood of intents with very similar sentences given as example increases, which could lead to difficulties in returning the appropriate intent. In this case, the Voting Model would probably deal better with these overlaps due to the possibility of defining "sub-intents". In our case, the Watson Assistant service would be the best choice, as it can be seen by the results achieved, but, this may not be the case for all domains.

5.10 Implementation using other toolkits

Much like what was done in the previous two chapters, in this section we will discuss how a natural language interface, using the Voting Model, could be implemented using Stanford CoreNLP and SpaCy.

5.10.1 Stanford CoreNLP

Recalling what was said in Section 5.6, the implementation of the Voting Model did not rely on complex NLP tasks. With NLTK, we only had to perform three NLP tasks:

- Stopword removal;
- Stemming;
- N-gram retrieval.

As such, the only thing that would change from NLTK's implementation, to this one, would be the execution of these three tasks. After researching online, we came to the conclusion that Stanford CoreNLP does not possess any annotator dedicated to stopword removal. The only way users could perform stopword removal would be if they implemented an annotator responsible for that particular task, which may not be very attractive to users due the additional level of code complexity.

On the other hand, regarding stemming, unlike what happened for stopword removal, Stanford CoreNLP provides its users with the `pos` and `lemma` annotators, which can be used to stem words. To do this task, all that developers must do is create a pipeline, and then specify which annotators they want the pipeline to consider (the `pos` and `lemma` annotators, in this case).

Finally, regarding the task of n-gram retrieval, it can be said that Stanford CoreNLP does not include this functionality. Once again, users would have to implement this task themselves. It would not be very complex, but, once again, it is a disadvantage in relation to NLTK, where these tasks can be easily performed by users.

Since we did not use any NER algorithm or model that comes with NLTK, only lists with entity names, that part of the natural language interface would remain equal, and as such, it will not be discussed in this section. With all this, we can say that it would be harder to implement the Voting Model using Stanford CoreNLP than it was NLTK. The fact that users do not have access to some simple NLP tasks and they would have to implement them themselves gives NLTK an advantage over this toolkit.

5.10.2 SpaCy

Regarding the three tasks mentioned in the previous subsection, after researching online, it can be said that SpaCy provides ways for users to perform all of them. For the first one, the removal of stopwords, SpaCy provides the `Defaults.stop_words` set, that contains all the stopwords for the English language. With this set, the removal of stopwords from sentences becomes an easy task. As for stemming, users can perform this task quickly with the `Lemmatizer` class. By calling the `lemmatizer` method and passing a word and its PoS tag, it returns the word's lemma.

Finally, for n-gram retrieval, SpaCy does not provide any function or class meant to perform this task. But, the `textacy` library provides this functionality. It was build on top of SpaCy, and provides additional functionality, such as n-gram retrieval. Using this library, users can call the `textacy.extract.ngrams` function, and it will return all the n-grams of the sentence passed to it, for the value of `n` specified by the user. As such, since all three tasks mentioned can be performed with SpaCy, we can say that a natural language interface for database, using the Voting Model as its semantic parser, could be implemented with this toolkit.

6

Conclusions and Future Work

6.1 Summary of the Dissertation

IBM[1]'s Watson[2] has gained popularity around the world since 2011, when it managed to win *Jeopardy!*[3] against human opponents. But, since then, many changes have been made to IBM's system, and now it is capable of performing of performing several NLP tasks. Some of this functionality is made available to developers through IBM's online platform, Bluemix[4], which can be used to integrate Watson's NLP capabilities in existing software. The NLP tasks that can be implemented with Bluemix can also be implemented with many other already existing NLP toolkits, meaning that a comparison can be made between Bluemix and these other toolkits. To our knowledge, no comparison of this kind was ever made.

Motivated by this fact, we set our goal as to perform this comparative study. To achieve this goal, we decided to implement three different NLP tasks using Bluemix, and another commonly used NLP toolkit, NLTK. The three tasks chosen were: sentiment analysis, NER, and the implementation of a natural language interface for database. Regarding sentiment analysis, a movie review corpus, and a Twitter[5] corpus, were used to train and test the systems. On NLTK, this task was implemented with the help of a Naive Bayes classifier, with four different feature generation functions. On the other hand, for Bluemix, the NLC and NLU services were used. After evaluating both systems, we came to the conclusion that better, and faster results were achieved with NLTK.

Moving on to the second task, the implementation of a NER system, the NLTK implementation was based on CRFs, which are already included in the toolkit's resources. The CRF model was trained with the GMB[6] corpus, which contains several different different types of named entities. On Bluemix, this task was implemented with the NLU service. Since some of the categories recognized by this service

[1] `https://www.ibm.com/pt-pt/` (last visited on 02/05/2018)
[2] `https://www.ibm.com/watson/` (last visited on 02/05/2018)
[3] `https://www.jeopardy.com/` (last visited on 02/05/2018)
[4] `https://www.ibm.com/watson/products-services/(lastvisitedon02/05/2018)`
[5] `https://twitter.com/?lang=pt_pt` (last visited on 02/05/2018)
[6] `http://gmb.let.rug.nl/` (last visited on 02/05/2018)

were different from the ones present in the corpus, these could not be considered during evaluation, there was no possible "translation" between the entities recognized by the service, and the ones present in the corpus. After evaluating both solutions, we concluded that with NLTK users can achieve quicker and better results, in terms of performance, than with the NLU service, provided they have a corpus for NER. But, if users do not possess this corpus, then it is preferable to use Bluemix, since no training is required. As such, there is no clear choice of better toolkit to use, in this case.

Finally, for the last and most complex task, before the implementation of any natural language interface, we had to choose a domain, and build a database to be later used by both systems. We chose the cinema domain, and built a database using the TMDB 5000 Movie Dataset[7], available on Kaggle[8]. Additionally, a training and a test corpus were created, with the former being composed of 208 questions, whilst the latter contains 40 questions. With NLTK, we developed the natural language interface by implementing from scratch a semantic parser based on the Voting Model. This model is based on a statistical learning approach, which calculates the probabilities of specific meaning elements being part of a given sentence, and outputs the most probable set of meaning elements for the sentences it receives as input.

On the other hand, for Bluemix, no semantic parser was implemented. Instead, we used the Watson Assistant service to define intents and provide exampled of names entities for the cinema domain. Having evaluated both systems, we could see that Bluemix achieved the best performance out of the two, but this does not mean that it is always the most appropriate choice for this kind of task. The Watson Assistant service is very useful for small domains, with a low number of training examples, as it was our case. It is also very simple to use, and results can achieved rather quickly. But, for more complex systems, with more intents and larger training corpora, NLTK seems the most correct choice, due to its greater robustness and versatility, in relation to the Watson Assistant service. Thus, once again, the choice of toolkit to use for the implementation of a system of this kind depends very much on what developer wants to accomplish and his preferences, since both have their advantages and disadvantages.

6.2 Final conclusions

Having implemented all the proposed tasks, and tested each of the systems developed, we now have a clearer idea of Watson's capabilities regarding these three tasks. In terms of performance, out of the three tasks implemented, Watson only performed better than NLTK on the final task, the natural language interface for database. On the other two tasks, NLTK outperformed Bluemix by a considerable margin, as it was shown in chapters 3 and 4. Although the results obtained with Bluemix are not poor, most of the time, they are worse than what can be achieved with NLTK. So, if performance was a determining factor, then NLTK would have the upper hand over Bluemix.

But, sometimes, this is not only factor taken into account when a developer chooses which toolkit to use. For all the three tasks implemented, we used services that did not need any training or implementation of complex models. For sentiment analysis and NER, we used the same already trained service, the NLU service. As for the natural language interface for database, there was no need to implement a semantic parser, we just had to provide training examples and the service trained on its own. This a very positive factor, since many users just want to perform small and simple experiments, and in those

[7] `https://www.kaggle.com/tmdb/tmdb-movie-metadata` (last visited on 02/05/2018)
[8] `https://www.kaggle.com/` (last visited on 02/05/2018)

cases, Bluemix has an advantage over NLTK. With all the pre-trained models and code simplicity, users can implement NLP tasks easily and quickly. The performance may not be as good as it is with a more complex model implemented with NLTK, but it is satisfactory nonetheless.

Another factor that can weigh in this decision is the execution time. If the systems have to process large amounts of data in a small period of time, then NLTK has an advantage over Bluemix. Since they are part of an online platform, Bluemix's services always take a longer time to present results than a system that is stored locally. Finally, in terms of help and information that can be found online to aid developers, NLTK has the advantage. It is a well-known NLP toolkit, used by many students and researchers, and it is open-source. This makes it very easy to find code examples and other types of useful information on the Internet. Bluemix, on the other hand, is an online platform, and users must pay to use it for more than one month, leading to far less usage by the community, and thus, less available information.

Having all these factors in mind, and although there are many more NLP tasks that were not tested, and other toolkits to compare with, for the context defined in this work, the overall advantage goes to NLTK. With it, users can achieve better results, they can develop their systems for free, and can find much more useful information online, which can aid and accelerate the development process. Although the simplicity that can be achieved with Bluemix is a very positive factor, it does not outweigh the others, in our opinion. But, as it was said before, this is a very small test sample. This may not be true other NLP tasks, or if other toolkits were used.

6.3 Future Work

Regarding future work, there are several possibilities that can be explored. As it was said previously, there are other NLP tasks that can be implemented being it with Bluemix or with NLTK. Thus, one possibility is to perform another comparative study, with different NLP tasks than the ones implemented in this work. This would give a bigger picture about Bluemix's performance in relation to NLTK's, and also about the other factors discussed throughout this work.

Another possibility to improve this comparative study, is to implement the same tasks but with another NLP toolkit. One of the alternatives considered in this work, before choosing NLTK, was SpaCy[9]. This would allow for an even broader comparison of Bluemix's services, having in mind not just the results achieved with NLTK, but also the results obtained with SpaCy. And, besides this toolkit, there are many others that could be chosen, such as Stanford CoreNLP[10], OpenNLP[11], among others. This would provide a greater depth to this comparative study, and would dissipate some doubts about whether the results achieved by other toolkits are consistently better than what is achieved with Bluemix, or if it just happened for NLTK.

In addition to all this, Bluemix is constantly updating its services, by discontinuing old ones and creating new ones, but also by changing already existing services. We do not know if these changes will be beneficial in terms of performance, or if the new services provide better results than the ones used in this work. As such, there is always the possibility of trying out new services, or check if the already used ones provide better results, as time passes.

[9] `https://spacy.io/` (last visited on 02/05/2018)
[10] `https://stanfordnlp.github.io/CoreNLP/` (last visited on 02/05/2018)
[11] `https://opennlp.apache.org/` (last visited on 02/05/2018)

Bibliography

[1] Rahul Bhagat, Anton Leuski, and Eduard Hovy. Statistical shallow semantic parsing despite little training data. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 186–187. Association for Computational Linguistics, 2005.

[2] Frédérik Bilhaut. The linguastream platfom. In *Proceedings of the 19th Spanish Society for Natural Language Processing Conference (SEPLN)*, pages 339–340, 2003.

[3] Frédérik Bilhaut and Antoine Widlöcher. Linguastream: an integrated environment for computational linguistics experimentation. In *Proceedings of the Eleventh Conference of the European Chapter of the Association for Computational Linguistics: Posters & Demonstrations*, pages 95–98. Association for Computational Linguistics, 2006.

[4] Steven Bird. Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*, pages 69–72. Association for Computational Linguistics, 2006.

[5] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[6] Jinho D Choi, Joel R Tetreault, and Amanda Stent. It depends: Dependency parser comparison using a web-based evaluation tool. In *ACL (1)*, pages 387–396, 2015.

[7] Cognitive Computation Group. Cogcompnlp - github main page. Internet: `https://github.com/CogComp/cogcomp-nlp` [last accessed: May. 3, 2017].

[8] Luísa Coheur. Lecture notes in natural language - chapter 1, 2016.

[9] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. Gate: an architecture for development of robust hlt applications. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 168–175. Association for Computational Linguistics, 2002.

[10] Hamish Cunningham, Yorick Wilks, and Robert J Gaizauskas. Gate: a general architecture for text engineering. In *Proceedings of the 16th conference on Computational linguistics-Volume 2*, pages 1057–1060. Association for Computational Linguistics, 1996.

[11] Laurent Debrauwer, Naouel Karam, and Jean-Pierre Brulé. Semantic searching and text analysis with distinguo®. 2008.

[12] DA Ferrucci. Introduction to this is watson. *IBM Journal of Research and Development*, 56(3):235–249, 2012.

[13] Mikel L. Forcada, Mireia Ginestí-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, and Francis M. Tyers.

Apertium: a free/open-source platform for rule-based machine translation. *Machine Translation*, 25(2):127–144, 2011.

[14] Ana Raquel de Oliveira Guimarães. Já Te Digo - Uma interface em língua natural para uma base de dados de cinema. Master's thesis, Instituto Superior Técnico, 2007.

[15] Matthew Honnibal, Mark Johnson, et al. An improved non-monotonic transition system for dependency parsing. In *EMNLP*, pages 1373–1378, 2015.

[16] Clayton J Hutto and Eric Gilbert. Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Eighth International AAAI Conference on Weblogs and Social Media*, 2014.

[17] Dan Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*. Dorling Kindersley Pvt, Ltd., 2014.

[18] Thomas K Landauer, Peter W Foltz, and Darrell Laham. An introduction to latent semantic analysis. *Discourse processes*, 25(2-3):259–284, 1998.

[19] Wei Li and Andrew McCallum. Pachinko allocation: Dag-structured mixture models of topic correlations. In *Proceedings of the 23rd international conference on Machine learning*, pages 577–584. ACM, 2006.

[20] Mykhailo Lobur, AnDriy Romanyuk, and Mariana Romanyshyn. Using nltk for educational and scientific purposes. In *CAD Systems in Microelectronics (CADSM), 2011 11th International Conference The Experience of Designing and Application of*, pages 426–428. IEEE, 2011.

[21] Edward Loper and Steven Bird. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective tools and methodologies for teaching natural language processing and computational linguistics-Volume 1*, pages 63–70. Association for Computational Linguistics, 2002.

[22] Nuno Mamede. Class notes in natural language - semantic roles, 2016.

[23] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60, 2014.

[24] Serguei A Mokhov. Introducing marf: a modular audio recognition framework and its applications for scientific and software engineering research. *Advances in Computer and Information Sciences and Engineering*, pages 473–478, 2008.

[25] Syed Shariyar Murtaza, Paris Lak, Ayse Bener, and Armen Pischdotchian. How to effectively train ibm watson: Classroom experience. In *System Sciences (HICSS), 2016 49th Hawaii International Conference on*, pages 1663–1670. IEEE, 2016.

[26] David Nadeau. *Semi-supervised named entity recognition: learning to recognize 100 entity types with little supervision*. PhD thesis, University of Ottawa, 2007.

[27] Lluís Padró, Miquel Collado, Samuel Reese, Marina Lloberes, and Irene Castellón. Freeling 2.1: Five years of open-source language processing tools. In *7th International Conference on Language Resources and Evaluation*, 2010.

[28] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[29] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[30] Push Singh, Thomas Lin, Erik Mueller, Grace Lim, Travell Perkins, and Wan Li Zhu. Open mind common sense: Knowledge acquisition from the general public. *On the move to meaningful internet systems 2002: CoopIS, DOA, and ODBASE*, pages 1223–1237, 2002.

[31] B Sujatha and S Viswanadha Raju. A generic model for natural language interface to database. In *Advanced Computing (IACC), 2016 IEEE 6th International Conference on*, pages 73–77. IEEE, 2016.

[32] Yee Whye Teh, Michael I Jordan, Matthew J Beal, and David M Blei. Sharing clusters among related groups: Hierarchical dirichlet processes. In *NIPS*, pages 1385–1392, 2004.

[33] The MARF Research and Development Group. Marf, the modular audio recognition framework, and its applications. Internet: `http://marf.sourceforge.net/` [last accessed: May. 2, 2017], 2007.

[34] Universitat Politecnica de Catalunya. Freeling - main page. Internet: `http://nlp.cs.upc.edu/freeling/` [last accessed: May. 1, 2017].

[35] Benjamin Waldron and Ann Copestake. A standoff annotation interface between delph-in components. In *Proceedings of the 5th Workshop on NLP and XML: Multi-Dimensional Markup in Natural Language Processing*, pages 97–100. Association for Computational Linguistics, 2006.

[36] Benjamin Waldron, Ann Copestake, Ulrich Schäfer, and Bernd Kiefer. Preprocessing and tokenisation standards in delph-in tools. In *Proceedings of the 5th International Conference on Language Resources and Evaluation*, pages 2263–2268, 2006.

A

NLP toolkits less related to this work

In this appendix, we provide a description of some Natural Language Processing (NLP) toolkits that were not taken into account in the process of deciding which ones to use to reach the goals proposed. These toolkits were not considered, because some are paid, and others only cover more specific topics, not related to the context of this work.

A.1 Apertium

Developed at Universitat d'Alacant, Apertium[1] is a free open source platform that can be used to perform Machine Translation (MT), that is, it translates written text from one language to another. This platform is particularly useful on cases in which the two languages in questions are similar, like Spanish and Portuguese for example. Although this platform was initially developed for this type of scenario, and its performance is better in these types of language pairs, it has also shown usefulness in more distinct languages. This platform uses hidden Markov models for PoS tagging purposes and finite state transducers to perform lexical transformations [13]. Since its latest update (May 2016), there are 43 pairs of languages that are considered stable and can be used to perform translation[2]. As it can be seen from the date of the latest update, this tool is still in use nowadays, with support for additional language pairs being developed.

A.2 Deeplearning4j

Much like the name suggests, DeepLearning4j[3] is an open-source library that includes implementations of several deep learning algorithms, such as neural networks, Boltzmann machines, recurrent and convolutional nets. Written in Java and Scala[4], this library can be deployed on either distributed or single-threaded environments, both on CPUs and GPUs. It also has APIs for development in other

[1] `https://www.apertium.org/` (last visited on 25/05/2017)
[2] `http://wiki.apertium.org/wiki/Main_Page` (last visited on 25/05/2017)
[3] `https://deeplearning4j.org/` (last visited on 25/05/2017)
[4] `https://www.scala-lang.org/` (last visited on 30/04/2017)

languages, such as Python and Clojure[5]. Developed by the San Francisco based software company Skymind[6], this library can be used to perform several tasks, such as image and face recognition, convertion of speech into text, fraud detection in finance and network intrusion detection. Since this is part of an open source initiative, the library's code is accessible publicly on GitHub[7].

A.3 Distinguo

Distinguo is a semantic search tool (it does not use WordNet) that allows its users to perform statistical and search operations based on the meaning of words or sentences, rather than on the number of occurrences of a word in a text. Developed by Ultralingua[8], Distinguo is available as a set of C++ libraries, which can be integrated into other software solutions. The languages supported by Distinguo are English and French. It is composed by two parts: **Distinguo Index** and **Distinguo Context**. Distinguo Index works with words, more precisely, their meaning (semantic). It breaks down the meaning of words and considers its synonyms, hyponyms, hyponyms and meronyms to broaden the search to documents or sentences with these new words. Distinguo Context is a contextual search tool that can recognize the grammatical roles of words in sentences, so it works with the meaning of sentences or full texts, rather than words. It can detect relationships between parts of sentences and texts and create matches between them, even if they do not use the same words [11]. Unlike the two previous tools already described in ths section, this one is of commercial use, users must pay to have access[9].

A.4 Machine Learning for LanguagE Toolkit (MALLET)

MALLET[10] is an open-source toolkit developed in Java, which can be used for statistical NLP, document classification, IE, clustering and other machine learning techniques applied to natural language. It has implementations several machine learning algorithms, such as Naïve Bayes, Decision Trees and Maximum Entropy, and code that can be used to evaluate the performance of classifiers with some of the most used metrics. MALLET also has tools useful for sequence tagging, which are implementations of several algorithms: Hidden Markov Models, Conditional Random Fields and Maximum Entropy Markov Models. There are also topic models, which can be used to analyse collections of unlabelled text. These models contain implementations some other algorithms, LDA, Hierarchical LDA and Pachinko Allocation. The tool can be used through the command line after installed, or other applications can use this toolkit, making calls to its Java API, described in detail on the tool's website.

A.5 Modular Audio Recognition Framework (MARF)

MARF[11] is an open-source platform that possesses a collection of text, speech, sound, voice and NLP algorithms, all of them written in Java and in a modular fashion, to make it easy to add new algorithms and modules [33]. As the name suggests, this framework was originally built for audio recognition purposes, but nowadays in not restricted to that field of NLP, as it has been stated before [24]. Presently, MARF focuses more on the pattern-recognition pipeline (loading, pre-processing, feature extraction, training and classification), including implementations of algorithms like Neural Networks, Linear Predictive Coding, distance classifiers, Fast Fourier Transformation (used in in feature extraction techniques),

[5] `https://clojure.org/` (last visited on 1/05/2017)
[6] `https://skymind.ai/` (last visited on 1/05/2017)
[7] `https://github.com/deeplearning4j/deeplearning4j` (last visited on 1/05/2017)
[8] `http://www.ultralingua.com/` (last visited on 28/05/2017)
[9] `https://en.wikipedia.org/wiki/Distinguo` (last visited on 25/05/2017)
[10] `http://mallet.cs.umass.edu/index.php` (last visited on 25/05/2017)
[11] `http://marf.sourceforge.net/` (last visited on 25/05/2017)

and grammar-based and top-down parsers [24]. This framework can be used in many ways, such as a Java API for other applications to make use of, and as tool for learning about the several algorithms implemented, or even extend them. The tool's source code is hosted at Sourceforge, and can be downloaded from here[12].

A.6 Rosoka NLP

Rosoka NLP[13] is a toolkit that can be used to identify different languages in text, to perform entity extraction, along with a set of other services that are offered. This tool has three major components, Rosoka Extraction, Rosoka Toolkit and Rosoka Analyst. Rosoka Extraction is an engine that allows users to perform entity and relationship extraction tasks, using Java API calls or using a REST web service. One of the features that stands out the most, is that this engine supports 230 different languages[14], which allows text processing in the most used languages in the world. The analysis performed can be outputted in different formats, like Extensible Mark-up Language (XML) or JSON. Rosoka Toolkit can be used to create new relationships, entity types, lexicons, character-based and semantic vector regular expression rules. Finally, Rosoka Analyst is the analytics module, which can be used to display the results of document analysis made by Rosoka Extraction. The results can be shown visually, and the user can choose from a range of visualizations, each focusing on different parameters. Also, these visualizations can be exported to PNG files, that can be used to show the results on a report or presentation. Much like Distinguo, this toolkit is not free of charge, user must pay to have access to the services offered.

A.7 Overview of the previous toolkits

Here, we present an overview of the toolkits presented in this appendix, considering the same criteria as in the table presented in Section 2.4.

Toolkits	**Functionality offered**	**Paid**	**Supported languages**	**Date of last update**
Apertium	Machine Translation	No	English, French, Spanish, Catalan, Portuguese, Italian, Dutch, Arabic, among others (43 language pairs in total)	April 2017
Deeplearning4j	Deep learning algorithms applied to face and image recognition, fraud detection, speech to text conversion and network intrusion detection	No	English	2017
Distinguo	Semantic search tool that analyses the meaning of sentences and texts	Yes	English and French	(Could not find that information)
MALLET	Statistical NLP, IE, document classification, machine learning techniques applied to NLP, like Naive Bayes, Maximum Entropy models and Decision Trees	No	English	(Not clear from website)
MARF	Audio recognition purposes and several ML algorithms, like neural networks, distance classifiers, top-down parsers	No	English	February 2006
Rosoka NLP	Language identification and entity extraction in texts. Also contains module for virtual display of the results of the experiments performed	Yes	230 different languages	(Not clear from website)

Table A.1: Comparison between the several NLP toolkits discussed in Appendix A

[12] `https://sourceforge.net/projects/marf/` (last visited on 2/05/2017)
[13] `https://www.rosoka.com/natural-language-processing` (last visited on 25/05/2017)
[14] `https://www.rosoka.com/content/faqs` (last visited on 3/05/2017)

B

Additional test results

In this appendix, we present the results of all tests that were executed, including results that were not discussed in previous chapters.

B.1 Sentiment Analysis

B.1.1 NLTK

Movie corpus

Test executed	Accuracy	Precision	Recall	F-measure
2-fold cross validation	0.7255	0.8032	0.7255	0.7064
3-fold cross validation	0.6767	0.7724	0.6766	0.6383
4-fold cross validation	0.6645	0.7631	0.6643	0.6297
5-fold cross validation	0.6365	0.7643	0.6364	0.5859
6-fold cross validation	0.7427	0.7747	0.7427	0.7335
7-fold cross validation	0.7088	0.7605	0.7093	0.6933
8-fold cross validation	0.6795	0.7663	0.6796	0.6512
9-fold cross validation	0.5891	0.7379	0.5913	0.5132
10-fold cross validation	0.6770	0.7469	0.6809	0.6531

Table B.1: Test results using the movie corpus with baseline features

Test executed	Accuracy	Precision	Recall	F-measure
2-fold cross validation	0.6985	0.7766	0.6966	0.6711
3-fold cross validation	0.6917	0.7802	0.6918	0.6647
4-fold cross validation	0.7215	0.7851	0.7217	0.7051
5-fold cross validation	0.5980	0.7440	0.6022	0.5182
6-fold cross validation	0.5616	0.7541	0.5611	0.4483
7-fold cross validation	0.6536	0.7581	0.6532	0.6112
8-fold cross validation	0.6530	0.7455	0.6529	0.6143
9-fold cross validation	0.7437	0.7550	0.7421	0.7342
10-fold cross validation	0.6500	0.7385	0.6494	0.6198

Table B.2: Test results using the movie corpus with stopword removing features

Test executed	Accuracy	Precision	Recall	F-measure
2-fold cross validation	0.7915	0.8123	0.7914	0.7878
3-fold cross validation	0.7843	0.8044	0.7840	0.7804
4-fold cross validation	0.7630	0.7995	0.7635	0.7555
5-fold cross validation	0.6720	0.7608	0.6721	0.6411
6-fold cross validation	0.7638	0.7839	0.7637	0.7594
7-fold cross validation	0.7539	0.7801	0.7494	0.7434
8-fold cross validation	0.7025	0.7671	0.7025	0.6820
9-fold cross validation	0.6707	0.7530	0.6701	0.6386
10-fold cross validation	0.7425	0.7514	0.7403	0.7357

Table B.3: Test results using the movie corpus with the most significant bigrams features

Test executed	Accuracy	Precision	Recall	F-measure
2-fold cross validation	0.8635	0.8667	0.8634	0.8632
3-fold cross validation	0.8604	0.8616	0.8598	0.8595
4-fold cross validation	0.8645	0.8661	0.8644	0.8642
5-fold cross validation	0.8310	0.8322	0.8304	0.8305
6-fold cross validation	0.8609	0.8657	0.8618	0.8602
7-fold cross validation	0.8501	0.8525	0.8500	0.8498
8-fold cross validation	0.8015	0.8162	0.8013	0.7987
9-fold cross validation	0.8223	0.8280	0.8221	0.8204
10-fold cross validation	0.8315	0.8317	0.8313	0.8306

Table B.4: Test results using the movie corpus with the most significant bigrams and most common features

Twitter corpus

Test executed	Accuracy	Precision	Recall	F-measure
2-fold cross validation	0.7831	0.7833	0.7832	0.7831
3-fold cross validation	0.7805	0.7805	0.7806	0.7805
4-fold cross validation	0.7668	0.7668	0.7669	0.7667
5-fold cross validation	0.7642	0.7674	0.7641	0.7634
6-fold cross validation	0.7604	0.7615	0.7605	0.7601
7-fold cross validation	0.7498	0.7551	0.7499	0.7485
8-fold cross validation	0.7453	0.7464	0.7454	0.7450
9-fold cross validation	0.7460	0.7468	0.7461	0.7458
10-fold cross validation	0.7477	0.7503	0.7471	0.7466

Table B.5: Test results using the Twitter corpus with baseline features

Test executed	Accuracy	Precision	Recall	F-measure
2-fold cross validation	0.7851	0.7852	0.7852	0.7851
3-fold cross validation	0.7768	0.7786	0.7774	0.7765
4-fold cross validation	0.7696	0.7701	0.7698	0.7696
5-fold cross validation	0.7617	0.7618	0.7617	0.7615
6-fold cross validation	0.7466	0.7514	0.7464	0.7451
7-fold cross validation	0.7549	0.7554	0.7550	0.7547
8-fold cross validation	0.7552	0.7560	0.7557	0.7549
9-fold cross validation	0.7405	0.7405	0.7405	0.7404
10-fold cross validation	0.7397	0.7408	0.7395	0.7391

Table B.6: Test results using the Twitter corpus with stopword removing features

Test executed	Accuracy	Precision	Recall	F-measure
2-fold cross validation	0.7823	0.7826	0.7822	0.7822
3-fold cross validation	0.7728	0.7729	0.7728	0.7728
4-fold cross validation	0.7717	0.7721	0.7719	0.7717
5-fold cross validation	0.7706	0.7709	0.7705	0.7705
6-fold cross validation	0.7678	0.7680	0.7678	0.7677
7-fold cross validation	0.7466	0.7467	0.7466	0.7465
8-fold cross validation	0.7532	0.7537	0.7532	0.7530
9-fold cross validation	0.7468	0.7469	0.7467	0.7466
10-fold cross validation	0.7528	0.7530	0.7529	0.7526

Table B.7: Test results using the Twitter corpus with the most significant bigrams features

Test executed	Accuracy	Precision	Recall	F-measure
2-fold cross validation	0.8148	0.8149	0.8149	0.8148
3-fold cross validation	0.8031	0.8033	0.8031	0.8030
4-fold cross validation	0.7947	0.7950	0.7947	0.7946
5-fold cross validation	0.7836	0.7840	0.7837	0.7835
6-fold cross validation	0.7778	0.7781	0.7781	0.7777
7-fold cross validation	0.7736	0.7739	0.7736	0.7735
8-fold cross validation	0.7689	0.7693	0.7688	0.7686
9-fold cross validation	0.7738	0.7744	0.7737	0.7733
10-fold cross validation	0.7646	0.7648	0.7649	0.7645

Table B.8: Test results using the Twitter corpus with the most significant bigrams and most common features

B.1.2 Bluemix

Twitter corpus

Number of tweets used for training	Accuracy	Precision	Recall	F-measure
5 tweets	0.5234	0.5121	0.9810	0.6729
10 tweets	0.5996	0.6026	0.5847	0.5935
20 tweets	0.5701	0.6955	0.2499	0.3677
50 tweets	0.5628	0.5549	0.6387	0.5938
100 tweets	0.6142	0.5769	0.8533	0.6884
200 tweets	0.6756	0.6641	0.7078	0.6852
300 tweets	0.7112	0.6870	0.7811	0.7310
400 tweets	0.7192	0.7038	0.7639	0.7327
500 tweets	0.7195	0.7116	0.7437	0.7273
750 tweets	0.7371	0.7152	0.7936	0.7524
1000 tweets	0.7479	0.7597	0.7340	0.7467
1500 tweets	0.7302	0.7267	0.7443	0.7354
2000 tweets	0.7385	0.7719	0.6913	0.7294

Table B.9: Test results using Bluemix's NLC service

B.2 NER

Test executed	Accuracy	Precision	Recall	F-measure
2-fold cross validation	0.7910	0.8661	0.8658	0.8659
3-fold cross validation	0.7951	0.8689	0.8667	0.8678
4-fold cross validation	0.7969	0.8702	0.8682	0.8692
5-fold cross validation	0.7964	0.8695	0.8687	0.8691
6-fold cross validation	0.7968	0.8699	0.8681	0.8690
7-fold cross validation	0.7962	0.8694	0.8681	0.8688
8-fold cross validation	0.7987	0.8711	0.8691	0.8701
9-fold cross validation	0.7986	0.8713	0.8671	0.8692
10-fold cross validation	0.7985	0.8709	0.8662	0.8695

Table B.10: Test results using the GMB corpus with the baseline features

Test executed	Accuracy	Precision	Recall	F-measure
2-fold cross validation	0.8118	0.8802	0.8738	0.8770
3-fold cross validation	0.8150	0.8822	0.8768	0.8795
4-fold cross validation	0.8166	0.8831	0.8778	0.0.8804
5-fold cross validation	0.8181	0.8845	0.8777	0.8811
6-fold cross validation	0.8191	0.8848	0.8786	0.8817
7-fold cross validation	0.8198	0.8855	0.8781	0.8818
8-fold cross validation	0.8201	0.8858	0.8794	0.8826
9-fold cross validation	0.8210	0.8862	0.8788	0.8825
10-fold cross validation	0.8206	0.8859	0.8788	0.8824

Table B.11: Test results using the GMB corpus with the previous and next word features

Test executed	**Accuracy**	**Precision**	**Recall**	**F-measure**
2-fold cross validation	0.8150	0.8820	0.8758	0.8789
3-fold cross validation	0.8193	0.8851	0.8792	0.8821
4-fold cross validation	0.8212	0.8864	0.8791	0.8828
5-fold cross validation	0.8224	0.8872	0.8798	0.8835
6-fold cross validation	0.8229	0.8872	0.8808	0.8840
7-fold cross validation	0.8234	0.8875	0.8803	0.8839
8-fold cross validation	0.8244	0.8881	0.8813	0.8847
9-fold cross validation	0.8236	0.8877	0.8808	0.8842
10-fold cross validation	0.8237	0.8880	0.8806	0.8843

Table B.12: Test results using the GMB corpus with the two previous and next word features

C

Named entity correspondences

In this appendix, we present the list of named entities that Bluemix's NLU service recognizes in the GMB corpus, along with their "translation", or equivalent, in the original GMB corpus.

NLU category	Corresponding GMB category
Award	art
Movie	art
SportingEvent	eve
Facility	geo
GeographicFeature	geo
Location	geo
NaturalEvent	nat
Broadcaster	org
Company	org
Organization	org
PrintMedia	org
Person	per
Crime	n/a
Drug	n/a
Hashtag	n/a
HealthCondition	n/a
JobTitle	n/a
MusicGroup	n/a
Quantity	n/a
Sport	n/a
TelevisionShow	n/a
TwitterHandle	n/a
Vehicle	n/a

Table C.1: Correspondence between NLU and the GMB corpus

D

Frame element types and respective values

In this appendix, we will provide an in depth list of all the frame element types created during the implementation of a natural language interface using NLTK. Additionally, we will cover the possible values the various frame element types can have. Starting with the frame elements that make up a question's intent, the first and most important frame element type is the `TARGET` type. It expresses the user's main intent, and can have several values:

- `actor_name` - used when the user asks a question regarding the actors of a specific movie (e.g. who played a certain role, which actors entered a certain movie);
- `budget` - used when the question is about a movie's budget;
- `character_name` - used when the user's question is about a movie's characters (a role played by a specific actor, or what characters are there in a movie, for instance);
- `genre` - used when the system is asked about the genres of a movie;
- `keywords` - used when the user asks a movie's keywords;
- `original_language` - used when the user asks a question regarding the most used language in a movie;
- `original_title` - used when the user wants to know the title (or titles) of movies that meet certain conditions (the movie with shortest run-time, movies with a certain actor, movies directed by a specific director);
- `overview` - used when the system is asked about the overview of a movie;
- `person_name` - used when the user wants to know who directed a movie;
- `production_company` - used when the question is about what companies produced a certain movie;
- `production_country` - used when the question wants to know in which countries was a movie produced;

- `release_date` - used when the system is asked about a movie's release date;
- `revenue` - used when the user wants to know the profit a movie has generated;
- `runtime` - used when the user asks a question regarding a movie's duration;
- `spoken_language` - used when the user's question is about the languages spoken in a movie (which languages are used, or how many are used);
- `vote_avg` - used when the question is about movie ratings (the rating of a movie, movie with best, or worst rating).

Next, there is the `AUX` frame element, which provides an extra degree of granularity to the main intent, providing additional information about what the user wants answered. It can have the following values:

- `count` - used in questions where the user wants to know a quantity (how many movies an actor has entered, or how many languages are spoken in a movie, for instance);
- `first` - used when the user only wants to know the first of a list of answers (first movie by an actor, first movie to be released, first movie by director);
- `full_date` - only used in questions where the `TARGET` frame element has the `release_date` value. This value indicates that the user wants a movie's full release date (day, month and year);
- `last` - used when the user only wants to know the last of a list of answers (latest movie by an actor or latest movie to be released, for instance);
- `least` - used when the user wants to know what entity has the smallest amount of a certain attribute (movie with smallest run-time, movie with least profit, for instance);
- `most` - same as the previous value, but for the biggest amount of an attribute (longest movie ever made, movie with best rating);
- `release_month` - only used in questions where the `TARGET` frame element has the `release_date` value. This value indicates that the user wants to know the month a certain movie was released;
- `release_year` - same as the previous value, but for when the user wants to know in what year was a movie released.

Another frame element type that is part of a sentence's intent is the `ACTION` frame element. This frame element is only used when, in the user's question, a person (actor or director) performs an action (direct a movie, perform a role), and it can have the following values:

- `act_with` - used in questions where the user wants to know what actors acted with a specific actor in a movie (who acted with Julia Roberts in Pretty Woman, for example);
- `directs` - used when the system is asked who was the director of a movie;
- `play_part` - used when the user wants to know what actor played a specific role in a movie (e.g. who played the role of Woody in Toy Story);
- `played_by` - used in questions where the user wants to the role played by an actor in a certain movie (what was the role played by Sam Worthington in Avatar, for instance).

And finally, there is the SORT_BY frame element type, the last of the frame types that make up the intent of a sentence. This frame element is used only when the frame also contains a AUX frame element, with the values least or most. As such, this frame element indicates which attribute should be used by the database to sort the rows. It can have the following values:

- revenue - used when the user wants to know what generated the biggest or smallest income;
- runtime - used when the user wants to know what generated the biggest or smallest duration;
- vote_avg - used when the system is asked what movie has the best, or worst rating.

These concludes the four frame element types that make a question's intent. The remaining frame elements are used to store the named entities mentioned in the user's question. In our case, there are four named entity types that should be recognized: actor names, director names, character names and movie titles. As such, a new frame element type was created for each of these types of named entities. The resulting frame element types are: ACTOR, DIRECTOR, CHARACTER, MOVIE. Unlike the frame elements that make up the intent, these can have a wide array of different values (all the names of the entities they represent).

Glossary

PoS tagging Consists of assigning, in an automatic fashion, a morpho-syntactic tag, or, as it is most commonly called, a **part of speech**, to words in a text [17]. A part of speech refers to specific category of words. These categories can be nouns, adjectives, adverbs, determiners, or prepositions, for instance. Like in other NLP tasks, this is not an easy task. Due to the ambiguity present in most texts, words can be assigned (mistakenly) multiple parts of speech, or categories, which is impossible to happen . 7, 9–15, 70

Hierarchical Dirichlet Process It is a clustering algorithm that takes as input groups of data, and organizes each group into clusters, with elements of the same cluster having similar properties [32]. 13, 15

hyponym A word is a hypernym of another word when it has a broader meaning than the other word. For instance, *color* has a broader meaning than *red*. So, it is said that *color* is a hypernym of *red*. 71

hyponym A word is a hyponym of another word when it has a more specific meaning than the other word. For instance, *bird* refers to a more strict group of beings than *animal*. So, it is said that *bird* is a hyponym of *animal*. 71

LDA Short for Latent Dirichlet Allocation, it consists in assigning to each a word in a document, a set of probabilities that indicate the likelihood of that word being related to a certain topic [5]. 13, 15, 71, 85

LSA Standing for Latent Semantic Analysis, this algorithm, for each word in a text, tries to discover all contexts in which it appears in that text. The result is a set of constraints that determines how similar words are [18]. 13, 15

meronym A word is a meronym of another word when it refers to a part of the entity that is reffered to by the other word. For instance, *engine* is a part of a *car*. So, it is said that *engine* is a meronym of *car*. 71

NER Short for Named Entity Recognition, it is the task of assigning to named entities in a text, a set of pre-defined categories. A named entity consists of anything that can be referred to using a proper noun [17]. Some of the most common categories are: person, location, monetary value or organization, among others. Although this may seem a trivial task, most of the time it is not easy to perform this automatically. For instance, company names and proper nouns can be difficult to identify, due to the multiple nouns present in each. 1, 7, 9–11, 14–16, 29

Pachinko Allocation it is an improvement over LDA. In this model, besides the correlations between words and topics, correlations between different topics are also computed [19]. 71

parsing Consists of analysing the syntactic structure of sentences (hence why this is also called syntactic parsing), and producing the corresponding parse tree [17]. There are many strategies to perform this task, being the most common the bottom-up and top-down strategies . 7, 9–12, 14, 15

semantic role Captures the semantic similarities between different entities referred in sentences [17]. Each entity is given a role that can be very specific, or very general. Although there is no well defined list of semantic roles, some of them are used commonly by the NLP community. Here are some of them [22]:

- **Agent** - normally represents a human noun (can also represent animals), which refers to a person who willingly performs an action;
- **Experiencer** - entity that represents animate objects that go through some psychological change or process;
- **Beneficiary** - entity that represents the person for whom a certain action or event is performed;
- **Theme** - refers to something that is being acted upon or going through change;
- **Instrument** - refers to a material or tool used to perform an action;
- **Force** - used when an event is caused by something that humans cannot control (the weather, for instance). Refers to the causer of this type of event.

. 4, 11, 12, 15

stemming Stemming, or lemmatization, is the process of transforming words, so as to words with the same meaning are represented by the same token [17], called **lemma**. An example of this would be with the words *runs* and *running*. The result of stemming these two words would be same, the word *run*. Stemming is very useful in, for instance, a search engine. If the user performs different searches using words that have the same lemma, most of the time, the results obtained will be similar . 7, 9, 11, 14, 15

tokenization Consists of identifying tokens (words) in a text [17]. This may seem like an easy task, but many problems may arise. For instance, in cases like *New York*, or *facebook.com*, it is not trivial to perform this task. In the first case, there is a space separating *New* and *York*, but normally, it is desirable to consider the term *New York* a single word, which represents the city in question. In the second case, the term contains a dot, which can be mistaken as the ending of a sentence, but in fact, it is part of a single word. So, to perform this task correctly, all of these more complex cases must be taken into account. 7, 9–11, 14, 15

word2vec An algorithm based in neural networks that transforms each word in a text into a vector. The result is a matrix that stores all words in a document [29]. 13, 15