Utrecht University

Thesis submitted for the degree

Master of Science

in

Computing Science

Faculty of Science

Dept. of Information and Computing Sciences

ProbQL: A Probabilistic Query Language for Information Extraction from PDF Reports and Natural Language Written Texts

by

Ing. Daniele Di Grandi

September 2022

Author:

Ing. Daniele Di Grandi

Supervisors:

Prof. dr. Yannis Velegrakis

Dr. Rick Vreman

Acknowledgements

Evitando di essere troppo prolisso (già lo sarò nella tesi), vorrei cogliere l'occasione per scrivere due parole per ringraziare giusto un paio di persone, cercando - con tutto me stesso - di evitare di scrivere una di quelle cose imbarazzanti che si leggono di solito all'inizio delle tesi.

Vorrei ringraziare Yannis Velegrakis, principale relatore di questa tesi, per il supporto e i preziosi consigli datimi lungo il percorso. In secondo luogo, ringrazio Rick Vreman e Jan-Willem Versteeg, del dipartimento di farmacia dell'Università di Utrecht, per i dati fornitimi, rivelatisi necessari per il corretto svolgimento delle analisi.

Passando ora ai ringraziamenti più di□cili da scrivere senza cadere nel banale, un ringraziamento va alla mia famiglia al completo (così fortunata ad avere un ragazzo così intelligente, bello, simpatico e soprattuto umile come me) per il supporto e l'essersi fidati di me durante questi anni universitari. Un ringraziamento un pò più speciale al nonno Peppino, senza i cui sacrifici non avrei mai avuto la notevole stabilità che mi ha permesso di vivere l'Università al meglio, a cui sono infinitamente grato.

Tornando più leggeri, un ringraziamento a me stesso per averci creduto fino in fondo, fin dall'inizio di questa avventura (bravo Denny). Se in futuro dovessi leggere queste parole, fermati a ricordare quanto impegnativa ma soddisfacente sia stata questa esperienza.

Infine (tranquilla non mi sono scordato) un grazie ad Alice, che mi ha accompagnato in questa avventura, rendendola meno impegnativa e ancor più soddisfacente: tu sai già tutto.

Affirmation

I hereby a□rm that this Master thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text. This work has not been submitted for any other degree or professional qualification except as specified; nor has it been published.

Utrecht (NL), September 2022

Ing. Di Grandi Daniele

Abstract

In recent years, Information Extraction (IE) has become an increasingly important field due to the vast amount of data being produced at an ever-increasing rate. However, it was estimated that about 80-90% of data produced by companies are unstructured - such as PDF documents and natural language written texts - meaning that if not refined, valuable information and interesting patterns can remain hidden. The problem with unstructured data is that the information extraction process is a difficult task, due to their intrinsic uncertainty nature.

This Thesis proposes a novel programming language - ProbQL - that is a rule-based query language through which it is possible to write queries able to extract information from a PDF document or a text file. The main idea of this language is to split the given document (or text) into a list of items, and then score each item with a probability value of being the correct data to extract, based on some rules defined by the final user, describing some properties that the desired data is thought to have. The language has been tested on the task of extracting 5 types of data from PDF documents - medicine reimbursement reports - obtaining an average extraction accuracy of 86% (with a minimum of 78.3% and a maximum of 99.1%). To the best of our knowledge, ProbQL is the first language for extracting information from PDF documents and texts that is probability based, which allows to perfectly deal with the uncertainty of a desired piece of data being located in different positions based on the type of considered PDF document.

Contents

List of Abbreviations

AD - Assessment Date
AQL - Annotation Query Language
AUC - Area Under the Curve
BERT - Bidirectional Encoder Representations from Transformers
DT - Decision Trees
HMM - Hidden Markov Model
HTA - Health Technology Assessment
I - Indication
IDE - Integrated Development Environment
IE - Information Extraction
KB - Knowledge Base
LSTM - Long Short-Term Memory
MBN - Medicine Brand Name
MGN - Medicine Generic Name
ML - Machine Learning
NLIDB - Natural Language Interface for Databases
NLP - Natural Language Processing
NP - Noun Phrase
OIE - Open Information Extraction
PDF - Portable Document Format
POS - Part-of-Speech
ProbQL - Probabilistic Query Language
RC - Reimbursement Class
RD - Read and Delete
RE - Regular Expression
RNN - Recurrent Neural Network
RO - Read Only
SQL - Structured Query Language
SVM - Support Vector Machines
VP - Verb Phrase

List of Figures

List of Tables

1 Introduction

Discovering knowledge and extracting information from data has always been an extremely important process ever since the first Databases appeared. The advantages that these techniques entail are not purely about economics and business, but can also embrace other aspects: knowledge is a complicated process, and trying to extrapolate it from existing data that, as human beings and as a result of the evolution in technology, we produce every second, can be a valid method to increase our culture and discover concepts and patterns otherwise inexplicable. Exactly for this reason, Information Extraction (IE) has become an increasingly important field that has been growing very fast in the last few years.

1.1 Information Extraction

The ultimate goal of IE is to extract valuable pieces of information from different data sources, often semi-structured or unstructured (such as text written in natural language), and organize the extracted information into a structured database. As of today, about 80-90% of data produced by companies are unstructured [1], and this means that if not analyzed, valuable information and interesting patterns can remain hidden in these data, penalizing and compromising the decision-making process and innovation in general. Hence, an IE system is only an intermediate task included in broader applications, also having diametrically opposed domains, but similar in the sense that they are trying to optimize some sort of decision-making process.

Nonetheless, IE is an extremely difficult task, that involves complicated techniques in order to cope with the different domains in which it is utilized. As an example, if information has to be extracted from documents written in natural language - such as Portable Document Format (PDF) documents or similar - a very common method is to define some sort of rules based on the layout and format of the document itself, as further described in Chapter 2. However, if a slightly modified version of the same document is provided, the same rule-based IE application would fail at extracting the relevant data. More generally, the re-usability in a different domain of an IE system that was designed to work in another domain is a major limitation of the IE field that researchers are trying to overcome. In this sense, an interesting direction of research is the definition of a standard programming language that can be used as a tool to extract information from unstructured data sources, in a similar manner that SQL (Structured Query Language) is used to extract information from structured sources [2].

This Thesis presents a rule-based IE application that involves the definition of a simple programming language to automatically extract different relevant data from PDF files written in natural language. Then, this language will be tested to extract data from health technology assessment re-

ports that assess if new health technologies (e.g. medicine) should be reimbursed. Please note that the extraction of these data from the field of medicine reimbursement is only a check that was used to test the IE application, but its underlying principles, the defined simple programming language and the general method involving the definition of rules through this programming language can both be used to extract relevant data from PDF documents of other domains.

1.2 Research Challenges

The main goal is to develop an application where it is possible to write in a new simple programming language a query that represents the rules that have to be used to extract specific metadata from a text written in natural language. Hence, given as input a PDF document and a configuration file where the query containing the rules to extract the desired metadata is specified, the application is able to execute the configuration file and by following the described rules can extract the desired metadata from the PDF file.

The simple programming language that will be developed to specify the rules should provide enough constructs in order to be able to deal with the rule definition task, to be able to correctly identify the part of the text to extract. As an oversimplified example, if one provides a rule that says "On page 3, line 12, after the word DOG and before the word CAT there is the desired text to extract", already 4 constructs have been introduced in the language: 1 - a construct to go on the desired page, 2 - a construct to go on the desired line in a text, 3 - a construct able to parse words after a specified word and 4 - a construct that recognizes to stop extracting text when a specified word is encountered.

Hence, fundamentally understanding the different tasks that a rule can specify allows a better understanding of the constructs that the language should provide. This leads to the following main research challenge:

Research challenge: Given a metadata extraction task, design a programming language in which it is possible to express queries to extract all the necessary information from a text.

The following sub-challenges will be tackled to solve the main challenge:

1. Convert a PDF file into plain text (`txt`) file.

2. Identify which constructs are needed for a language in order to be expressive enough to write queries for a metadata extraction process.

3. Define a complete grammar of the language such that all the identified constructs can be implemented.

4. Implement these constructs in a program that given a text file and a configuration file containing the query written in the developed language, can execute the query and extract the metadata from the given text.

1.3 Outline

The rest of the Thesis is organized as the following. Chapter 2 will present and discuss the related state-of-the-art approaches and the research that has been conducted in the IE field. Then, Chapter 3 will provide a more detailed problem statement, that contains more information about the Pharmaceutical Department problem to tackle. In Chapter 4, the full solution will be extensively discussed, Chapter 5 will present the experiments performed and the queries written in the new language to solve the Pharmaceutical Department problem and Chapter 6 will address the limitations and possible future directions. Finally, Chapter 7 will conclude the Thesis.

2 Related Work

Many research works have been conducted about information extraction, where di□erent techniques and approaches were investigated and implemented. To maintain a coherent, clear and readable structure, the discussed approaches are clustered based on the treated topic, as visible in Figure 1.

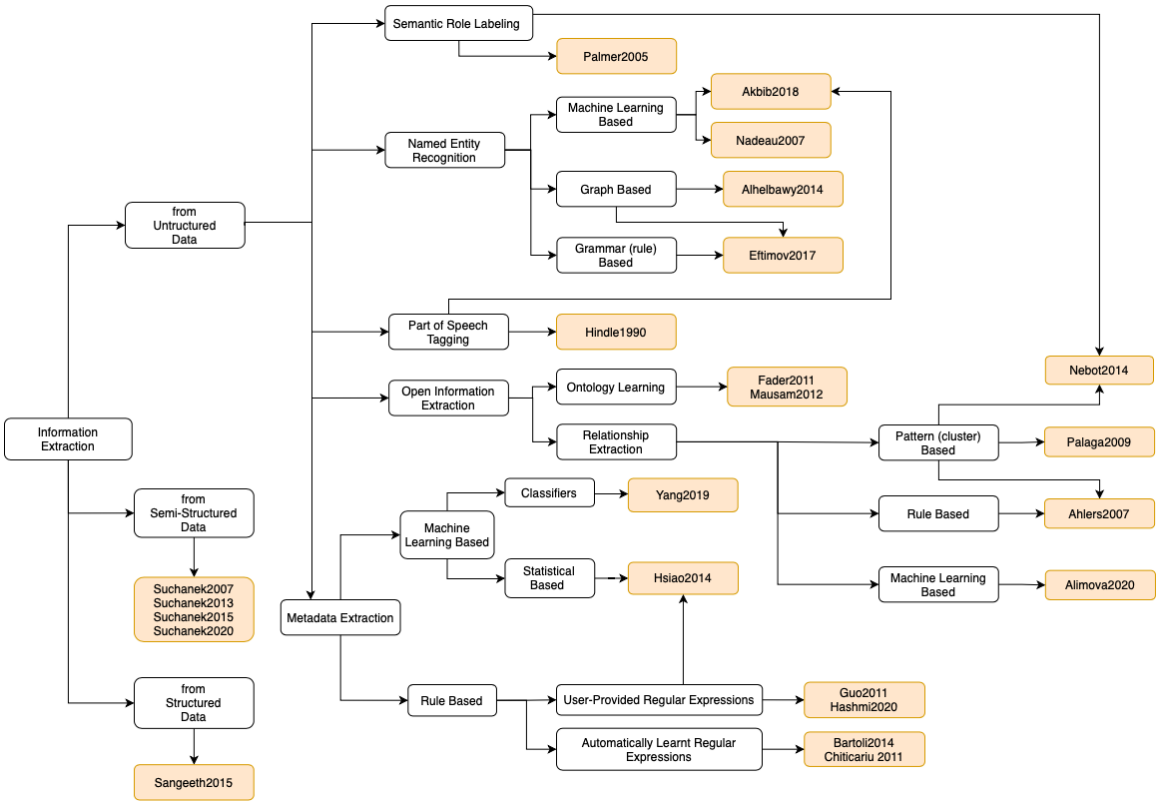

Figure 1: *Information extraction approaches.*

Starting from the general topic of information extraction, a first subdivision can be identified obtaining three branches: Structured-Data, Semi-Structured Data and Unstructured Data.

2.1 IE from Structured Data

In the literature, the term *information extraction* usually refers to the process of extracting information from semi-structured or unstructured data, but not from structured data. This is because extracting information from structured data (i.e. relational databases) is a problem that has well-

known and standardized solutions, among which the most popular is to extract information from a relational database using SQL - Structured Query Language - placing itself as the standard technology to be used. Indeed, it is common sense that extracting information from structured, thus organized, sources it is easier and less challenging than extracting information from unorganized and often chaotic sources, such as plain texts written in natural language.

However, while it is true that information stored in relational databases can be accessed only by executing SQL queries, this implies two main requirements that not always can be met: database users have to be able to write SQL queries, as well as they should know the exact underlying database structure, with columns and tables names.

Hence, to extract information from a relational database, a user that fails to meet one of the aforementioned requirements may need a Natural Language Interface for Databases (NLIDB) system that allows accessing the database by typing requests in natural language (English). The processes a NLIDB [3] is comprised are as follows. The natural language query is tokenized (i.e. split word-by-word based on the space character) and tagged mostly using the *PENN Tree bank POS tag set* [4], in order to classify each word into its lexical category (i.e. noun, verb, adjective, adverb, etc.). This process is known as Part-of-Speech (POS) tagging and it will be further described in Section 2.3.1. Then, the input is chunked, in order to classify it into its noun phrase (NP) and verb phrase (VP).

All the processes described thus far, are comprised in the so-called linguistic module, which can be implemented using a Hidden Markov Model (HMM). Finally, the chunked tag of the user's request is translated to an SQL query by the database module, which analyzes the chunked input to search for table names, column names or their synonyms. Moreover, logical connectors like AND, OR and NOT can be searched, as well as conditions like "greater than" and "less than", by looking for the tags CC, JJR and JJS given by the POS tagger, which respectively correspond to "Coordinating conjunction", "Adjective, comparative" and "Adjective, superlative".

2.2 IE from Semi-Structured Data

The problem of extracting information from semi-structured data sources, such as Wikipedia infoboxes or websites, constitutes a first initial challenge in the IE domain. The most important research effort that has been done is related to the so-called ontology learning and definition. An *ontology* is a representation that shows the characteristics of a domain area (under the form of categories, properties, concepts, entities and relations) in order to be used as a knowledge base (KB) by applications that aim at solving problems within the realm of the world for which the ontology has been defined. While it is possible to manually define an ontology for a specific domain, this process can be very tedious and time-consuming, not to mention that it can be very error-prone, considering that modern ontologies contain millions of entities and entries. Hence, an automatic way of learning and defining ontologies is needed. In this sense, the most popular effort in automatically constructing an ontology from semi-structured data is the YAGO knowledge base, considering all of its versions produced over time.

YAGO contains knowledge in the form of entities, facts, relations between facts (and entities) and properties of relations. The architecture of YAGO is based on declarative rules in the form of subject-predicate-object triples, and most of them are based on text-pattern matching through the

use of regular expressions, which are going to be further discussed in Section 2.3.5.

The main characteristic that differentiates YAGO from other ontologies is that YAGO was built from different sources, in contrast with using a single source of background knowledge [5]. Moreover, YAGO is decidable, and not only it can be easily extended by facts extracted from other sources, but the more facts YAGO contains and the easier it becomes to extend it further, in a positive feedback loop.

Indeed, most of the knowledge encoded in YAGO has been extracted using as sources the Wikipedia category pages and a semantic lexicon for the English language called WordNet [6]. While Wikipedia category pages are organized in a hierarchy, this is too simple and not useful in order to build an ontology. On the other hand, WordNet provides a carefully assembled hierarchy of thousands of concepts, but sadly, the Wikipedia concepts cannot be easily matched to the WordNet ones. However, by precisely designing its structure (e.g. every object is considered to be an entity, and between two entities there could be a relation, etc.), YAGO was able to link the two sources with near-perfect accuracy, which for this task is estimated to be around 97%. The remaining part of the encoded knowledge has been extracted from three different sources: GeoNames [7] to include knowledge about spatio-temporal references [8], Wikidata [9] to include knowledge from Wikipedia pages of 10 other languages other than English [10], and schema.org [11] to substitute the complex taxonomy of Wikidata with a simpler and human-readable taxonomy [12].

More precisely, including spatio-temporal references allows for example to ask an ontology for the distance between two events that take place in two different locations, rather than asking the distance of those locations themselves. YAGO represents this type of knowledge by including information about time (in the format of a date) and location (a coordinate-pair of latitude and longitude). However, while for the temporal dimension it was possible to extract information only from the Wikipedia infoboxes, this was not the case for the spatial dimension, thus, YAGO matches its entities with the entities contained in the geographical database GeoNames to extract spatial facts.

Regarding the handling of knowledge contained in Wikipedia pages of 10 other languages, this was a necessary feature because only using the English version of Wikipedia (which at the time contained about 4.5 million articles) can be restricting, considering that Wikipedias of other languages have more than a million articles. However, some of them can be overlapping, thus creating duplicate entities, because the same article can be described in different languages. YAGO uses an approach that maps multilingual infobox attributes to canonical relations and to overcome the overlapping article's problem, it merges equivalent entities into canonical entities with the help of Wikidata, which maintains its own repository of entity and category identifiers and maps them to articles in Wikipedias in different languages. Wikidata is a knowledge base that shares the same goal as the YAGO project, however, it is built as a community effort, where the members manually add facts to the knowledge base, while the approach of YAGO is automated, which means that if merged, the two approaches can benefit from each other. However, the collected facts about instances of Wikidata are forced into a rigorous type hierarchy with semantic constraints, resulting in the complex taxonomy of Wikidata being substituted with the simpler and human-readable taxonomy of schema.org. The obtained result is a knowledge base that not only is large, but also logically consistent, allowing OWL-based semantic reasoning.

Finally, YAGO achieves a 95% average accuracy between extracted relations, and contains information for more than 64 million entities and more than 2 billion facts about these entities.

2.3 IE from Unstructured Data

Unstructured data - such as text written in natural language - represent the ultimate challenge in extracting and organizing information in an automatic manner, and most of the research in IE focused on this type of extraction, providing a variety of different techniques and approaches. According to [13], in general, rule-based systems proved to be better extractors than other methods, although this highly depends on the used evaluation criteria and the application domain. Other than rule-based systems, several solutions are based on some type of Machine Learning (ML) techniques.

While rule-based systems work through the definition of some types of rules to identify the correct information to extract (eg. a simple rule can be "on page 3 of the document the first sentence at line 4 is the data to extract" for a Metadata Extraction task, or even "capitalized words are candidate named entities" for a Named-Entity Recognition task), ML based systems work by being trained on large (usually) labeled corpus, containing examples of data to be extracted. Then, through the definition of some features, and an implementation of a ML algorithm (eg. Support Vector Machines, Random Forests, etc.) that tries to learn how the defined features can point towards a certain label, they are able to recognize the correct data to extract, and when a new unseen example is provided to the ML system, it will be able to extract the data with a certain accuracy.

Generally, each method has its own benefits and drawbacks [2]. For rule-based systems, they are easy to comprehend and maintain, although the maintenance and development could require tedious manual labor. Also, rule-based systems still have to improve their flexibility to be used across different document layouts. However, it is easy to incorporate domain knowledge in the defined rules, although sometimes the rules are only heuristics and can produce wrong results, but tracing back the causes of errors is a simple task. On the other hand, ML based systems can reduce manual effort because they can be automatically trained and easily adapted to perform different tasks only by being trained on different documents. However, the training often requires labeled data - usually not directly available in real-world scenarios - or a large corpus at best, and the re-training for domain adaptation can be a time-consuming process. Finally, errors are not easy to trace due to the opaqueness nature that characterizes ML based approaches. By also looking at the business world, the output of an extraction system is often the input to a larger process, and it is the quality of the larger process that drives business value. In order to define some quality metrics, an important aspect to consider is what happens when an application has to be modified. While ML based systems have to be completely retrained, for rule-based systems is often enough to add/remove a couple of rules. Hence, the commercial world greatly values rule-based systems for their interpretability, understanding, debugging, and maintenance in the face of changing requirements. Considering these judgments, it is clear that rule-based systems are still a flourishing approach used in the real-world, as well as researched in the scientific literature, where a part of the research effort is now focusing on trying to develop a standardized language - such as SQL for structured data - to extract information on unstructured data. Considering this, the simple language that we will define aims to be general enough to extract information from a text source. Moreover, as previously mentioned, this language will be tested by extracting specific information (metadata) from medicine reimbursement reports.

More generally, IE on unstructured data proved to have common issues across all techniques. The first and major limitation is that there is no generalized mechanism for all the domains: the described approaches are highly customized over a specific domain, which in some cases is considered to be the

document layout. In fact, even a slightly modified version of a document can cause serious drops in the quality of the extracted information, either because rules fail at extracting data or because a ML system was trained on different layouts. Hence, the type of document is a critical factor, especially when the document is an old scanned PDF, where a further layer of complexity is required because of the optical character recognition techniques that have to be applied to obtain a text version of the document. Second, there are no unified goals: depending on the domain of application and the document analyzed, one may look for different pieces of information, that can range from just specific keywords to even semantic relations between entities, with the consequence of increasing the overall complexity by having very specifically designed techniques for each domain. Third, since the documents to be analyzed are written in natural language, information extraction techniques have to deal with all the problems that concern Natural Language Processing (NLP) techniques. Just to mention a few, natural languages are inherently ambiguous, colloquialisms and slang are very difficult to process, and most of the systems in place are tailored on the English language, which of course may be limiting since they cut off a large portion of non-English documents potentially containing useful information, as already discussed for YAGO [10].

The relevant literature about extracting information from unstructured data can be divided based on the aimed goal into five main approaches: Part-of-Speech Tagging, Named-Entity Recognition, Semantic Role Labeling, Open Information Extraction and Metadata Extraction.

2.3.1 Part-of-Speech Tagging

Part-of-Speech (POS) tagging - or grammatical tagging - is the task of labeling each word in a given text with a part-of-speech, based on syntactic or semantic analyses, in order to consider the context of the text under analysis. In this field, multiple English parts of speech are used, where the main ones are: noun, verb, article, adjective, preposition, pronoun, adverb, conjunction, and interjection. However, from these nine main categories, different sub-categories have been individuated, such as singular or plural for the noun category. Usually, POS applications are not useful by themselves, but rather are incorporated into larger pipelines to simplify some tasks or to achieve a relevant result.

As an example, an early approach to the usage of a POS tagger [14] classifies English words according to the predicate-argument structures they show in a large corpus of text, with the objective of retrieving semantically similar nouns and clustering them together. The idea is that each noun may be characterized according to the verbs that it occurs with, then, nouns can be grouped based on the frequency that they have in similar environments. By using the co-occurrence metric - an estimate of the mutual information metric - between a noun n and a verb v, it is possible to give a score based on the frequency that n occurs as object of v. If this co-occurrence score is computed for each noun and verb pair in the corpus, it is possible to compute the object, subject and overall similarity between any two nouns and thus obtaining different groups of semantically similar nouns (e.g. a boat is semantically similar to a ship).

Modern POS taggers are usually based on ML techniques [15] where the text is treated as a sequence of words to be labeled with linguistic tags, trying to learn the next character based on the previous characters. However, these systems are often implemented together with a Named-Entity Recognition tool, hence, a detailed presentation will be given in Section 2.3.2, when the Named-Entity Recognition task has been presented.

2.3.2 Named-Entity Recognition

Named-Entity Recognition (NER) is the task of automatically recognizing and classifying named entities mentioned in an unstructured data source into predefined categories such as person names, locations, organizations etc.

Each of these categories can in turn be divided into multiple sub-types of fine-grained data. For example, the type “location” can either be a city, a state, or a country.

However, NER tasks have to deal with NLP and when a NLP task is involved, due to its intrinsic nature, two decisions are automatically taken: the language and the domain [16]. These external factors play a crucial role in the portability of the system itself in other domains and other languages. For this reason, the NER task is well studied not only in English, but also in other languages, predominantly, German, Spanish, Dutch, Japanese, Chinese, French, Greek and Italian. However, although any domain can potentially be supported, when the same NER system was tested on a different domain than it was designed for, a serious drop in performance was reported, ranging from 20% to 40% for precision and recall [17].

The literature on NER can be divided into three main approaches: Machine Learning based, Graph based and Grammar (rule) based.

Machine Learning based. These types of techniques involve the usage of a ML model in order to perform the NER task, and are considered to be the techniques of choice for recent state-of-the-art systems.

Indeed, a recent system [15] aimed at improving the current state-of-the-art regarding both the NER and POS tagging tasks. The main idea is to treat text as a sequence of words to be labeled with linguistic tags, with the objective to model language as a distribution over characters. The underlying concept at the basis of this model is to learn to predict the next character on the basis of previous characters, aiming at internalizing concepts such as words, sentences or even sentiment, seemingly being able to understand the semantic features of a text. The system is using the Long Short-Term Memory (LSTM) variant [18] of a Recurrent Neural Network (RNN) [19], which is a particular type of neural network that is able to deal with temporal sequences, as one may consider a text to be. As previously mentioned, the RNN uses characters as atomic units of language modeling, in order to be able to learn how to predict the next character. Using this procedure and considering the fact that the RNN is trained on unlabeled corpora, the network is trained without any explicit notion of words, achieving the goal of fundamentally modeling words as sequences of characters. Furthermore, hidden states of a forward-backward RNN are utilized to contextualize words in the text, by producing embeddings that are computed not only on the characters of a word, but also on the characters of the surrounding text, meaning that the same word will have different embeddings depending on its contextual use.

For the NER task, this system achieved for the English language an F1 score of 93.09%, classifying itself as a new state-of-the-art, increasing this score by + 0.87% if compared to the old state-of-the-art system [20]. Moreover, it also raised the F1 score for the German language, achieving a result of 88.33%, an improvement of + 9.56% compared to the old state-of-the-art system [21]. Finally, a minor improvement was also achieved for the POS tagging task, although this improvement is not comparable to the one achieved for the NER task.

Before this system was introduced in the literature, it was noted [16] that old state-of-the-art

ML systems relied on techniques such as Hidden Markov Models (HMM) [22], Decision Trees (DT) [23], Support Vector Machines (SVM) [24] as well as Bootstrapping [25].

While HMM, DT and SVM all require the reading of a large annotated corpus - for training reasons - in order to be able to learn lists of entities and create disambiguation rules based on discriminative features, bootstrapping works differently. Indeed, bootstrapping doesn't require a large annotated corpus - often difficult and costly to create - to start the learning procedure, but only a set of seeds that constitutes some annotated examples of the searched named entity. Then, it is possible to identify in the given input text some contextual clues that are common to the given examples through the definition of some rules (eg. if the spelling contains "Mr." then it is a Person), in order to find new instances of the searched entity and new contexts, that can all be included in the examples set. Iterating this procedure leads to the discovery of several instances of the searched named entity, making it possible to generate for example one million facts with a precision of about 88% starting from a seed of 10 examples of facts [26].

When implementing a NER system using ML techniques (especially HMM, DT and SVM), designing and using the correct features is a crucial aspect as important as the choice of which ML algorithm to use [16]. A feature is a quantifiable and measurable property of a phenomenon. The important concept of a well-designed feature is that it should take similar values for phenomenon perceived as being similar, and should take different values for phenomenon perceived as being different. Usually, in the NER task, the features are tailored over the words, hence, for each word in the corpus, different feature values are computed and aggregated in the so-called feature vector. In a sense, the resulting vector containing the whole features for a single word can be seen as an abstract representation of that word: each feature catches different aspects and, if the combination of all the features has been properly designed, then the "important" parts of the word will be represented either as boolean, numerical or nominal values, which is indeed, an abstract representation. For example, a NER system may represent each word of a text with 3 attributes: 1 - a boolean value True or False based if the word is capitalized, 2 - a numerical value corresponding to the word length in characters, and 3 - a nominal value corresponding to the lowercase version of the word. Possible usage of these feature vectors is to apply a rule system over them. For example, candidates named entities can be individuated based on the rule "capitalized words are candidate named entities" and these entities may be classified based on the rule "the type of candidate named entities of length greater than 3 words is *organization*".

Graph based. Graph based techniques use elements of graph theory in order to perform the NER task. During this task, an important step is the disambiguation between named entities. In fact, there could be a many-to-many relation between named entities mentioned in a text and the corresponding entities in the real world. For example, when "Norfolk" is mentioned in a text, it can either refers to a person ("Peter Norfolk", a wheelchair tennis player) or a Canadian city ("Norfolk County"). Hence, establishing a correct mapping between each named entity mentioned in a document and the entity it represents in the real world is a crucial step. A possible solution is a graph-based solution [27]. By modeling each possible candidate for every named entity mention in the document as a node in a graph, and by saying that two nodes are connected by an edge if there is coherence between the candidates represented by the considered nodes (i.e. two candidates are considered to be coherent if real-world relations hold between them, and Wikipedia documents of the

two candidates entities are used to establish this relation), the problem of finding a set of nodes in which only one node between a subset of connected nodes is selected can be solved with well-known graph techniques. In fact, the just mentioned problem is exactly the problem known as maximum clique problem, that is, finding the largest subset of nodes in the graph such that every two distinct nodes in the subset are connected by an edge. Since that there are no edges drawn between different nodes for the same mention, or in other words, two candidates of the same named entity mention are not connected, finding the maximum clique ensures that no more than one candidate for each textual mention is selected.

Generally, a graph theory technique can also be incorporated in a system not as the main solution technique, but rather as a supplement technique to achieve the final result, as in drNER [28], a system that will be discussed in the next paragraph.

Grammar (rule) based. The NER task can also be solved using grammar rules, including a final graph based step. DrNER [28] is a system to extract information from evidence-based dietary recommendations from unstructured text data that is not annotated. Because it would require too much time and effort to produce the rules, drNER's rules are not associated with the characteristics of the entities (or entities attributes) but rather they are Boolean algebra rules that allow the system to recognize the phrases in which the entities are mentioned, and grammar rules to search for useful entities. The first step of the process is splitting the text using sentence segmentation (using Apache OpenNLP Maxent sentence detector [29]) and default sentence chunking (using Apache OpenNLP Maxent chunker [30]). Then, if specific conditions on a certain chunk are met on a sentence, that sentence is further split in order to extract more information that could have remained hidden. After this, each sentence segment is analyzed in order to detect and determine the entities mentions using the aforementioned Boolean algebra rules, under the form of Boolean functions. Furthermore, to perform this task, some NLP methods (such as composing trigrams and evaluating them based on the defined Boolean functions; or tokenizing each sentence and linking each token to a dictionary) and matrix theory techniques are used as well. The result from this first step of the drNER method are sets of entities mentions for each entity. Finally, entities are selected and extracted based on the information acquired during the previous step, by representing each sentence as a graph where each chunk is connected only with its predecessor and successor chunks, and the search for the initial node of the graph - corresponding to the predicate of the sentence - can begin, by defining the rules that the initial node is the verb chunk that is closest to the root of the sentence (in terms of number of edges from the verb node to the root node) and it is located in the verbal phrase that is closest to the root. Then, some grammar-based searching rules (i.e. based on text syntactic analysis, by tagging each entity with the P (Predicate), S (Subject) and O (Object) tags based on the current evaluated chunk) are defined to let the system understand which entities are useful.

2.3.3 Semantic Role Labeling

In Semantic Role Labeling (or Semantic Annotation), the main goal is to understand the meaning (or semantic role) of a sentence, by assigning labels to words inside the sentence, such as understanding who is the *agent* that is doing something, what is the agent's *goal* and the expected *result*. Hence, Semantic Role Labeling differentiates itself from the POS tagging task because the latter tries to classify each word of a sentence in a part-of-speech category, while the former tries to understand

the semantic roles of words within the sentence.

A better understanding of semantic roles of words can be beneficial to fields like automatic question answering and text summarization, since the output quality of these systems is highly a□ected by a deeper semantic understanding of a text, rather than only syntactically analyzing it.

An earlier work [31] in the Semantic Role Labeling field aimed at solving the problem that syntactic parsers had at the time, that is, the produced syntactic analysis didn't represent the full meaning of the sentence that was parsed. As an example, consider the sentences "John broke the window" and "The window broke". In the first sentence, a syntactic analysis would assign to "the window" the tag of verb's direct object, while in the second sentence "the window" is the subject. In this example is quite evident that although the underlying semantic role of "the window" is the same in both sentences, this is not displayed nor can be inferred from the syntactic analysis. This specific semantic-role annotation process aims at improving the syntactic structure of the Penn Treebank corpus [32] by constructing another corpus - called Proposition Bank - where it is possible to distinguish the di□erent roles played by a verb's grammatical subject or object in order to facilitate the development of better domain-independent language understanding systems. In fact, the Proposition Bank corpus can be used as a training dataset for supervised automatic role labelers. The overall pipeline works as follows. In the first step, samples of the sentences from the corpus containing the verb under consideration are examined and a frameset is constructed, that corresponds to a collection of roles matching a distinct usage of the verb. Then, the collection of framesets for a specific verb creates the verb's frames file. After this, the annotation process can begin by running a rule-based argument tagger [33] on the corpus, obtaining an 83% accuracy. Finally, the output of this tagger is corrected by hand where annotators can access both the frameset descriptions of any verb and the full syntactic parse of any sentence.

As previously mentioned for the POS task, even Semantic Role Labeling applications are not useful by themselves, but rather are used in a larger pipeline to achieve the desired result. As an example, a system [34] that exploits Semantic Role Labeling to better perform the Open Information Extraction task will be discussed in Section 2.3.4.

2.3.4 Open Information Extraction

Open Information Extraction (OIE) is the task that aims at extracting relation tuples, triples or generally any *n*-ary propositions from text, without requiring a pre-specified vocabulary (that is, the schema for these relations does not need to be specified beforehand), by identifying relation phrases and associated arguments in arbitrary sentences. More specifically, with the term "relation" are identified all the connections between two entities. As an example, a possible valid relation can be $isFriend(Mark; Paul)$ to express that between two person entities (Mark and Paul) there is a friendship relation.

Due to its nature, OIE based systems produce structured outputs, such as relation ontologies or single relations expressed as *n*-ary propositions. Indeed, the two main sub-fields of OIE are: Ontology Learning and Relationship Extraction.

Ontology Learning

A general explanation of what an ontology is, was already discussed in Section 2.2. The difference though, is that the systems described later in this section learn from an unstructured source, rather than a semi-structured one.

To overcome the problem that the outputs of multiple OIE systems contained several uninformative and incoherent extractions, a system that automatically constructs an ontology of relations - called ReVerb [35] - was introduced. ReVerb is an open extractor based on two constraints that should produce better outputs, which are assigned a confidence score using a logistic regression classifier. The two constraints that ReVerb is composed of, are a syntactic constraint and a lexical constraint. The syntactic constraint states that every multi-word relation phrase must begin with a verb, end with a preposition, and be a contiguous sequence of words in the sentence. The benefits of including this constraint are basically two. First, it eliminates incoherent extractions, and second, it reduces uninformative extractions by capturing relation phrases expressed by a verb-noun combination. The lexical constraint states that a valid relation phrase should take many distinct arguments in a large corpus, and the benefit that produces is to separate valid relation phrases from overspecific ones. In fact, this constraint is necessary for the model since there could exist phrases that do satisfy the syntactic constraint but are not relational. Although the precision of the extractor is increased due to these constraints, for the same constraints the obtained recall is reduced.

However, ReVerb and similar OIE extractors, have two important weaknesses: they only extract relations that are mediated by verbs, and they completely ignore context. In fact, by only extracting pieces of information that are mediated by verbs, these systems are missing other important pieces of information that are mediated via other syntactic constructs such as nouns and adjectives. Moreover, the context is ignored, since the analysis of a sentence is only locally performed.

OLLIE [36] provides a solution to both of these problems. To include other mediation part-of-speech that are not only verbs - therefore expanding the syntactic scope - first, it automatically creates a large training set by observing that almost every relationship can be expressed via a ReVerb-style verb-based expression. Then, it learns general open pattern templates over this training set, where an open pattern template encodes the ways in which a relation may be expressed in a sentence. Finally, these pattern templates are applied at extraction time to extract binary relations from a new sentence.

To be able to consider the context, different cases can be identified and hence handled by OLLIE, where the relation is not a fact, but rather is hypothetical or conditionally true. The overall result produced a system that has an Area Under the Curve (AUC) metric which is 2.7 times the AUC of ReVerb.

Relationship Extraction

While the main scope of both ReVerb and OLLIE is to create an ontology of relations, in this section the actually used methods to extract and recognize these relations will be described. Three are the main methodologies used to perform relationship extraction: Pattern (cluster) based, Rule based or Machine Learning based.

Pattern (cluster) based. A pattern (or cluster) based system tries to recognize similar patterns to extract useful relationships. AliBaba [37] extracts biomedical knowledge from PubMed [38] (a

collection of biomedical scientific publications) in the form of biological entities (proteins, diseases, drugs etc.) and semantic relationships between those entities (protein-protein interactions, association of a gene to a disease, etc.). The approach used by AliBaba is pattern-based, which is known to provide high precision results, at the cost of increased complexity. From a training sample, patterns are inferred by means of POS tags and word lists to generalize observed sentences. A mixture of sentence clustering and multiple sentence alignment techniques to derive patterns from sentence examples are then applied. The system is generalizable enough, since the patterns are clustered based on their similarity, defined as the weighted mean edit-distance [39] between them. Then, a "consensus pattern" is derived from a cluster, which may be thought of as the "average" of all patterns in that cluster, in a sort of condensed representation of the patterns contained in a cluster. Essentially, AliBaba is a method for learning language patterns from a database of pairs of related objects, which enables the system to extract information on the result of a query - from PubMed in this case - while other systems have to work on the entire corpora. Finally, AliBaba is domain-independent: PubMed was only a usage example, but it can easily be adapted to entirely different scenarios from biomedical information extraction. An important remark has to be done: as AliBaba used PubMed as a usage example, even the IE system presented in this work uses the drugs reimbursement reports only as a usage example, but the system is generalizable enough to be used in different domains.

As mentioned in Section 2.3.3, an approach [34] that exploits Semantic Role Labeling to perform an OIE task works as follows. This approach tried to overcome the problem that in the biomedical domain, the majority of relationship extraction methods only focus on extracting high precision relations on a very small pre-defined set of relations of interest, also requiring either hand-crafted extraction patterns or hand-labeled training data. All these problems make the available methods very difficult to scale. The system is an unsupervised and scalable method to perform relation extraction and relation synonyms identification tasks based on semantic annotation (or labeling) of textual sources. This method relies on two basic assumptions: a domain knowledge resource and a semantic annotation tool are available. As a domain knowledge source, the UMLS [40] Semantic Network was chosen, that is an upper level ontology of medicine composed by two hierarchies (entities and events) of 135 semantic types. On the other hand, the role of the semantic annotation tool is to map recognized entities in the text to concepts in the knowledge resource of choice. Then, on the annotated text is performed a pattern extraction procedure based on the labels that were given by the semantic annotation tool. The pattern extraction procedure extracts candidate relations in the form of a triple (argument, relation, argument). After this, the synonymy model is used to determine the similarity between two relations, and it is based on a statistical model that computes the joint probability of generating two relation strings for any pair of semantic types. This component is necessary because a same relation string can be applied over different domain and range types. As an example, both the relations "to_treat_with" and "to_inject_with" hold for the semantic types "(Human, Drug)", but not in other contexts. These probabilities are then used to cluster abstract semantic relations. In this step, relation candidates that are considered to be similar based on the assigned probability are clustered together. Based on an algorithm - linear with respect to the number of relation string pairs - that takes into account the distribution probabilities of the semantic types of the arguments of the candidate relations, synonymous relations are clustered together.

Finally, a hybrid approach (Pattern based and Rule based) called SemRep [41] will be described in the next paragraph.

Rule based. Rule based approaches try to extract relations by defining some kind of rules. SemRep [41] is a system able to extract a range of semantic relations from Medline citations on pharmacogenomics. The extraction procedure of this system is composed by different phases. First, it produces an underspecified syntactic analysis, where the most important step is the identification of simple noun phrases. Subsequently, through the use of MetaMap [42] the system maps these phrases to concepts, obtaining a structure where the original phrase is tagged and mapped, and that constitutes the input for the final phase. In this final phase, indicator rules are used to map syntactic elements (e.g. verbs and nominalizations) to predicates (e.g. TREATS, CAUSES) in the already mentioned UMLS Semantic Network [40]. An output example of the indicator relation that can be obtained is as follows: "Pharmacological Substance" CAUSES "Disease or Syndrome". Finally, two concepts that can correctly be substituted as arguments for this structure are "Phenytoin" and "Gingival Hyperplasia" because their semantic type matches the types in the indicator relation given by the Semantic Network, thus obtaining the relation: "Phenytoin" CAUSES "Gingival Hyperplasia". Moreover, the system can be further improved by noting that most of the missed relations were caused by the used Semantic Network. In order to solve this problem, the Semantic Network was modified by including five Semantic groups (or clusters) to organize the semantic types into broader semantic categories considered to be relevant for the clinical domain (e.g. "Substance" or "Pathology").

Machine Learning based. Finally, recent state-of-the-art methods employ ML based techniques in order to extract relationships. In ML, the relation extraction task can be seen as a binary classification problem where a classifier takes as input pre-annotated pairs of entities and aims at identifying the relation between them [43]. Four groups of features are used to classify the relation: 1 - distance-based (e.g. the number of words between entities or the number of punctuation characters between entities), 2 - word-based (e.g. bag-of-words or bag-of-entities), 3 - embedding (e.g. the vectors obtained from pre-trained word embedding models for each entity or the similarity measure between entities embedding vectors), and 4 - knowledge-based (e.g. the UMLS [40] semantic types of entities represented with binary vector or the number of entities co-occurrence in biomedical texts). With these features, a set of independent Random Forest classifiers - one for each relation type - is trained, and the tuning of the parameters is performed using a 5-fold cross validation procedure. Furthermore, for comparison reasons, a BERT (Bidirectional Encoder Representations from Transformers) Neural Network is also trained, which is a relatively novel model for NLP analysis presented by Google [44]. By means of some experiments, it appears the distance and word-based features are valuable for the classifier, since it is shown that they considerably improved the overall performance. Furthermore, the prior knowledge (which is coded in the knowledge features) improves the results only on particular relation types. The resulting classifier was shown to outperform state-of-the-art models as BERT which leads to the conclusion that the context between entities, included in the used features, plays a crucial role in relation extraction.

2.3.5 Metadata Extraction

The task of Metadata Extraction focuses on extracting particular pieces of information - considered to be of high interest - from a text, and give them a structure. The underlying motive of automatically performing metadata extraction is that the process of manually extracting and entering metadata is

time-consuming and the chance for errors is high, while automatic extracting and entering metadata can speed up the process and grant accuracy. Regarding this task, a sub-division can be drawn between Rule based and Machine Learning based methods.

Rule based

Rule-based systems can be further divided into two categories: User provided Regular Expressions and Automatically Learnt Regular Expressions.

User provided Regular Expressions. A common area of research regarding metadata extraction is extracting information such as title, authors, abstract etc. from scientific articles, through the usage of Regular Expressions (RE). Regular expressions are sequences of characters specifying search patterns in a text. The search is very similar to the search that is commonly known and performed on a document, but regular expressions differentiate themselves in that the searched text is not known beforehand, but is encoded in a pattern. As an example, a normal search for a date in a document is done by typing the date (e.g. 01/02/1995), while using regular expressions it is possible to write as follows:

```
^[0-3]{1}[0-9]{1}/[0-1]{1}[1-2]{1}/[1-9]{1}[0-9]{3}$
```

to match all the dates in the format dd/mm/yyyy in the document.

An example of a rule-based technique [45] that uses RE for extracting metadata such as title, authors, and abstract from scientific papers works as follows. The extraction technique begins by utilizing a web-crawler to look for a candidate scientific paper in PDF format on the Internet and then it downloads it. Subsequently, the system transforms the PDF file to a text and XML files, and then determines whether or not this is a scientific publication, by means of mostly two rules: first, if there are words like “IEEE”, “ACM”, or “PROCEEDINGS” in the first page then it is a scientific paper; second, if there are words like “abstract” or “introduction” in the first two pages, or “references” in the last two pages, it is a scientific paper. If the document is recognized as a scientific publication, the aforementioned metadata will be extracted and saved in OWL format in the ontology database. The extraction method is based on human behavior when reading a paper: guided by visual and spatial knowledge, humans are able to identify which part of the document contains the needed metadata, hence, providing rules to retrieve these pieces of information was the technique of choice. An example of this, are the following rules, derived after studying different layouts of hundreds of papers: 1 - The header information including title, authors and abstract are located on the first page. 2 - The title is located on the upper portion of the first page and it usually uses the largest font on the first page. 3 - Authors are listed under the title. 4 - Abstract appears between titles and the main body of the paper and has a certain prefix such as “abstract”. However, these rules are found to be not correct in all cases, since there could be some exceptions. In fact, in some papers, the first letter of the main body uses the largest font on the first page, or in other papers not all abstracts have the prefix such as “abstract”. Handling these exceptions is the hardest task of rule-based IE because finding them may require a good amount of manual inspection and sometimes even luck. However, once an exception is found, a new rule is added to the system, so that it will be able to handle the exception, allowing to cope with more kinds of scientific papers, obtaining an accuracy improvement as well.

A recent example [46] of a rule-based approach using manually defined RE to extract metadata from scientific PDF documents works as follows. In this case, the metadata to be extracted are: Title, Author, Email, A□liation, Country, Section heading/title, Table caption, Figure caption, and Funding agency. First, the PDF document is converted to an XML document through the usage of an online converter [47], which can provide di□erent useful information that can be used to define strong rules, such as the utilized font, the font size and other geometrical features. Then, several extractors were defined based on some rules, which are able to recognize the needed information. For example, considering the Title to be the text having the largest font in the header section provides an F1 score of 100%, in contrast with the previous approach. The reason for this is to be found in the set of documents that was used to compute the metric, and one can conclude that the first approach considered a broader set of documents having di□erent layouts, as in some of them it was the first letter of the main body that uses the largest font, and not the whole title.

To identify the A□liation information, several keywords - Laboratory, Intelligence, Institut, Division, Faculty, University, College, Universit, Educational, Department, School, Centre, Institute, Group, Universitat, Escuela, Engineering, Dept, St., Research - were defined by critically analyzing the research articles in the training set, and the A□liation is considered to be the part of text after these keywords. As a final example, the Country names are extracted from the A□liation section: a dictionary of predefined 195 countries and their short forms was used to identify the Country, obtaining an F1 score of 100% for A□liation and 93% for Country extraction. Similar rules were manually defined to extract all the other required information.

Automatically Learnt Regular Expressions. Although defining a regular expression is a well-established technique to perform several pattern matching and text processing tasks, manually constructing a regular expression can be a tedious and error-prone process for obtaining a tool that often can only be used to match a specific text on a specific domain (e.g. the same regular expression may not work for documents having di□erent layouts). For example, consider the regular expression provided in the previous paragraph to match a date. If the date in the document is not in the format dd/mm/yyyy, but it is in the format yyyy/mm/dd, that regular expression would completely fail to do the task it was defined for.

A first method [48] to automatically generate regular expressions from examples is based on genetic programming. The overall benefit is that the method doesn't require the final user to have any familiarity with regular expressions syntax. The examples are provided in the form of strings, where each string is provided together with the substring that has to be extracted. Then, the system is able to automatically construct a regular expression that minimizes two fitness functions: 1 - the sum of the edit distances [39] between each detected string using the current configuration and the corresponding desired string, and 2 - the length of the obtained regular expression. Hence, the problem is a multi-objective optimization problem. The genetic programming algorithm starts with an initial random population of individuals (candidate regular expressions) represented as abstract syntax trees. Then, genetic operators such as "mutation" or "crossover" are applied to some individuals of the current population in order to randomly produce changes, and the fitness function is next computed for each individual. After this, a new population is constructed from the individuals that have obtained the highest fitness in the previous generation and the same genetic operators are applied to the new population. This process is iterated until an individual obtained

a perfect fitness score or a predefined maximum number of generations have evolved. Hence, the individual with the highest fitness score is given as output as the automatically generated regular expression. Tested on 12 different extraction tasks (e.g. URL extraction, IP address extraction etc.) and by providing around 50 examples for each task, this system proved to generate regular expressions that obtained precision, recall, and F1 score around or higher than 90%.

However, other systems have been proposed in the literature. SystemT IDE [49] is an integrated development environment for extracting information and specifying rules. Using this system, it is possible to perform a variety of extraction tasks, due to the declarative language AQL - Annotation Query Language, specifically developed to write rules under the form of queries, with a very similar SQL syntax - that is expressive enough to handle very complex rules and patterns. The architecture of SystemT IDE contains different components. First, through the AQL declarative language, a user can express a rule, which contains a regular expression that expresses a pattern that has to be matched. Then, an *Annotation Viewer* allows the user to examine the extraction result retrieved by the AQL query and the *Annotation Provenance Viewer* displays the provenance of that result, in order to quickly understand the rule's component responsible for a mistake. After this, a *Contextual Clue Discoverer* tries to detect frequent patterns by clustering the context surrounding the extraction results (e.g. for a telephone number extraction task, clustering the region of text between incorrect telephone number pairs may reveal that not only telephone numbers were retrieved, but also fax numbers). Hence, this component can speed up the process of collecting important contextual clues from the input document. Furthermore, SystemT IDE is completed by two additional side features, being the *Regular Expression Learner* and the *AQL Rule Refiner*. From a simple regular expression defined by the user that captures with high recall but low precision the target entity, and a set of labeled examples for that rule, through a hill-climbing algorithm the Regular Expression Learner is able to generate a more complex rule having a much higher precision while maintaining a similar recall, that is, the new complex rule will capture most of the positive examples given as input, and as few negative examples as possible. Finally, the AQL Rule Refiner automatically suggests concrete refinements at the level of a defined rule, increasing the precision of the obtained results. The user can automatically choose one of the proposed refinements to the rule or can even enhance the chosen proposed refinement with domain knowledge before applying it.

SystemT IDE proposed the idea of defining a new language - AQL - through which express the rules to retrieve information. As previously discussed, trying to develop a language that can be the counterpart of SQL for unstructured data sources is a flourishing research field. Other than AQL, other minor languages - such as xLog [50], UIMA [51] and GATE [52] - have shown that formalisms of rule-based IE systems are possible [53].

The methodology of defining a language tailored on expressing rules is the same methodology behind the simple language that will be created in this work. However, there are some differences between our new language and the already existing ones. For example, in AQL it's only possible to extract some data from a text using regular expressions, which can be a critical factor when the same rule is applied over different documents having different layouts. Moreover, defining a regular expression that alone is able to sufficiently describe the desired information to extract can be very challenging, due to the complex syntax used by regular expressions. Finally, refining the same regular expression such that it can be used across different document layouts can be a tedious procedure, not always possible. In our language, not only regular expressions are fully supported,

but the way the language grammar is designed allows to express different regular expressions in different statements, use these statements to score each piece of information with a probabilistic value and then combine these values based on which regular expression they matched. In this way, a sort of "divide and conquer" approach has been used: rather than further complicate the same regular expression to be used in documents having different layouts, different regular expressions can be expressed in statements, and a system that is able to give a score (giving the user complete control over the weights that determine these scores) to each piece of data, for each defined statement has been designed. Furthermore, unlike AQL, in our language regular expressions are only one of the different ways in which a user-defined rule can be expressed. In fact, it is possible to express a different variety of rules, spanning from - for example - specify that the desired information can be present in the first page of the document, or that it's present in a certain user-defined list. To the best of our knowledge, the method of expressing different statements, each one containing a rule that better specifies a characteristic that the desired data might (or might not) have, and then use these rules to give a probabilistic score to each isolated piece of information in the given text is what really differentiate our system from all other proposed systems in the literature, and is also what allows the system to be used across documents having different layouts, which was considered a major limitation of rule-based IE systems.

Furthermore, our system differentiate itself from other systems like UIMA and GATE because unlike them, its semantics is better defined (although not yet *formally* defined), meaning that a better reasoning on possible optimizations and changing can be performed. In fact, optimizing UIMA and GATE has proven to be difficult exactly for their lack of semantics definition, for example, it is unclear whether an IE program in such languages produces the same result if the execution order of certain rules or parts of the program workflow are changed. Hence, without a proper semantics it's difficult to know what would be the result of an optimization on the program result. In our language the semantics is defined on a high level, allowing abstract reasoning. For example, it's clear that changing the order of certain rules does not affect at all the retrieved result.

Finally, xLog is performing the IE task in a completely different manner: it decomposes an IE task into smaller subtasks (called *procedural predicates*, that are pieces of code in e.g. C++ or Java cleanly encapsulated and "plugged" into a Datalog rule) and then re-combines them using Datalog, a declarative logic programming language. Moreover, even xLog heavily relies on pattern matching using regular expressions.

Machine Learning based

Systems based on ML to perform metadata extraction can be divided into Statistical based and Classifiers.

Statistical based. An important research effort has been done in extracting metadata through the Hidden Markov Model (HMM). An example [54] proposed an efficient automatic metadata extraction hybrid approach (it uses both rules and ML) for retrieving bibliographical information from a variety of digital research papers and academic documents in PDF formats. The approach relies on three key steps: first, using the PDFBox software [55] to extract the font size and text information, second, the rule-based method is utilized to identify the titles of the papers - that is considered to be the text with the largest font - and third, if the rule-based method fails at extracting

the required information, the authors and titles are extracted using the HMM. Finally, the obtained results (titles and authors) are subsequently transmitted as query strings to various digital libraries to retrieve the rest of the metadata. The HMM one-state model, which is the proposed model, proved to be more accurate and therefore obtained a higher performance compared to the HMM multi-state model. This method showed a high performance also in the extraction of bibliography data even though they have been extracted from documents having di□erent formats and structures.

Classifiers. Recent approaches in the ML field make use of classifiers. In [56] a ML based pipeline to extract information from PDFs in the domain of nanomaterials synthesis is proposed. More specifically, the aim is to extract so-called "recipes", which are sets of specific actions that are applied to some recognized base materials in nanomaterials synthesis experiments, that are considered to be the metadata to extract. The extraction pipeline works as follows. A configuration file of seeds (URLs) is passed to a web-crawler that constructs from them a B-Tree and from the nodes of this tree, it determines which ones are downloadable PDFs and proceeds to download and save them in the output directory. Then, for each PDF, an open-source tool called MATESC [57] is used to group lines into paragraphs and paragraphs into sections based on font size, font type and character spatial location in order to generate a text file as output. After this step, a binary Naive Bayes model is applied and each sentence is classified either as relevant or irrelevant. To train this model, more than 2600 sentences were manually labeled as relevant or irrelevant and di□erent types of features have been used, such as N-grams or a matrix of term frequencies (bag-of-words). Finally, another open-source tool for semantic text mining in the chemistry field called ChemicalTagger [58] is used on the sentences that were classified by the model as relevant to generate an XML file where the input text file has been tagged, making it possible to parse the XML file to extract the recipes from the sentences containing specific tags. ChemicalTagger comprises 5 steps in order to produce a tagged XML file from an input text string: text normalization, tokenization, tagging, phrase parsing and action phrase identification. The first three steps are similar to what is described in Section 2.3.1, and involve the tasks of cleaning the input text from non-printing characters such as non-breaking spaces or tabs, splitting the text into a sequence of meaningful elements called tokens (usually words or composition of multiple words) and tagging each token by using first a chemical entity recognizer, and then a regex and POS tagger for the tokens which have not been identified. Then, the phrase parser component assigns syntactical structure to the tagged text, by creating the Chomsky [59] tree structure of a sentence, assigning S for sentences, D for determiners, N for nouns, V for verbs, NP for noun phrases and VP for verb phrases. To do that, a formal grammar was defined to describe the type of language in the corpus by defining some rules. For example, the top-rule in the grammar is a *sentence* S and it was defined to be made up of a *noun phrase* NP and a *verb phrase* VP. Then, the left non-terminal token noun phrase is selected, and the grammar says that a noun phrase can be made up of a *determiner* D and *noun(s)* N. By running this grammar over the tagged text, it is possible to obtain the Chomsky tree structure of each sentence. Finally, roles are assigned to the parsed phrases by the action phrase identification component. For Chemicaltagger, roles correspond to actions carried out during a chemical synthesis (e.g. adding, dissolving, evaporating etc.). By using linguistic context such as action names (manually pre-defined) and their position in the text, it is possible to detect the role of a sentence.

2.4 PDF Text Extraction

As discussed in previous sections, a number of IE methods relied on different ways to convert PDF files to plain text, which is considered to be the input of all processes. Although it sounds simple, converting a PDF file into a text file is a difficult task. The PDF document is a layout-based format emphasizing more the fonts and placement of the single characters rather than focusing on the semantic aspects of the text - which is more important - and for this reason, extracting a coherent sequence of logical text is far from trivial.

A crucial benchmark [60] that compares different PDF text extraction tools with the objective to use the extracted text for information/metadata extraction purposes has been conducted. The proposed methodology involves TeX or PDF file parsing, identification of the logical text blocks, and serializing the logical text blocks to files. Then, fourteen different PDF extraction tools are compared and evaluated based on different evaluation criteria. In conclusion, Icecite [61] slightly outperformed the other extraction tools, but also PDFExtract [62] and PDFMiner [63] obtained very good results.

However, PDFplumber [64] is the method of choice to convert PDF documents into text documents. The reason for this choice is a result of an exclusion process performed on Icecite, PDFExtract and PDFMiner. Starting from PDFExtract, this library (as of today) received its last GitHub commit 11 years ago, pointing to the fact that is probably not maintained anymore, thus not updated. Icecite is a tool written in Java, while the application that will be developed in this work will be implemented in Python, therefore, the two programs could have a gap to communicate, which can be complex to solve and far more difficult than using an already Python implemented library, as PDFMiner. PDFplumber is a Python library that is built on top of PDFMiner, providing everything that PDFMiner provides, plus more useful tools that can be used when needed. Hence, PDFplumber was chosen as a PDF to text converter tool.

3 Problem Statement

Manually extracting and entering metadata is a time-consuming and error-prone procedure, while automatic extracting and entering metadata can speed up the process and grant accuracy. Hence, the main focus is about developing an application able to extract metadata from an unstructured source as a PDF document written in natural language, using a new simple programming language developed in order to write specific extraction rules. The proposed solution for developing such a programming language will be extensively explained in Chapter 4.

The application will be tested on the task of extracting different types of metadata from medicine reimbursement assessment reports published by Health Technology Assessment (HTA) organizations. These reports assess whether the balance of benefits and costs of medicines justify their reimbursement. In this domain, manually extracting data takes indeed a lot of time and in several cases is a limiting factor to comparative health policy research. By increasing the speed, more and more extensive research can be performed to improve health policy. Moreover, it is known that HTA organizations of different countries may classify the same medicine into different reimbursement classes, due to the diverse approaches they adopt to assess new technologies [65]. Hence, an IE tool that is able to automatically extract relevant information from medicine reimbursement reports and organize the extracted information in a structured database can help European policy-makers to understand the extent of these differences and therefore help the decision-making process.

For each HTA organization and for each medicine, a public report is compiled and published in PDF form on the considered HTA's website. The five metadata that have to be extracted from these PDF reports are the medicine brand and generic names, the assessment date, the indication and the reimbursement class, and they will be explained in Chapter 5.

Hence, the language that will be developed has to be expressive enough to allow a final user to write queries that can recognize and extract specific metadata (such as medicine names or reimbursement classes), dates, or even entire sentences (such as the indication), independently from the application domain. Therefore, the language will be designed by defining different constructs through which it will be possible to express the extraction rules, in order to handle the extraction process.

Problem setting. Given as input a configuration file (`txt`) where a query has been written in the new programming language, and a PDF document, the final objective is executing the configuration file assuming it contains a valid query to extract a certain type of metadata from the PDF document, producing as output a structured file (JSON) containing the extracted metadata.

4 Solution

The intended way to solve the problem described in Chapter 3 is to develop a programming language through which it is possible to write queries allowing the extraction of data from unstructured sources, such as PDF documents written in natural language. The presented programming language has been named ProbQL, which stands for *Probabilistic Query Language*. In order to accomplish this task, different challenges need to be addressed and different decisions and assumptions have been taken, which will be discussed throughout this Chapter.

4.1 Language overview

ProbQL is a probabilistic rule-based query language, hence, its structure resembles the structure of most of other query languages, in particular, SQL. However, in contrast to other existing query languages, ProbQL will be used to extract data from unstructured data sources (PDF documents) rather than from structured databases (either relational or non-relational). In this regard, a major challenge regarding the extraction of data from PDF documents is that even in the same domain, the same data have to be extracted from PDF having different layouts and structures. Therefore, the same query has to be executed once for each PDF document, without any knowledge of which type of structure - amongst the possible ones - that PDF document will have. In fact, it is possible that the data to extract is located in different locations based on the structure of the document. In order to create a language that can be used by different users having different data to extract from different domains, it is impossible to encode and pre-compute in a language all the possible existing documents (and their structures) in all possible domains. Rather, what a query language should aim for, is to define a set of constructs and a logic that encodes how to combine them in order to write a query in ProbQL, so that users can express their own rules on how to find a data in a certain document. Furthermore, when it comes to analyze a PDF document to extract some metadata, it is frequent that a probabilistic problem arises. Indeed, metadata may be found as a result of different probabilistic rules conjunction. To better explain this problem, consider the following example. It has been noticed that the *medicine brand name* metadata from the medicine reimbursement reports is probably located at the beginning of the document (is always present on the first page) but can also be located in multiple sections. Most of the time is an upper-case word. Most of the time can be found in bold. Sometimes is a frequently repeated word throughout the document, and is probably a non-English word. Finally, sometimes is followed by a number and the word “mg”.

Each of these statements represents a rule by itself. However, depending on the structure of the analyzed PDF, some rules may be true for the *medicine brand name* metadata, and others may be false. In some domains and for some types of metadata, it is simply not possible to define a rule

that will always draw a correct line on every document that separates the right data to retrieve from the rest of the data of the document. Hence, ProbQL provides a way to encode these rules in a probabilistic manner, and will eventually retrieve the data from the document having the highest probability of being the correct metadata that was searched, based on how many rules it satisfies, and on how much those rules count in the probability function that will be defined. In this way, a ProbQL user has complete control over the rules that have to be defined for their problem, and also on how much each rule counts in the probability function definition that will assign a probability value to each piece of data in the document. Finally, another benefit of this method is to provide a tool in which it is possible to encode the final user's domain knowledge into the query they will write in ProbQL, because of the complete control they have in defining the probabilistic rules and the weight of each rule in the probabilistic function. These rules will now be referred to as P-Rules, for Probabilistic-Rules. Thus, a P-Rule is a probabilistic rule that given an item (word, sentence, etc.) returns as output either True or False, based if the item satisfies or dissatisfies the rule.

Hence, ProbQL is a valid initial attempt of dealing with the uncertainty that is present in the unstructured data world.

4.2 Language Grammar

The ProbQL grammar can be divided into its *Informal Semantics* and *Syntax*. First, the Informal Semantics will be described so that when the Syntax will be introduced, a reader will already have an idea of how the query is trying to retrieve the correct data, and will better understand the constructs defined in the Syntax and used to achieve that objective. Moreover, this grammar doesn't provide any information on how the execution of a ProbQL query was implemented. The implementation of the program and all the technical details will be discussed in Section 4.3.

4.2.1 Informal Semantics

The Informal Semantics of a programming language aims at describing its semantics (meaning) using natural language sentences.

A ProbQL query can be divided into 3 parts. The first part is called *Main-List definition*. The second part is called *P-Rules definition*, and the third part is called *Additional-Context definition*.

When a query is executed, the PDF document has already been converted into a `txt` file, which contains one string comprising of all the textual elements that were found in the PDF, and that will be the context on which the query is executed. In this string, to keep trace where in the original PDF document there was a page interruption, the `\p` token is used. The Main-List definition part is in charge of reading that context, slicing the string based on some defined positions (if needed) and producing a list (which is called main-list) by splitting the string based on some token provided by the user. Hence, the output of the Main-List definition part is a list that contains items that were tokenized from the context. As an example, if one knows that the desired metadata is a word and they want to analyze the entire document, during this part they can express to not slice the original context, and to tokenize it based on the space character to produce a valid list of words. Each item that this main-list contains has 5 properties: a text, its starting (START_POS) and ending positions (END_POS) with respect of the context string, a length and a probability, corresponding to the probability of being the correct data, which at this stage is initialized at 0. In fact, the desired

data will be searched in this main-list, and the probabilities of each item will be updated based on the P-Rules it satisfies.

The P-Rules definition part is where the P-Rules are defined. The output of each P-Rule is a Boolean array that maps each item in main-list with either 1 if the item satisfies the P-Rule, or 0 otherwise. Hence, each P-Rule will be tested on each item in main-list, and each P-Rule produces its own Boolean array.

Finally, in the Additional-Context definition part, it is possible to define additional information regarding the results of the query. For example, the probability function that will be used to update the probability values of each item in main-list is defined here. Also, it can be defined the sorting method (descending or ascending) of main-list, in order to have at first position the item with the highest (or lowest) probability of being the correct metadata to extract.

Please note that not all the 3 parts are mandatory in order to define a valid query. In fact, in this language it is possible to write nested queries during the P-Rules definition. Hence, as a general rule, if a user is writing the main query, then all the 3 parts needs to be there for a query to have sense. However, if the user is writing a nested query, different rules apply. In a nested query, only the Main-List definition part is mandatory, while the P-Rules definition part will be there only if needed, and the Additional-Context definition part needs NOT to be there, because it's not needed in the scope of a nested query. Rather, at the end of the execution of a nested query, the main-list that is defined there will be filtered, and only the items for which each P-Rule has been satisfied are kept. By using this method, it is possible to define P-Rules that check whether an item is for example in a list that is dynamically constructed and returned by a nested query. A simple example is indeed a P-Rule that assigns 1 to the items in main-list that are present in the first page of the document, by simply checking if the item is in the element in the first position of a list produced by a nested query where during the Main-List definition part, the context has been tokenized to produce a list where each element contains the text of a page of the original PDF document. Then, in the P-Rules definition part of the nested query it will be expressed a P-rule that assigns 1 only to the first element of this main-list, which will be filtered by correctly keeping only the element containing the text in the first page.

4.2.2 Syntax: General Rules

The syntax of a programming language aims at defining the rules that control the structure of its symbols, punctuation, constructs (even referred to as keywords in this Thesis) and words. These rules needs to be satisfied in order to have a syntactically valid query. Furthermore, Appendix A will provide some examples on the usage of each construct that will be defined in the language.

ProbQL syntactic rules can either be of a general nature, or can relate to a specific construct. While the latter will be described in the appropriate sections, the general rules are listed as follows:

- Each line in the query is an instruction to be performed, and the first word of that line is the construct that expresses the behavior of the program for the execution of that instruction. All the constructs are written in upper-case.
- The Main-List definition part is always required in a query (normal or nested) in order to consider it syntactically valid, while the Additional-Context definition part must not be present in nested queries.

- Every query (normal or nested) must ends with a ENDQUERY keyword, that has to be written in its own line. It can be followed by a closing round parenthesis (e.g. the closing parenthesis of a nested query) and by square brackets (refer to the SQUARE BRACKETS paragraph).
- In ProbQL, it is possible to make comments. Each line that immediately starts with a # symbol in first position is considered to be a comment, and it will be ignored during the execution of the query.
- When an argument has to be passed to a construct, it is passed by distancing the construct name and the argument with a white space, in Haskell style [66]. If multiple arguments need to be passed, the reasoning is the same, and to correctly understand the order of the passed arguments, refer to the corresponding construct syntax definition.
- Every time a quotation mark is needed somewhere in the language, a double quotation marks must be used instead.
- In order to define a valid nested query in a P-Rule, the round opening parenthesis that started the nested query must always be at the end of the same line of that P-Rule:

```
CONSTRUCT (
nested_query)    -> valid

CONSTRUCT
(nested_query)   -> not valid
```

- The starting position of an item in a list is considered to be position 0. The ending position of an item in a list is considered to be position -1. Hence, position 1 corresponds to the element in second position of that list (and so on), and position -2 corresponds to the second last item of that list.

4.2.3 Syntax: Main-List definition

In this part there are 2 main tasks that need to be performed: the slicing of the context string and the splitting given a token. Also, the Main-List definition part must always be present in any query (both normal and nested).

WITH. The slicing of the context string is performed by the construct WITH. This keyword takes two positional arguments as input, corresponding to the starting position sp and ending position ep so that the context is sliced to only keep the text from sp (included) up to ep (excluded). In ProbQL, both in strings and lists, the starting position is considered to be position 0, as in (almost) all famous programming languages. Hence, both sp and ep need to be evaluated as integers. To express to not slice the context string, but rather keep it as a whole, sp must be 0 and ep must be the keyword LEN, to declare that the ending position must be the length of the context, thus, equal to the last position of the last character in context (this time included, so to not exclude the last element). Only in a nested query, the two positional arguments of the WITH statement can assume different values, but they however must be evaluated as integers. For example, let's say that a user may need to assign a value of 1 to an item, if two words after that item there is the word "mg"

(this is a P-Rule example, and this task will be executed by the construct EXISTS). In this case, a nested query needs to be written. For each item in main-list, we need to check whether two positions after that item, there is an item whose text is “mg”. Hence, by modeling the current checked item of main-list with the word $this, it is possible to slice the context string using the WITH construct from the position of the item $this until the end of the context string (LEN) so that when the splitting by words is completed, only two P-Rules needs to be defined: return 1 to the element in position 2 and return 1 if the text of an item in main-list of the nested query matches the word “mg”. This nested query needs to be executed for each item in main-list, and it will correctly return 1 if two words after that item there is the word “mg”. To see an example of this task, refer to Chapter 5.

Using the same reasoning, it is possible to express a P-Rule that assigns 1 to an item if there is a pre-defined item before - and not after, as in the previous case - always through the construct EXISTS. In this case, the WITH statement needs to take 0 as its starting position and the end position of the current item ($this) as its ending position.

Hence, a valid WITH statement can either take the form of one of these examples:

```
(1). WITH int int
(2). WITH int LEN
(3). WITH $this.START_POS int
(4). WITH $this.START_POS LEN
(5). WITH $this.END_POS int
(6). WITH $this.END_POS LEN
(7). WITH int $this.START_POS
(8). WITH int $this.END_POS
```

Where forms from 3 to 8 can only be used in nested queries. When $this.END_POS is used as sp, the slicing will not select the actual position of the last letter of $this, but the immediately next position, so that the last letter of $this will not be selected as first letter of the sliced string. Finally, the WITH construct must always be the first construct of any query (both normal and nested).

TOKENIZEDBY. The splitting of the sliced string (output of the WITH statement) given a token is performed by the TOKENIZEDBY construct. In its basic form, this construct takes one argument that needs to be passed between double quotation marks, and is the token that will be utilized to break up the sliced string and construct the list where each element is an item having 5 properties: text, starting and ending positions in the context, length and probability initialized at 0. This list has been (and will be) referred to as the main-list (or nested main-list if the TOKENIZEDBY it’s performed inside a nested query execution). The token can be completely decided by the final user (and can have di□erent lengths), but the execution performance of the program is optimized when the token is either one of the following tokens:

```
(1). " " :: list of words
(2). "\bigrams" :: list of bigrams
(3). "\trigrams" :: list of trigrams
(4). "." :: list of sentences
(5). "\p" :: list of pages
```

While words, sentences and pages are clear, the meaning of bigrams and trigrams may be not. In this context, a bigram is a pair of consecutive written words, while a trigram is a triplet of consecutive written words. As an example, consider the sentence “Hello my name is Daniele”. In this sentence, the list of bigrams will then be (Hello, my), (my, name), (name, is) and (is, Daniele). While the trigrams of the same sentence will be (Hello, my, name), (my, name, is) and (name, is, Daniele). It is easy to note that the number of bigrams that can be extracted from a sentence containing n words is always $n - 1$, while the number of trigrams is always $n - 2$. These structures can be very important to model specific types of data. For example, a date (e.g. 01 February 1995) can be considered to be a trigram. Hence, if a date needs to be extracted from a document, it can be a clever idea to split the sliced string by using `"\trigrams"` as a token.

However, the program fully supports - at the cost of being slower in its execution - a user-defined token to be used to split the sliced string.

Regarding the optimization, in order to correctly exploit the benefits of the faster method, the assumption that has been made is that in the query, every `TOKENIZEDBY` instruction that is contained in a nested query that is using `$this` in its `WITH` statement, must use the same token to split the sliced string as the first `TOKENIZEDBY` instruction, the one that is producing the main-list. If this is not the case for a user written query, then the optimized fast solution method needs to be deactivated when the execution of the program is performed (see Section 4.3.3), otherwise it will return a wrong result.

In an advanced form, the `TOKENIZEDBY` instruction can also accept multiple tokens that can be passed between double quotation marks, and must be separated by the `AND` keyword. In this scenario, the `TOKENIZEDBY` instruction will produce a main-list containing the items that are constructed from the sliced string, using all the passed tokens. Hence, if the tokens `" "` and `"\bigrams"` are passed, main-list will contain first, all the words that are found in the sliced string, and second, all the bigrams. An example of the syntax to express this instruction can be as follows:

```
TOKENIZEDBY " " AND "\bigrams"
```

The convenience of this advanced `TOKENIZEDBY` instruction is to be found when the same data can assume different forms. For example, is frequent that a person’s name and surname is a bigram, however, this is not always the case. In fact, this data can sometimes be a trigram, and a valid way to correctly extract the actual data in the vast majority of cases, is to write the `TOKENIZEDBY` instruction using both `"\bigrams"` and `"\trigrams"` as tokens. However, when this advanced version of the `TOKENIZEDBY` construct is used, a special attention has to be put in the designing of the rules within the P-Rules definition part, because the success of the query to extract the correct metadata is even more strongly dependent on the quality of the defined P-Rules, due to the new nature of main-list, that is no more containing only homogeneous items.

Finally, the `TOKENIZEDBY` construct must always be the second construct of any query (both normal and nested), following the `WITH` construct.

4.2.4 Syntax: P-Rules definition

The P-Rules definition part starts with the `WHERE` keyword. Hence, when a single line that contains the keyword `WHERE` is read from the configuration file, it means that following, there will be some P-Rules definitions, one for each line, unless the P-Rule contains a nested query to be performed.

In this part, it is possible to express P-Rules, that are rules used to express certain characteristics that the desired metadata is thought to have (eg. being frequent with a certain threshold, being on the first page of the document, etc.), hence giving as output either `True` or `False`.

The syntax of expressing a P-Rule is the following:

```
WHERE
label1: CONSTRUCT_NAME_1 arguments
label2: CONSTRUCT_NAME_2 arguments
```

In fact, each P-Rule is given a label, that will be useful later on, when defining the probability function. As a rule, the labels are capital alphabetic letters (such as A, B, C, etc.). After the label and the colon punctuation mark, a space character needs to follow, and after the space character, the construct and its arguments must be provided.

By default, each provided P-Rule is collected and tested on each item of main-list that is returned by the very first `TOKENIZEDBY`. Then, each item is evaluated either to 1 or 0 based if it returned True or False on the tested P-Rule. Regarding the usable constructs, there are 7 of them that can be used to define a P-Rule. Moreover, in 3 of them (`SUBSTRING`, `IN` and `EXISTS`) it is possible to write a nested query. As a nested query example using the `IN` keyword, the syntax must be as follows:

```
label: IN (
    WITH arguments
    TOKENIZEDBY arguments
    ...Possible P-Rules...
    ENDQUERY)
```

In this example, the presence of a round opening parenthesis in the same line as the P-Rule definition suggests the presence of a nested query that has to be executed afterwards, hence, the parenthesis must be on the same line as the P-Rule definition.

The 4 spaces indentation of the nested query with respect to the main query is not necessary, but it can be added for a better readability of the query. Also, please note that every query (both normal and nested) must finish with the `ENDQUERY` keyword.

`IN`. The `IN` construct is used in a P-Rule to determine if an element in main-list is also in a list (called temp-list) that has to be passed as an argument to this construct. Hence, if an item in main-list is also in temp-list, the construct `IN` will assign 1 to that item, and 0 otherwise. The temp-list can be constructed in four different ways:

- As a list returned from a nested query (thus, temp-list is a nested main-list).
- As a list loaded from a file (see: `LOAD_LIST_FROM_FILE` construct).
- One of the embedded lists (see: Section 4.2.6).
- Manually, using the regular Python syntax.

`IN` is in fact a construct that can accept a list returned by a nested query, as seen in the above example. Moreover, the temp-list can be loaded from a `txt` file through the `LOAD_LIST_FROM_FILE` construct, or it can assume values of one of the embedded lists that will be presented in Section

4.2.6. Finally, a list can also be manually passed to the IN construct, following the Python syntax of expressing a list: each element is separated by a comma, strings must be expressed between double quotation marks and the list must be written between square brackets. As in Python, this list can be heterogeneous, however, the text of the elements in main-list is always a string type, therefore, each element in this hard-coded temp-list is automatically converted to the string type, regardless of its original type.

LOAD_LIST_FROM_FILE. The LOAD_LIST_FROM_FILE construct is used when a user wants to load a list contained in a txt file. There are two arguments that can be passed to this construct, of which the first is mandatory and the second is optional. The first argument is the path (between double quotation marks) where the txt file is located on the machine. The second (optional) argument is the keyword SPLIT. This argument needs to be passed if main-list contains items that are not matching the structure of the items contained in the txt file, but are (for example) multiple words (e.g. bigrams, trigrams, sentences, etc.), while the txt file is only containing words. Then, if SPLIT is passed, an item in main-list will be given a value of 1 only if all the words that is composed by are independently present in the txt file. Hence, the actual presence check is performed on each single word that the item in main-list is composed by, and not on the item itself.

```
label: IN LOAD_LIST_FROM_FILE "file_path"
label: IN LOAD_LIST_FROM_FILE "file_path" SPLIT
```

Each row of the txt file will be converted in its lower-case version, and will become an item of the list returned by LOAD_LIST_FROM_FILE. This construct can only be used following the IN construct, in order to check if the lower-case version of each item in main-list is in the list loaded from LOAD_LIST_FROM_FILE.

NOT. The NOT construct takes as argument another construct that produces a Boolean vector, hence, every construct explained in the P-Rules definition part. What it does, it simply revert the Boolean array produced by the input construct, by switching every 1 to 0 and every 0 to 1.

POS. The POS construct accepts an integer x as argument, and it will simply assign 1 to the element in main-list in position x, and 0 everywhere else. As previously mentioned, in this language the starting position of each structure is considered to be position 0. Hence, $x = 0$ will be the first element of the list.

In the POS construct, x can also be a negative number, expressing the fact to count from the right instead of the left. Hence, $x = -1$ is the last element of the list, $x = -2$ is the second-last element, and so on. Usually, it is effective if employed inside nested queries.

MATCH. The MATCH construct takes a regular expression (written as a string between double quotation marks) as input argument, and it assigns 1 to the items in main-list which matches the given regular expression, and 0 to the all the others. In the provided regular expression, eventual spaces need to be written as “\s”, and not using a space character.

EXISTS. The EXISTS construct assigns 1 to an item in main-list if something (expressed as a nested query) exists after or before that item. Hence, the EXISTS construct must always contain

a nested query to express what has to be searched and where it has to be searched. Usually, the nested query contains the MATCH and POS keywords in the P-Rules definition part.

As an example, consider the following scenario, to extract the medicine brand name metadata (ABRAXANE) from an already tokenized main-list:

```
context :: "SUMMARY ABRAXANE 5 mg/ml"
main-list :: ["SUMMARY", "ABRAXANE", "5", "mg", "/", "ml"]
```

```
A: EXISTS (
    WITH $this.START_POS LEN
    TOKENIZEDBY " "
    WHERE
    A: MATCH "mg"
    B: POS 2
    ENDQUERY)
```

As previously mentioned, for this metadata, it has been noted that 2 words after it may be the word “mg”. Hence, since we are trying to search for the existence of a word somewhere after a certain element, it is possible to use the EXISTS construct. Since the WITH statement contains the word $this, it means that the nested query has to be performed for each element in main-list. Thus, the first time it is performed, we will have $this = “SUMMARY”, and the nested main-list (produced by the nested TOKENIZEDBY) will be the same as main-list. Then, the P-Rules definition part starts, and the Boolean vector returned by the two rules are as follows:

```
A: [0, 0, 0, 1, 0, 0]
B: [0, 0, 1, 0, 0, 0]
```

Since we are in a nested query, the conjunction of all the Boolean vectors is performed in order to filter the nested main-list - where only the elements having a value of 1 will be kept - obtaining a vector having 0 to all of its elements. Hence, an empty list is returned, indicating that for the word “SUMMARY” it doesn’t exists an item “mg” after 2 words. Therefore, a 0 is assigned to “SUMMARY”.

Then, the second iteration of the nested query is performed, and now we have $this = “ABRAXANE”. This time, the nested WITH clause will slice the context by excluding the first word, and the nested TOKENIZEDBY will produce the same list as main-list, but without the item having the text “SUMMARY” in it:

```
context :: "SUMMARY ABRAXANE 5 mg/ml"
sliced context :: "ABRAXANE 5 mg/ml"
nested main-list :: ["ABRAXANE", "5", "mg", "/", "ml"]
```

Then, the P-Rules will produce the following Boolean-vectors:

```
A: [0, 0, 1, 0, 0]
B: [0, 0, 1, 0, 0]
```

And the conjunction of these vectors is producing the vector: [0, 0, 1, 0, 0]. When the nested main-list is filtered by keeping only the elements having a value of 1, a list containing the word “mg”

is returned. Hence, since the returned list for the word $this = “ABRAXANE” is not empty, it means that 2 words after it actually exists the word “mg”, assigning 1 to the word “ABRAXANE”.

The procedure will finish by checking all the other elements in main-list, correctly assigning a 0 to each element.

Hence, with the keyword EXISTS (combined with MATCH and POS) it is possible to express a P-Rule that checks the existence of an item after (or before) a certain item. The condition “before” can be expressed by only changing the WITH clause (by saying for example: WITH 0 $this.END_POS) and the POS clause (by using negative numbers depending on the desired position).

Finally, it is possible to match different types of data (words, sentences, bigrams, etc.) than the type in main-list, by simply expressing the correct token in the nested TOKENIZEDBY.

SUBSTRING. The SUBSTRING construct takes two arguments, of which the second one is optional. In fact, SUBSTRING can be used in two different ways: first, to check if an item in main-list is a substring of something, and second, if something is a substring of an item in main-list.

Regarding the first usage, the mandatory argument can either be a nested query or a hard-coded string passed between double quotation marks. In this case, since the result of a nested query is always a list, there is the need to introduce in the language a method to access to a certain element stored at a certain index inside a list. To do that, as is usually done in Python, the square brackets are used, and they need to be placed right after the closing round parenthesis of the nested ENDQUERY keyword. Inside the square brackets it needs to be specified an integer number corresponding to the position in the list where the item of interest is stored. Regarding the SUBSTRING construct, the number inside the square brackets needs to be a single integer number for the construct to have sense, however, extracting a range of elements (that are always stored in a list) from the returned list is also possible, since this will be the primary method to select how many data one wants to retrieve at the end of the main query. For a complete description about the usage of the square brackets to extract items from a list, refer to the SQUARE BRACKETS paragraph.

A practical usage example of the SUBSTRING keyword, can be to express the P-Rule that assigns 1 to all the items in the first page of the document. Consider the following scenario:

```
context :: "SUMMARY ABRAXANE 5 mg/ml \p Hi"
main-list :: ["SUMMARY", "ABRAXANE", "5", "mg", "/", "ml", "\p", "Hi"]
```

```
A: SUBSTRING (
    WITH 0 LEN
    TOKENIZEDBY "\p"
    WHERE
    A: POS 0
    ENDQUERY)[0]
```

In this example, in the nested query it has been created a nested main-list with two elements: the first one is the text before the \p token (which designates a page interruption in the original document) and the second, is the text after. Then, the P-Rule A: POS 0 will assign 1 to the first element of the nested main-list (that is, the text in the first page) and the whole nested query will filter the nested main-list and will return a list containing only that element that received a value of 1. Then, the [0] after ENDQUERY is expressing the fact to return the element in first position of the

returned list, thus, the text that in the original document was found to be in the first page. Please note that the WITH instruction of this nested query doesn't contain the element $this, thus, its execution is independent with respect to the items contained in main-list, and therefore will return the same result each time is executed. In this situation (and similar), the nested query is executed only once for optimization reasons.

By finishing the example, the steps required to compute the Boolean-vector for main-list will be as follows:

```
context :: "SUMMARY ABRAXANE 5 mg/ml \p Hi"
main-list :: ["SUMMARY", "ABRAXANE", "5", "mg", "/", "ml", "\p", "Hi"]
nested main-list :: ["SUMMARY ABRAXANE 5 mg/ml", "Hi"]
nested main-list - BOOL :: [1, 0]
returned nested main-list :: ["SUMMARY ABRAXANE 5 mg/ml"]
extracted-string :: "SUMMARY ABRAXANE 5 mg/ml"
main-list-BOOL :: [1, 1, 1, 1, 1, 1, 0, 0]
```

Hence, this is a valid way to check if an item in main-list is a substring of something that has been computed dynamically.

However, SUBSTRING can have a second usage, that is, to check if something is a substring of an item in main-list. To use SUBSTRING with this second usage, the first argument cannot be a nested query anymore, but rather a hard-coded string, and the second optional argument $this needs to be passed. Hence, if $this is passed as second argument of SUBSTRING and a string *s* is passed as first argument, the program will assign 1 to an item in main-list if *s* is a substring of that item, and 0 otherwise.

SQUARE BRACKETS. The square brackets are used in ProbQL to extract some data from a list. If needed, the square brackets must be positioned after the ENDQUERY keyword (and if ENDQUERY is followed by a closing round parenthesis, the square brackets must be positioned after that round parenthesis). The reason of this positioning is that when ENDQUERY is parsed by the program, it must mean that a list has been constructed and returned from either a nested query, or the normal query. Hence, if only certain elements of the just constructed list need to be returned, this can be expressed after ENDQUERY and within square brackets. There are three types of extracting options:

- Single item extraction: In this case, only an integer number corresponding to the position of the desired item in the list needs to be expressed. An example to extract the item in first position would be: ENDQUERY)[0].
- Item range extraction: If a user wants to extract a consecutive range of items from a list, the syntax to perform this task is the following: ENDQUERY)[int1-int2] where int1 is the starting position of the defined range (included) and int2 is the ending position of the defined range (excluded). Hence, to extract the first 4 items from a list, a syntactically valid statement would be: ENDQUERY)[0-4].
- Multiple item extraction: If a user wants to extract multiple non-consecutive items from a list, they must specify the positions of all the desired items, each separated by a comma. Thus, to extract the items in positions 0, 2, 5 and 6 a syntactically valid statement would be:

ENDQUERY)[0, 2, 5, 6]. Please note that a space character between a comma and the next position can be added (or removed) depending on the user preferences.

4.2.5 Syntax: Additional-Context definition

At this point of the query, there are two tasks that still have to be performed: define a probability function that will be used to score each item in main-list using their previously computed Boolean-vectors, and sort main-list based on the probability score of each item.

DEFINE_PROBABILITY. The DEFINE_PROBABILITY construct is used to let the user define their own probability function. Indeed, this construct takes one argument, that is, the probability function used to give each item in main-list a probability score of being the correct metadata to extract. Basically, this function gives a weight to each defined P-Rule, in order to have better control over the importance of each P-Rule to identify the correct metadata. The syntax to express the probability function is as follows:

$$\texttt{DEFINE\_PROBABILITY} \sum_{n \in RULES} w_n label_n$$

Hence, the probability function is basically a weighted sum function over the Boolean-vectors of each P-Rule. Thus, for each P-Rule defined in the P-Rules definition part, a weight (float number between 0 and 1) has to be decided and added as a multiplicative factor in front of the label name of that P-Rule, using the dot (and not the comma) as decimal separator. The label name of a P-Rule must be a single alphabetic letter (from A to Z). Then, each element x of main-list has already been given its own value (1 or 0) for each P-Rule, and that value is multiplied by the weight factor of that P-Rule. Finally, all the results for x are added together and this final value corresponds to the probability of x to be the correct desired metadata. The sum of all the weights w_n needs to be equal to 1. In this way, an item that satisfies all the rules will be given a probability value of 100%.

To better understand this concept, consider the following example.

```
context :: "SUMMARY ABRAXANE 5 mg/ml \p Hi"
main-list :: ["SUMMARY", "ABRAXANE", "5", "mg", "/", "ml", "\p", "Hi"]
```

After some P-Rules (labeled for example from A to F) are defined to extract the medicine brand name metadata (ABRAXANE in this example), suppose that the following are the Boolean-vectors that has been given to each item in main-list:

	A	B	C	D	E	F
"SUMMARY"	1	0	0	0	1	0
"ABRAXANE"	1	1	0	1	1	1
"5"	1	0	1	0	0	0
"mg"	1	0	1	0	0	0
"/"	1	1	1	1	0	0
"ml"	1	0	1	0	0	0
"\p"	0	0	0	1	0	0
"Hi"	0	0	1	0	0	0

And suppose the following is the probability function definition:

DEFINE_PROBABILITY 0.1A + 0.15B + 0.1C + 0.3D + 0.15E + 0.2F

In this example, P-Rule D has been given a weight of 0.3, expressing that the reward given to an item that satisfies that P-Rule should be three times more important than the reward given to an item that satisfies P-Rule A, having a weight of only 0.1. Again, it should be noticed that if an item satisfies all the P-Rules, the total reward of that item is 1, expressing the 100% probability of that item being the correct metadata to extract.

By finishing the example, given the probability function, it is now possible to multiply each value by the weight of the P-Rule, and give a score (column TOT) to each item in main-list:

	A	B	C	D	E	F	TOT
"SUMMARY"	0.1	0	0	0	0.15	0	0.25
"ABRAXANE"	0.1	0.15	0	0.3	0.15	0.2	0.9
"5"	0.1	0	0.1	0	0	0	0.2
"mg"	0.1	0	0.1	0	0	0	0.2
"/"	0.1	0.15	0.1	0.3	0	0	0
"ml"	0.1	0	0.1	0	0	0	0.2
"\p"	0	0	0	0.3	0	0	0.3
"Hi"	0	0	0.1	0	0	0	0.1

In this example, the word “ABRAXANE” has been given a score of 0.9 (the highest in the example), representing the 90% probability of being the correct metadata. In case multiple items have the same maximum probability scores, when main-list will be sorted (usually in descending order of probabilities) using the SORT construct, all these items will be ordered by default in order of first appearance in main-list. In this case, the user can decide to extract multiple items (using the square brackets notation) and post processing the metadata, filtering out the incorrect ones. Or, a better tuning of the weights used in the DEFINE_PROBABILITY construct can help to disambiguate this scenario.

Also, please note that in the above example, the slash punctuation - even though it scored some points given the P-Rules it satisfied - its final probability has been forced to be 0, because it can never be the correct metadata to extract from a text. Moreover, not only the slash, but a user must be aware that the final probability of the following punctuation symbols: ( ) [ ] { } ? ! € £ $ % / , ; : " # . - + * ^ = or of a bigram or a trigram that contains one of these symbols, will always be forced to 0 for the same reason. Finally, also when an item i_1 is a substring of an item i_2, and $probability(i_1) = probability(i_2)$, only the maximal item will be given a probability score, meaning that the probability score of i_1 will be forced to 0, because the assumption that the more information a data contains and the better it is has been made.

SORT. The last task to be performed within the Additional-Context definition part is sorting main-list based on the probability scores that have been given to each item. To perform the sorting, the SORT construct is used. This construct takes only one argument that has to be passed between double quotation marks, that is, the sorting method. Two are the possible sorting methods: descending or ascending. As a convention, the standard sorting method is considered to be descending and the syntax is as follows:

```
SORT "descending"
```

After sorting in descending order, in first position of main-list there will be the item with the highest chances of being the correct metadata. However, it is also possible to sort in ascending order of probabilities, so that the item with the highest chances of being the correct metadata will be in last position of main-list.

4.2.6 Embedded Lists

In this language there are also three embedded pre-computed lists that a user can utilize in the P-Rules. Because of their nature, these lists can only be used in a P-Rule containing the `IN` construct. These lists are called: `_FREQLIST`, `_BOLDLIST` and `_CAPSLOCKEDLIST`.

`_BOLDLIST` is a list that contains all the bold elements in the PDF, regardless of their position in the document. Hence, this list can be used to express a P-Rule to find a metadata that is usually in bold (or is usually not in bold, if `NOT IN _BOLDLIST` is used instead) somewhere in the original PDF.

`_CAPSLOCKEDLIST` is a list that contains all the upper-case elements in the PDF, regardless of their position in the document. The concept is the same as `_BOLDLIST`, but in this case, a P-Rule can express the fact that the correct metadata is usually (or not usually, as above) written in upper-case in the PDF document.

`_FREQLIST` is a list containing each item of main-list that has been created during the very first `TOKENIZEDBY` instruction, together with their frequency in the original PDF document. In order to use `_FREQLIST`, the threshold frequency parameter (an integer value) has to be decided by the user based on its use case and passed as input. Hence, this list can be used to express a P-Rule that will return 1 if an item has been mentioned at least times throughout the entire PDF document, and 0 otherwise. As before, this behavior can be reverted if the `NOT` construct is used before the `IN _FREQLIST` statement.

4.3 Implementation

The implementation of `ProbQL` has been coded in Python. Hence, Python acts as an interpreter for `ProbQL`, and its job is parsing a `ProbQL` query line by line, translating each instruction in a Python instruction, that will match the proper Python function. The system is composed by two main modules: the PDF to `txt` Converter Module and the Data Extractor Module.

A representation that summarizes the whole system's architecture can be found in Figure 2. In this figure, the only inputs required by the user (PDF file and `txt` file containing the `ProbQL` query) are colored in red, the output file (JSON) is colored in orange, while the entry point is the cmd execution block. Moreover, it can be noted that for two blocks (namely, "CHECKS" and "Methods definition and execution") the input and output arrows may have different colors. This is because for these two blocks a reading rule applies, that is, it is possible to exit that block only using an arrow of the same color than the arrow used to enter. Hence, for example, if a green arrow is used to enter the "Methods definition and execution" block, only green arrows must be used to exit. Please note that this rule is only valid for the "CHECKS" and "Methods definition and execution" blocks.

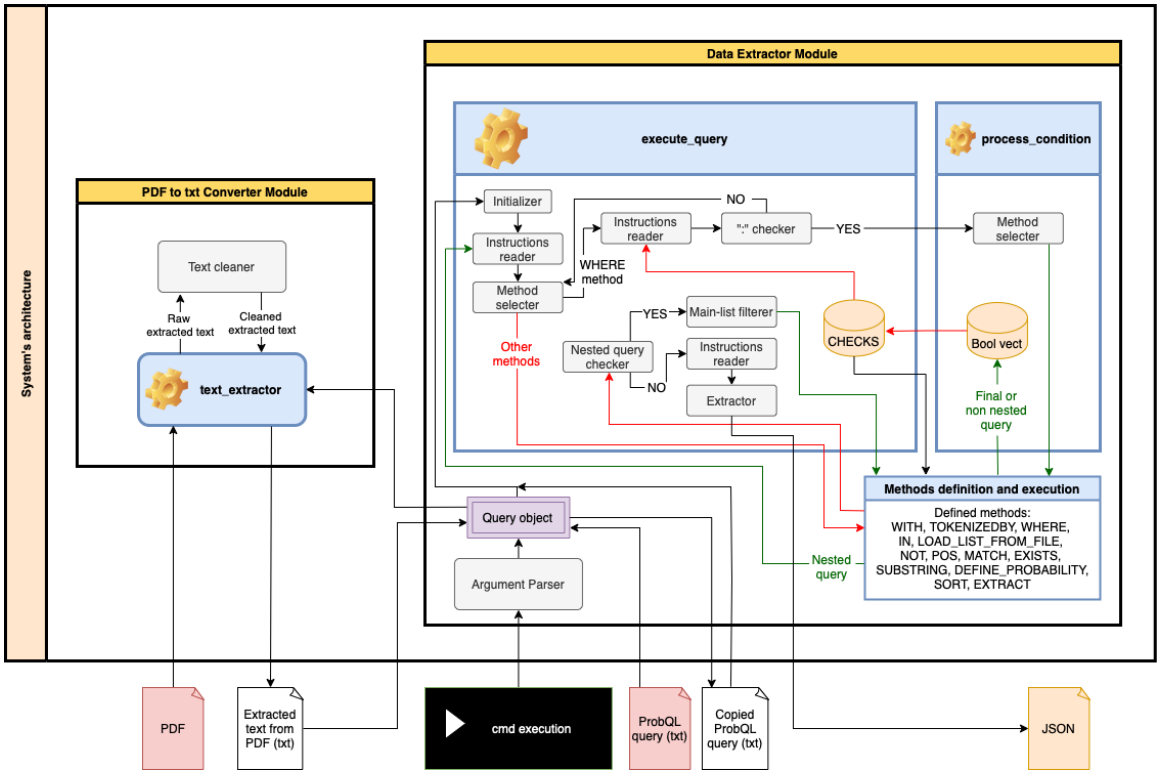

Figure 2: *Architecture of the developed system.*

4.3.1 PDF to txt Converter Module

This module - that has been named `pdf2txt` - is responsible to extract the text written in a PDF document and save it as a string into a `txt` document. As discussed in Section 2.4, extracting the text from a PDF file with high consistency across all document types is a difficult task. For this reasons, a number of automatic and optimized tools appeared in the literature, but all of them are using some kinds of heuristics - based on visual and text recognition algorithms - in order to produce the best guess of the text contained in the considered PDF, and extracting it in the correct reading order by trying to correctly identifying the logical text blocks. These are however heuristics: the output `txt` file may contain small text errors (e.g. wrong positioning of words, or wrong placement of logical text blocks) due to weird layouts of the original PDF document.

For the reasons explained in Section 2.4, the PDFplumber library [64] for Python has been chosen as the core system for the `pdf2txt` module. The main function of this module that will extract the text from the PDF is called `text_extractor`, and it requires one mandatory argument - that is, the path on the machine of the PDF file - to properly work. Moreover, 5 optional arguments can be passed:

- `text_file_path`: the path where the output `txt` file has to be saved, including the name that the `txt` file should have. By default, this path is equal to the PDF file path. In this case, the `txt` file will be named in the same way as the PDF file.
- `clean_txt`: a Boolean argument that decides whether a cleaning on the extracted text has to be performed or not. By default, its value is False, meaning that no cleaning is performed.

- `detect_bold_items`: a Boolean argument that decides whether during the text extraction, the bold items in the original PDF have to be detected and saved in a list, that will be returned at the end of the algorithm. This list is actually a list of lists: the inner lists are reflecting the pages of the document. Hence, the first list is containing all the bold items in the first page of the PDF, the second list is containing all the bold items in the second page, and so on. By default, this argument has a False value.

- `add_endpage_markers`: a Boolean argument that decides whether to add hard-coded markers at the end of each extracted page. If True, when the text of a page has been extracted and stored in a string, at the end of the string will be added the marker " . \p . ".

  (that is: space point space backslash p space point space).

  This specific marker has been chosen for its optimal performance during the tokenization (using the `TOKENIZEDBY` construct) to produce sentences, because adding a dot before and after the actual marker (`\p`) ensures that it will not be wrongly included as part of a sentence.

- `return_num_pages`: a Boolean argument that decides whether the total number of pages that the PDF is composed by has to be returned or not. By default, its value is False.

Once the settings of the `text_extractor` function are completed, the actual extracting algorithm starts. The `open` function of the PDFplumber library will open the PDF file stored at the input path, and will produce a `pdf` object. This object has a `pages` argument that contains a list of `page` objects. By iterating on this list, is possible to call the `extract_text` method on each `page` object, which returns the text of that page as a string. Then, if `clean_txt` is True, a cleaning is performed on this string. The cleaning consists of 6 steps:

1. Remove any space, `\n` and `\t` elements from the start and end of the string.

2. If the last element of the string is a number (or is in the format of number/number) and if it is equal to the actual page number (individuated by the PDFplumber library), then remove it.

3. Perform again the stripping as described in Point 1.

4. To all the following punctuation symbols: ( ) [ ] { } ? ! € £ $ % / . , ; : " “ ” # a space is added before and after. The reason of this step, is to separate words and punctuation symbols with a space character, in order to produce a cleaned and better version of main-list if the `TOKENIZEDBY` construct is producing words. For example, the word “Hello.” will become “Hello .”. In this way, several benefits are obtained, e.g., the frequency in `_FREQLIST` of the word “Hello” will be correct.

5. At this point, the string can have a random number of spaces between words (either produced by the previous step, or produced by errors in the text conversion by PDFplumber). In this step, if multiple spaces are found in the string between each word w_1 and w_2, these spaces are eliminated until only one remains.

6. In an attempt to preserve in the output `txt` file the structure of the PDF, PDFplumber is sometimes adding a number of `\n` between words. Hence, if a number $x > 1$ of `\n` are found in the string between a word w_1 and w_2, a number $x - 1$ of `\n` are removed, while the x-th

one is transformed in a dot. However, if $x = 1$, that one \n is directly transformed in a space character. Hence, after steps 5 and 6, it is certain that between each word w_1 and w_2 (where w_1 and w_2 can be punctuation symbols) there could be one and only one space.

After the cleaning, if `add_endpage_markers` is True, a " . \p . " marker is added at the end of the string.

Then, if `detect_bold_items` is True, a filtered version of the page object is produced, where all the `char` elements containing either `Bold` or `F1` in their fontname are kept. Then, the `extract_text` method called on the filtered version of the page will produce a string in which all the consecutive bold elements are separated by a \n marker. Hence, by splitting on this marker, a correct list of consecutive bold elements for the considered page is produced.

Finally, a `txt` file with Linux permissions 777 (anyone can read, write, or execute) is created, and the final extracted string will be written on it, encoded with UTF-8 encoding. The `text_extractor` function will then return the complete path where the `txt` has been saved (if needed), or - if only the `detect_bold_items` argument is True - a tuple containing the complete `txt` file path as first element, and the detected bold list as second element, or - if only `return_num_pages` is True - a tuple containing the complete `txt` file path as first element, and the total number of pages as second element, or - if both `detect_bold_items` and `return_num_pages` are True - a triple containing the complete `txt` file path as first element, the detected bold list as second element, and the total number of pages as third element.

4.3.2 Data Extractor Module

The Data Extractor Module (`extractor.py`) is the core system of the `ProbQL` language, because it contains all the functions that are responsible for the mapping from a `ProbQL` construct to a Python construct, together with the logic that determines the correct workflow of the program. All the aforementioned functions (methods) and all the attributes are organized in the implementation in four main objects: `Query`, `Keyword`, `DefaultLists` and `Item`.

Logic - Argument Parsing and Instructions Reading

When the program is executed (via command-line), several positional and optional arguments can be passed as input, which determines the desired settings. Two of these arguments are mandatory: the path pointing to a `txt` file containing the query written in `ProbQL` to execute, and the path pointing to a PDF file where the query has to be executed. For a complete description of all the possible arguments and settings, refer to Section 4.3.3.

The parsing of the arguments to internally set the correct configurations is happening through the `argparse` library for Python, which creates an `ArgumentParser` object, where it is possible to specify custom arguments. Then, through the `parse_args` method, the `ArgumentParser` is returning the value of each argument - based on the user input - which can be used to internally set the right configurations.

After the parsing of the command-line arguments, a `Query` object is created and initialized. The initialization consists of 3 phases. First, the `text_extractor` function of the `pdf2txt` module - described in Section 4.3.1 - is used (with all the optional arguments set to True) to extract the text from the given PDF, and a `txt` file is created. Second, the `txt` file is read, and its text is stored in

the Query object attribute called context, and the txt file is deleted from the machine. Third, a copy of the txt file containing the query to execute is created with Linux permissions 777, and it will be named in the exact same way as the original file, with the adding of "-copy.txt". The reason of this copy is that the program is executing the query line by line, and when an instruction has been executed, it will be deleted from the copied file. In this way, the first line will - almost - always be the line to execute at any moment. The only scenario where this is not the case, is when an instruction inside a nested query containing $this in the WITH statement has to be executed. To handle this situation, the program can read the copied query file with two modes: RO (Read Only) and RD (Read and Delete). When the program reads the copied file, it stores each line in a list called instructions. The RD mode is used by default every time, but not when an instruction is part of a nested query having $this in the WITH statement, where the RO mode is used instead. This is because these types of nested queries have to be executed exactly n times, where n is the length of main-list. Hence, the first $n - 1$ times the nested query is executed, the instructions are read using the RO mode, while the n-th time, the instructions are read in RD mode, in order to delete the nested query lines the last time it has to be executed, so to be ready to execute a new instruction the next time the copied file is read. The nested query is composed by at least 2 lines (WITH and TOKENIZEDBY statements), hence, when reading in RO mode, a counter (starting from 0) is used to keep trace which line has to be executed. Each time a line is read and executed, the counter increments by 1, in order to execute the next line of the nested query, when the next instruction has to be performed. Hence - in RO mode - from the instructions list, the line that will be executed will be the line in position equal to the counter value, while in RD mode, the line that will be executed will always be the line in position 0. Finally, when the read instruction contains the ENDQUERY keyword, the counter is set back to its initial value of 0, in order to restart the nested query execution if the next time the copied file is once again read in RO mode.

In both modes (RO and RD), all lines that starts with an hashtag symbol (which are considered to be comments) and all lines that only contain a \n symbol are ignored, and another line is read until it doesn't satisfies these two conditions. Then, the instruction is cleaned by removing all the space characters, \n and \t symbols from the start and the end of the instruction, in order to remove the indentation (if used) and the newline character at the of the line.

Logic - Keywords Mapping and Workflow

Once the Query object has been initialized, the execute_query method of the Query object is called. The pseudo-code of this method is described in Algorithm 1. This method is one of the two main methods that are responsible for the correct workflow of the system and the mapping of the keywords, together with the process_condition method. The execute_query and the process_condition methods will be recursively called throughout the entire program workflow, and their tasks are as follows.

For the execute_query method, it accepts as input the query_path (which will be the copied query path), the pdf_path, an elem element (necessary for the implementation of the WITH construct) that by default is null, two flag variables subquery and last_subquery_execution (which clarify whether the execute_query method has been called from a nested query or not) that are False by default and a tokenize_counter (necessary for the implementation of the TOKENIZEDBY construct) that is 0 by default.

The first task of execute_query is checking whether it is called from a nested query, and if it is not the case, then the initialization previously described is actually performed at this time. Then, a Keyword object is created, which will be responsible for the mapping from the ProbQL instruction to the corresponding Python function that will actually perform the instruction, and an hash-map CHECKS is initialized as an empty hash-map. This hash-map will contain key-value pairs, that store the Boolean vectors for the P-Rules as values, and the labels of the P-Rules as keys.

Then, an instruction is read from the copied query file (in RD or RO mode as previously described, and as always from this point ongoing), and if that instruction is not a WITH statement, an error is raised. Subsequently, another line is read from the copied query file, and if that instruction is not a TOKENIZEDBY statement, an error is raised. Once these two instructions are collected, the corresponding methods (WITH and TOKENIZEDBY) contained in the Keyword objects are called, using as input the respective collected instructions. The list that has been referred to as main-list thus far, is exactly the list produced the first time that this TOKENIZEDBY instruction is performed. For the sake of clarity, the implementation of all the methods contained in the Keyword object will be explained further on. Then, another line *x* is read from the copied query file. This line can either contain another instruction, or a ENDQUERY keyword. Hence, while ENDQUERY is not in the line *x*, there are only 3 possible instructions: DEFINE_PROBABILITY, SORT or WHERE. The line *x* is split using the space character as a token, and the element in first position of the produced list will always be the name of the instruction that has to be performed. If the instruction is a DEFINE_PROBABILITY instruction - or a SORT instruction - then the DEFINE_PROBABILITY method - or SORT method, accordingly - of the Keyword object is called and executed with line *x* as input, then, another line is read and stored as line *x* and the while condition on line *x* is checked. If the instruction is instead a WHERE instruction, then another line is immediately read from the copied query file, and will be referred to as line *y*. According to the defined syntax, this line will contain a P-Rule that has to be executed. Hence, since there could be multiple P-Rules to execute, while line *y* will contain a colon punctuation mark (necessary to define a P-Rule), the WHERE method of the Keyword object is called and executed using as input line *y*. The WHERE method will eventually update the hash-map CHECKS that has been created thus far, by adding a new pair key-value having as key the label of the P-Rule that has been executed, and as value a list having 1s and 0s as elements, that corresponds to the Boolean vector produced by that P-Rule for the items contained in main-list. Then, another line is read and stored as line *y*. If that line *y* doesn't contain a colon punctuation mark, the while statement becomes False, and line *x* is set to be equal to line *y*. Subsequently, the while statement of line *x* is checked, and if line *x* contains an ENDQUERY instruction, it will become False.

At this point, if execute_query has been called from a nested query (even if it has been called from a last execution of a nested query), according to the semantics of the language, main-list has to be filtered based on the P-Rules that each item satisfied. This information is stored in the CHECKS hash-map. The values of this hash-map are all Boolean vectors, hence, if put together they will form a table where each column represents an item in main-list, and each row represents each P-Rule. Thus, only the items having an entire column of 1s are kept in main-list, while if there exists even only one 0 in that column, the corresponding item in main-list will be removed.

Then, the program will check the presence of an opening and closing square brackets in line *x*, and if the result is True, the EXTRACT method of the Keyword object is called and executed, giving as input line *x*.

Finally, if execute_query has not been called from a nested query, it means that the whole extraction process is completed, thus, main-list is returned and the copied query file is deleted from the machine.

Algorithm 1 execute_query

```
1: if not subquery and not last_subquery_execution then
2:    initialize()
3: end if
4: method Keyword(); CHECKS {}; with_instruction readline()
5: tokenizedby_instruction readline()
6: sliced method:WITH()
7: main-list method:TOKENIZEDBY()
8: line_x readline()
9: while "ENDQUERY" not in line_x do
10:   keyword line_x:split()[0]
11:   if keyword == "DEFINE_PROBABILITY" then
12:      main-list method:DEFINE_PROBABILITY()
13:      line_x readline()
14:   end if
15:   if keyword == "SORT" then
16:      main-list method:SORT()
17:      line_x readline()
18:   end if
19:   if keyword == "WHERE" then
20:      line_y readline()
21:      while ":" in line_y do
22:         CHECKS method:WHERE()
23:         line_y readline()
24:      end while
25:      line_x = line_y
26:   end if
27: end while
28: if subquery or last_subquery_execution then
29:   main-list filter_list()
30: end if
31: if "[" in line_x and "]" in line_x and "ENDQUERY" in line_x then
32:   main-list method:EXTRACT()
33: end if
34: if not subquery and not last_subquery_execution then
35:   remove_copied_query_file()
36: end if
37: return main-list
```

Once the execution of execute_query is completed, the Data Extractor Module receives the data

returned by that method, and based on the passed arguments at execution time, the module can manipulate the data based on the given directions. In order to understand all the possible options, refer to Section 4.3.3.

The other method responsible for the correct mapping of the keywords and the workflow of the system, is the `process_condition` method. The `process_condition` method takes as input a line *y* that contains an instruction where a P-Rule is defined, the main-list, the `copied_query_path` and the `context` attribute of the `Query` object. The purpose of this method is to process the different P-Rules that have been defined in the query, and therefore is called at need from the methods contained within the `Keyword` object.

As soon as it is called, the `process_condition` method creates a `Keyword` object and splits line *y* based on the space character to obtain a list of arguments. If the square brackets are not found in line *y* (meaning that a hard-coded list is not present in the given instruction), then the splitting happens with the regular Python `split` function, otherwise, the `split` function is called only on the instruction until the first square bracket is met, and the hard-coded list is then manually added to the just created list of arguments, considering that list to be the portion of text from the first square bracket ongoing. The first element of the arguments list is the keyword that corresponds to the instruction to perform. In this method, the only valid keywords that can be used are only the keywords `IN`, `SUBSTRING`, `POS`, `NOT`, `EXISTS` and `MATCH`, that are the keywords described in the P-Rules definition part. Therefore, what this method does is basically calling the correct method of the `Keyword` object, based on which keyword has been found as the first argument in the arguments list. Then, the appropriate method of the `Keyword` object will compute a Boolean vector based on the defined P-Rule and will return that vector to the `process_condition` method that in turn, will return the same Boolean vector at the end of its execution. Hence, the `process_condition` method is basically a method to disambiguate and map the correct Python function to process the defined P-Rule.

The Item Object

An `Item` object is created every time a `TOKENIZEDBY` instruction is performed. When the sliced text (after the `WITH` statement) is split based on some token, every created piece of the text becomes an `Item` object, and it will have 5 attributes:

1. `text`: a string corresponding to the actual text of the split element. This attribute is required to create an `Item` object.

2. `start_pos`: the starting position (position of first character of `text`) in the `context` attribute of the `Query` object. The computation of the starting position will be described in the `TOKENIZEDBY` construct implementation description. This attribute is required to create an `Item` object.

3. `length`: the length of the string contained in the `text` attribute. This attribute is computed automatically, therefore, is not required when an `Item` object is created.

4. `end_pos`: the ending position (position of last character of `text`) in the `context` attribute of the `Query` object. This attribute can be specified at creation time of an `Item` object (in

order to correctly deal with bigrams and trigrams) or - if not specified - it will be computed automatically as `start_pos + length - 1`.

5. `probability`: the probability value that the string contained in `text` is the correct metadata to extract. The `probability` attribute is initialized at 0, and it will be updated when the `DEFINE_PROBABILITY` construct is called.

Hence, main-list is actually not composed by strings, but rather by multiple `Item` objects.

The Keyword Object

The `Keyword` object is composed by 13 methods, one for each possible construct - `WITH`, `TOKENIZEDBY`, `WHERE`, `IN`, `LOAD_LIST_FROM_FILE`, `NOT`, `POS`, `MATCH`, `EXISTS`, `SUBSTRING`, `DEFINE_PROBABILITY`, `SORT` - and one to handle the square brackets (`EXTRACT`). As previously discussed, this object is created in both the `execute_query` and `process_condition` methods, in order to map the correct keyword read from the `ProbQL` instruction with the Python function responsible to execute it, that is defined exactly as a method of the `Keyword` object.

WITH. The `WITH` method takes 4 input arguments: the `instruction`, an `elem` parameter, the `context` attribute of the `Query` object and the `Query` object itself. The `elem` argument can either be `null`, or it can be an `Item` object. Though, this second scenario is only possible when the `WITH` method has been called as part of a nested query having `$this` in it, and the `elem` argument is exactly the `Item` object identified by `$this`. The algorithm works as follows. First, it splits the instruction based on the space character in order to create a list with three elements: the `WITH` keyword in first place, the starting position `sp` in second place and the ending position `ep` in third place, used to slice the `context` from `sp` (included) to `ep` (excluded). Second, it checks whether `sp` is an integer number: if that is the case, the checks on `ep` will start, otherwise, it checks if `$this` is a substring of `sp`. If this is not the case, an error is raised, otherwise, if `START_POS` is a substring of `sp`, `sp` will considered to be the `start_pos` attribute of the `elem` argument, else, if `END_POS` is a substring of `sp`, `sp` will considered to be the `end_pos` attribute - incremented by 1 - of `elem`. If none of this are True, an error is raised.

Then, similar checks are applied over `ep`. First, if `ep` already is an integer number, the checks are finished. Otherwise, if `ep` is equal to `LEN`, then `ep` is considered to be the length of `context`, else, if `START_POS` is a substring of `ep`, then `ep` is considered to be the `start_pos` attribute - decremented by 1 - of `elem`, else, if `END_POS` is a substring of `ep`, then `ep` is considered to be the `end_pos` attribute - incremented by 1 - of `elem`. Even here, if none of this are True, an error is raised.

Finally, a sliced version of `context` from the position `sp` (included) to the position `ep` (excluded) is returned.

TOKENIZEDBY. Given a token (or multiple tokens separated by the keyword `AND`), this construct will split the sliced string returned from a `WITH` statement every time a token is found, and each separated piece will become an `Item` object (thus having all the `Item` object attributes), that will be inserted in a list. Hence, for each token found in the `TOKENIZEDBY` statement (by splitting the instruction using `AND` as a token) a splitting algorithm that is also saving the position of the text

of the created Item objects in the original string has to be executed, and all the produced lists by the splitting algorithm are then merged together to obtain only one final list.

The first time the TOKENIZEDBY instruction is performed, the created list will be main-list, and it will correctly be composed by all the items having the text as specified by the splitting token(s), and the position referring to the context attribute of the Query object.

The pseudo-code of the splitting algorithm can be found in Algorithm 2 and it works as follows. As input, it requires the string to be split and the token to be used for splitting the given string. First, if the given token is either \bigrams or \trigrams, then a flag variable is set to be True, the original token is stored in another variable and the actual token is transformed to be a space character, so to obtain a list of words. This is because from the list of words, is very easy to construct a list of bigrams or trigrams. Second, if the given token is not found in the given string, then an Item object having the entire string as text attribute and 0 as starting position is created, inserted in an empty list and returned by the splitting algorithm. Otherwise, the core execution of the splitting algorithm starts by saving the length in characters *l* of the given token, initializing a start variable at 0 and creating an empty list *L*. Then, while the token is found to be in the string to split, a sequence of instructions are performed. First, the Python find method is used on that string to get the starting position *p* of the first time the token appears in the string (scanning from left to right). Second, a version of the chopped string from position 0 to position *p* (excluded) is saved into a text variable. Third, a left_stripped variable is created, that is, text having all the space characters and dots removed from its left side, and a completely_stripped variable is also created, that is the left_stripped variable having all the space characters removed from its right side. The reason to have two separate variables is that when computing the position of text in the original input string, only the starting position matters to create an Item object (the ending position can be automatically computed), and to update the starting position (because on text could have been removed some characters) the computation has to be performed only on the left_stripped version of text. In fact, if completely_stripped is not an empty string, the number of removed elements *n* is computed by calculating the difference between the length of text and the length of the left_stripped variable. Then, an Item object is created having as text completely_stripped and as starting position the current value of start plus *n*, and is appended in *L*. After this, the start variable is incremented by the token position *p* plus the length of the token *l* and the original string is chopped from position *p* + *l* ongoing. Once the while instruction has finished (hence, the token is no more found to be in the string), if the string is still not empty, a final Item object is created having as text the remaining part of the string, and as starting position the current value of the start variable, and is appended in *L*.

In a normal case, *L* is now complete and it will be returned from the splitting algorithm. However, if the flag variable is True (hence the actual token was either \bigrams or \trigrams) *L* is now a list of Item objects having words as their text attributes, and further steps are needed to construct the list of Item objects having bigrams or trigrams as their text attributes. Hence, if the original token was \bigrams, then for each Item object in *L*, a variable w1 is set to be equal to the text of the current Item object and a variable w2 is set to be equal to the text of the next Item object respect to the current one. Then, by concatenating w1 and w2 - with a space character that separates the two - it is possible to create another Item object having as text attribute exactly this concatenation, as starting position the starting position attribute of the current Item object and as ending position the

ending position attribute of the next `Item` object (the one having `w2` as text). Thus, this new `Item` object is added to a different list than *L*, that will eventually be returned by the splitting algorithm once each `Item` object in *L* (though not the last one) have been processed. This algorithm has been named in Algorithm 2 as `compose_bigrams`. If otherwise the original token was `\trigrams`, a similar version of the just explained algorithm is used, and the small difference is that a variable `w3` is also needed, that is the text contained in the `Item` object after 2 `Item` objects than the current one. Then, a space character is also added between `w2` and `w3`, and the final `Item` object containing the trigram as text attribute will have as ending position the ending position of the `Item` object having `w3` as text attribute, and it will be added in a different list rather than *L*. As before, once each `Item` object in *L* (excluded the last two) have been processed, that different list will be returned by the splitting algorithm. This algorithm has been named `compose_trigrams` in Algorithm 2.

Algorithm 2 split

```
1: if token == "\bigrams" or token == "\trigrams" then
2:     flag     True; original     token; token     " "
3: end if
4: if token not in string then
5:     item     Item(text     string, start_pos     0)
6:     return  [item]
7: end if
8: l     len(token); start     0; L     [ ]
9: while token in string do
10:     p     string:find(token)
11:     text     string[0 : p]
12:     left_stripped     text:strip(space; dot)
13:     completely_stripped     text:strip(space)
14:     if completely_stripped is not empty then
15:         n     len(text)     len(left_stripped)
16:         item     Item(text     text, start_pos     start + n)
17:         L:append(item)
18:     end if
19:     start     start + p + l
20:     string     string[(p + l) :]
21: end while
22: if string is not empty then
23:     item     Item(text     string, start_pos     start)
24:     L:append(item)
25: end if
26: if flag == True then
27:     based on the value of original do
28:     L     compose_bigrams(L) or L     compose_trigrams(L)
29: end if
30: return  L
```

However, as it currently is, this solution would have been perfect if implemented in a faster programming language than Python. Indeed, throughout all the popular programming languages, Python is known to be on the slower side in terms of performances. In fact, Python is a dynamically typed language: there is no need to declare the type of the various objects, and objects can change type during runtime. Python uses a concept called Duck Typing, synthesized in the phrase: "If it walks like a duck and it quacks like a duck, then it must be a duck". Using Duck Typing, Python does not check types at all. Instead, Python checks for the presence of a given method or attribute. Though, the benefit of this choice is that by emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Although this choice confers the flexibility that made Python so popular, it is also the reason why it is so slow compared to best performances languages such as C or C++, since extra work is required to infer the type of an object, each time that object is used. Hence, the same operation that in C++ requires less than a second, could require minutes in Python. However Python was a necessary choice for the long-term perspective of the `ProbQL` language, because in the future, `ProbQL` - which now it's a completely Rule-Based approach - can be integrated with different Machine Learning approaches in order to improve its extraction accuracy, by combining the best techniques of both worlds, as will be further discussed in Section 6.2. Since Python is commonly the language of choice when it comes to implement a Machine Learning approach - due to the great library ecosystem it has for this scope - the implementation of `ProbQL` is justified to be done in Python. Hence, this means that some kind of optimizations are needed for making the `TOKENIZEDBY` construct runs faster, due to the fact that the split algorithm has a linear time complexity (which becomes quadratic, if `\bigrams` or `\trigrams` are used as tokens) and the problems arise when the `TOKENIZEDBY` operation is performed in a nested query having `$this` in its `WITH` statement. In fact, let's consider an example where main-list has been created by tokenizing using the space character, building a list composed by 3904 items, and the following nested query is executed:

```
A: EXISTS (
    WITH $this.END_POS LEN
    TOKENIZEDBY " "
    WHERE
    A: MATCH "mg"
    B: POS 1
    ENDQUERY)
```

In this example, the nested query has to be performed 3904 times (because the token is also the space character), meaning that even the `TOKENIZEDBY` operation (together with the others) is performed 3904 times. Each time is performed, the tokenization will create a list with one less item than the iteration before, meaning that the 3904th time the `TOKENIZEDBY` operation is performed, the produced list will contain only 1 item. Figure 3 shows an analysis that highlights the time required for each iteration for each keyword, and the experiment was performed on a MacBook Pro having a 3.1 GHz Intel Core i5 dual-core processor and 8GB RAM.

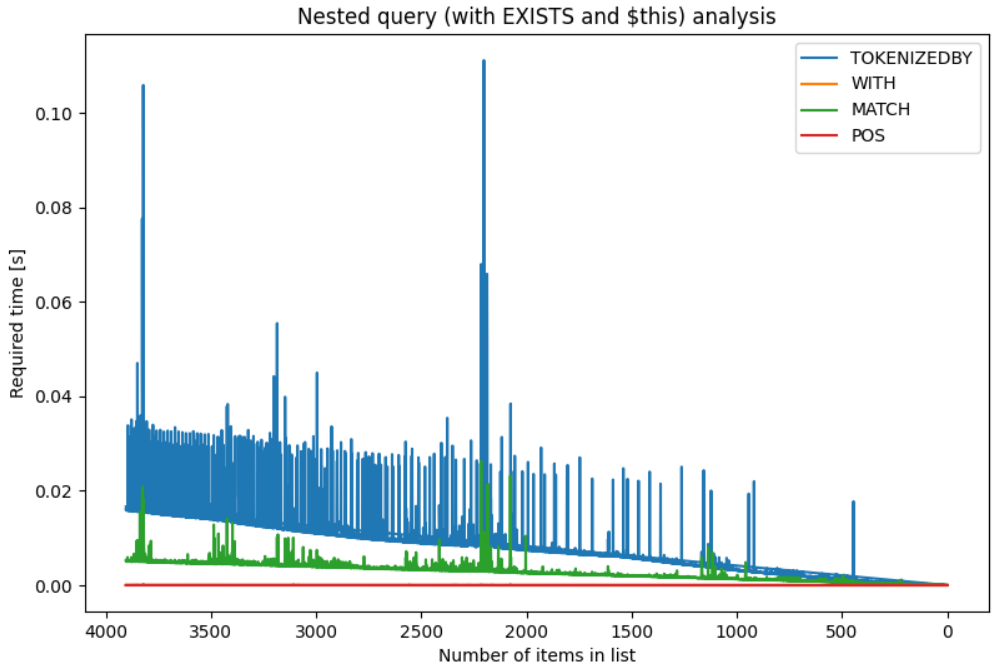

Figure 3: *Required time for each iteration as a function of the number of items in the created list, before optimization.*

As expected, the time required is decreasing with the number of items to analyze for each keyword. While `POS` and `WITH` - which are overlapping - clearly have a constant time complexity (and they required overall 0.04 and 0.06 seconds respectively) `MATCH` and `TOKENIZEDBY` are indeed linear, and took overall for their execution 21.5 and 62.8 seconds respectively. However, in practice, `TOKENIZEDBY` has both its baseline and its peak fluctuations much higher than `MATCH`. Considering that `TOKENIZEDBY` potentially has much room for optimization, this construct has been identified as a bottleneck of this system.

The optimized solution that has been devised is to pre-compute (while the very first main `TOKENIZEDBY` instruction is performed) the most common lists that a user may want to use. Five common lists have been individuated that are thought to be common and useful in every domain for a metadata extraction process, and they are the following lists: words, sentences, bigrams, trigrams and pages. By using this solution, every time a `TOKENIZEDBY` instruction has to be performed, if the given token(s) is either a space character (for words), a dot (for sentences), `\bigrams`, `\trigrams` or `\p` (for pages), the pre-computed lists are retrieved instead of being calculated at running time. Then, if the `TOKENIZEDBY` instruction - using one or more aforementioned tokens - is executed inside a nested query having `$this` in its `WITH` clause, rather than computing each time the list, the `WITH` instruction is read to understand the behavior that the `TOKENIZEDBY` instruction should have, and a sliced version of the correct pre-computed list is returned each time. Moreover, this optimization can be enabled/disabled at will when the execution of the program starts, as will be explained in Section 4.3.3.

The pre-computed lists are stored in the `DefaultLists` object, together with other embedded

lists. This object has 5 attributes and 5 methods. The 5 attributes are the following:

- `BOLDLIST`: a list that stores all the bold elements found in the PDF. By default is empty. Its computation is described in Section 4.3.1 and its usage in a `ProbQL` query is described in Section 4.2.6.
- `FREQLIST`: a hash-map whose keys are the texts of the `Item` objects created during the tokenization, and whose values are the number of times each object appears in the given PDF. By default is empty. Its usage in a `ProbQL` query is described in Section 4.2.6
- `CAPSLOCKEDLIST`: a list that stores all the upper-case elements of the given PDF. By default is empty. Its usage in a `ProbQL` query is described in Section 4.2.6
- `COMMONLISTS`: a hash-map that will have 5 keys (words, bigrams, trigrams, sentences and pages) whose values are the corresponding pre-computed lists. By default is empty.
- `available_tokens`: a list that contains all the 5 available tokens (space character, `\bigrams`, `\trigrams`, a dot and `\p`) that can be used to create a pre-computed list.

The first method of the `DefaultLists` object is computing `FREQLIST` and accepts main-list as input. Basically, for each item (its text) in main-list, its frequency in main-list is computed by using the Python function `count`. Then, a pair key-value having as key the text of the considered `Item` object, and as value its frequency in main-list is created and added to the `FREQLIST` hash-map.

The second method is computing `CAPSLOCKEDLIST` and accepts main-list as input. Basically, for each `Item` object in main-list, if its text is all in upper case (checked using the Python function `isupper`) and if it's not yet in `CAPSLOCKEDLIST`, then append its text in it.

The third method is computing the common lists, and requires the `context` attribute of the `Query` object as input. To do that, the split function described in Algorithm 2 is called 5 times to create the 5 aforementioned common lists, giving as input the correct token to produce the desired list each time, and the complete string containing the converted text (`context` attribute of the `Query` object). Then, a pair key-value is created having as key the name of the list (either words, bigrams, trigrams, sentences or pages) and as value the corresponding created list, and is added in the `COMMONLISTS` hash-map.

The fourth method accepts a token as input, and returns True if the given token is in the `available_tokens` list and False otherwise.

Finally, the fifth method accepts a token as input, and based on the given token, if it is one contained in the `available_tokens` list, it will return the correct pre-computed list from the `COMMONLISTS` hash-map.

Therefore, the complete algorithm for the `TOKENIZEDBY` construct is as follows. It requires 8 inputs: the instruction to perform containing the `TOKENIZEDBY` construct, the output string from the `WITH` construct, two Boolean variables `subquery` and `last_subquery_execution` that are True if the `TOKENIZEDBY` method has been called from a nested query, the `DefaultLists` object, the instruction containing the `WITH` statement that directly precedes the considered `TOKENIZEDBY` instruction, the `Query` object and a `counter` variable that by default has a value of 0. First, an empty list L that will contain the final result is created. Second, if both `subquery` and `last_subquery_execution` are False, it means that for this query, the `TOKENIZEDBY` method is called for the first time, thus, the

computation of the common lists is executed by calling the third method of the DefaultLists object. Then, a list *T* that contains each token found in the instruction is created by splitting a chopped version of the given instruction - from the first double quotation mark (included) ongoing - using the keyword AND as splitting token. After this, each token in *T* is first stripped by each starting and ending space characters and then further stripped by starting and ending double quotation marks, in order to have a cleaned version of the tokens. Then, for each token *t* in *T*, the following routine starts. The fourth method of the DefaultLists object is called giving *t* as input, and if it returns True, and also the fast method is enabled (by checking the fast_method attribute of the Query object, that is True if the program has been executed in this way) the fast method algorithm starts, otherwise, the split algorithm (Algorithm 2) is called using *t* as input. The fast method algorithm will retrieve the correct list *C* from the pre-computed lists using the fifth method of the DefaultLists object and giving *t* as input. Then, the arguments of the WITH instruction are isolated in order to obtain sp and ep (as explained in the WITH description), in order to understand if *C* has to be returned as it is, or if it has to be chopped because the current iteration of the TOKENIZEDBY instruction is part of a nested query having $this in its WITH statement. Hence, if at least one between subquery or last_subquery_execution is True and the keyword $this is in sp or ep, three cases are considered:

- If sp is equal to $this.START_POS and ep is either LEN or contains only digits, then only the items in *C* from position counter (included) ongoing are kept. counter is an input variable that keep trace how many times in a nested query, the TOKENIZEDBY instruction has been performed. Hence, if the complete pre-computed list of words *C* is for example [“Hello”, “my”, “name”, “is”, “Daniele”], the first time the TOKENIZEDBY instruction is called, counter will have a value of 0 by default, and the entire list is returned. However, the second time, counter will have a value of 1, meaning that the list [“my”, “name”, “is”, “Daniele”] is returned instead, which is correct considering the case we are analyzing.

- If sp is equal to $this.END_POS and ep is either LEN or contains only digits, then only the items in *C* from position counter + 1 (included) ongoing are kept. The reasoning is similar as before, however this time, the first element is not to be kept because $this.END_POS was used in sp.

- If ep is equal to $this.START_POS and sp contains only digits, then only the items of *C* from position sp (included) to position counter (excluded) are kept.

- If ep is equal to $this.END_POS and sp contains only digits, then only the items of *C* from position sp (included) to position counter + 1 (excluded) are kept. In this case, if the complete pre-computed list of words *C* is for example the same as before [“Hello”, “my”, “name”, “is”, “Daniele”], and supposing that sp has a value of 0, the first time the TOKENIZEDBY instruction is called, counter will have a value of 0 by default, thus, a list with only the item containing the text “Hello” is returned, which is correct because now ep is also considering the END_POS position. However, the second time is performed, counter will have a value of 1, meaning that the list [“Hello”, “my”] is returned, which is correct considering the case we are analyzing.

The resulting list is then appended to the list *L*.

After each token in T has been processed, all the created lists in L are merged together in the order they have been created and stored (hence, from the left-most token to the right-most one).

Finally, if both `subquery` and `last_subquery_execution` are False, then the computation of `FREQLIST` and `CAPSLOCKEDLIST` starts by calling respectively the first and second methods of the `DefaultLists` object, and at the end of its execution, the `TOKENIZEDBY` method will return the merged list L.

By running the same query that produced the graph in Figure 3 under the same settings, but using the optimized solution with the computation of the common lists, the graph in Figure 4 is obtained.

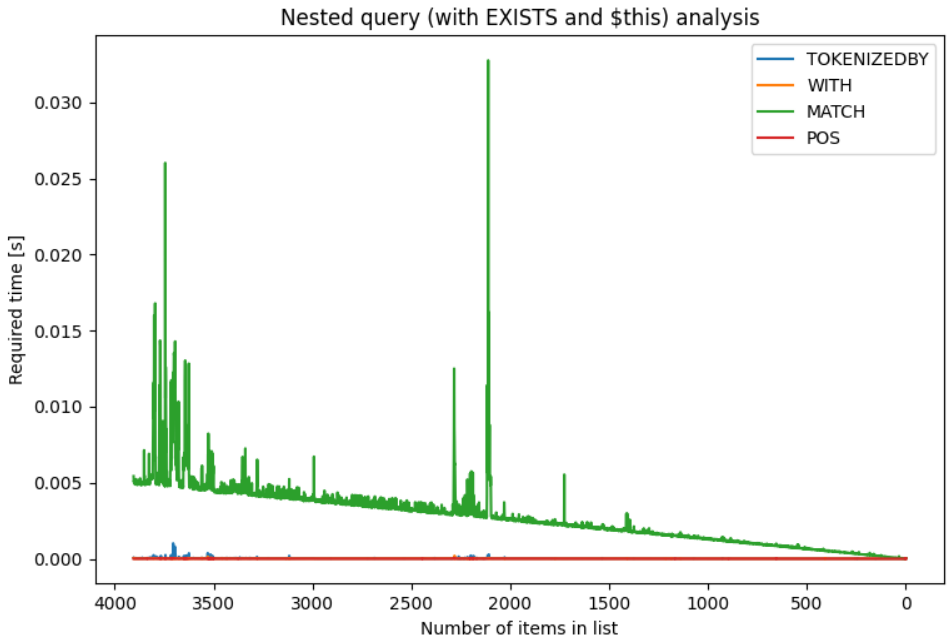

Figure 4: *Required time for each iteration as a function of the number of items in the created list, after optimization.*

From Figure 4 it can be seen that the `TOKENIZEDBY` instruction is now performed in constant time (while all the other instructions have the same time complexities as before), obtaining a huge performance increasing compare to Figure 3. In fact, while the total time across all iterations to perform the `MATCH` instruction is not changed (roughly about 21 seconds), the total time to perform the `TOKENIZEDBY` instruction now is 0.1 seconds, that if compared to the 62.8 seconds as before, the achieved improvement is huge.

However, an implicit assumption has been made in the optimized version. As previously explained, the optimized version make use of a `counter` variable in order to correctly chop the list C based on the `WITH` statement. In order for this to properly work, and to properly assign the correct Boolean values considering that we are evaluating the item `$this` and it needs to match with the correct version of the chopped pre-computed list, the returned list L (from a nested query) and main-list must have the same length, meaning that they have to be tokenized using the same tokens.

In other words, in order to safely use the optimized version of the program, all the TOKENIZEDBY instructions contained in nested queries having $this in their WITH statements must use the same tokens as specified in the TOKENIZEDBY instruction that created main-list.

WHERE. The WHERE construct takes 6 inputs: the instruction to perform - that contains a P-Rule - a list *L* of Item objects, the path of the copied query file, the context attribute of the Query object, the Query object itself and the current CHECKS hash-map. From the instruction that has to be performed, the label of the P-Rule is isolated, that is considered to be the portion of text before a double column punctuation sign. Then, the process_condition method of the Query object is called giving as input the right inputs from the WHERE construct. As previously discussed, the process_condition method will map the keyword contained in the P-Rule with the correct Python function, in order to compute a Boolean vector based on how many items in main-list are satisfying that P-Rule. Hence, as a final step of the WHERE construct, the isolated label and this Boolean vector create a pair key-value that is added to the current CHECKS hash-map. Finally, this updated hash-map is returned by the WHERE construct.

IN. The IN construct takes as input the instruction containing the P-Rule, the path of the copied query file, a list *L* of Item objects, the arguments list created in the process_condition method, the Query object and the DefaultLists object.

First, the IN construct creates an empty Boolean vector, represented by a regular Python list object. Then, if an opening round parenthesis is found in the given instruction (meaning that IN contains a nested query) and nor an opening square bracket neither the keyword LOAD_LIST_FROM_FILE are found in the given instruction, the following routine is performed. Using the RO mode, the next line in the query txt file is read. When using the RO mode, the reading happens without deleting the read instruction from the query file, hence, it's a valid way to know which instruction has to be performed next, without skipping its execution. Since the current IN instruction is containing a nested query, it means that the next instruction has to be a WITH statement. If the keyword $this is found in that WITH statement, then the nested query has to be executed multiple times, as already explained. Then, a counter variable is initialized at 0 (that will be the counter accepted by the TOKENIZEDBY method when the optimized solution is enabled) and for each position in *L*, if the considered position is different from *length*(*L*) 1, the execute_query method of the Query object is called, using the current Item object as its elem argument, True as its subquery argument value (in order to read the instructions in RO mode, since they don't have to be deleted because they will have to be performed multiple times) and the defined counter variable as its tokenize_counter argument, used during the optimized solution of the TOKENIZEDBY construct. Otherwise, it means that the algorithm is evaluating the last Item object in *L*, thus, the execute_query method of the Query object is called - always using the current Item object as its elem argument, and the counter variable as its tokenize_counter argument - but this time, True as its last_subquery_execution argument value, in order to read the instructions in RD mode.

Hence, a recursive approach is used. The execute_query method will execute the nested query (this time considering $this as the current Item object in the WITH statement) and since it's a nested query, it will return as output a filtered list of Item objects, containing only the objects that satisfied the nested query P-Rules. Since this routine - from the reading of the next line in RO

mode until this point - will be repeated also for other keywords, from now on it will be referred to as Routine *R*.

Continuing with the `IN` keyword algorithm, if the text of the considered `Item` object is present in the returned list, a 1 is appended in the Boolean vector, otherwise, a 0 is appended. Finally, the `counter` variable is incremented by 1.

However, if `$this` was not present in the `WITH` statement, the `execute_query` method of the `Query` object is called only once - since the nested query has to be performed only once - giving True as its `last_subquery_execution` argument value, since the returned list still must be filtered, and the reading mode needs to be RD. Then, as before, for each `Item` object in *L*, if its text is present in the returned list, a 1 is appended in the Boolean vector, otherwise, a 0 is appended.

At this point, if the previously cited condition about the opening round parenthesis, square bracket and the keyword `LOAD_LIST_FROM_FILE` is not met, this procedure is used instead. The second element of the arguments list is read. In this case, following the grammar of the language, this second element can only be one of these 5 cases:

- `_FREQLIST`: In this case, the frequency threshold is read from the arguments list (third argument). Then, for each `Item` object in *L*, if its text is present in the list of keys that are in the `FREQLIST` hash-map of the `DefaultLists` object, and if the corresponding value is greater or equal than the frequency threshold, then a 1 is appended in the Boolean vector, otherwise, a 0 is appended.

- `_BOLDLIST`: In this case, an empty string *B* is created and filled with all the bold elements from the `BOLDLIST` list of the `DefaultLists` object. Then, for each `Item` object in *L*, if its text is in *B*, then a 1 is appended in the Boolean vector, otherwise, a 0 is appended.

- `_CAPSLOCKEDLIST`: In this case, for each `Item` object in *L*, if its text is in the `CAPSLOCKEDLIST` list of the `DefaultLists` object, then a 1 is appended in the Boolean vector, otherwise, a 0 is appended.

- `LOAD_LIST_FROM_FILE`: In this case, the path of the file containing the list to load is considered to be the third argument of the arguments list, and it is stripped from the double quotation marks. Then, the `LOAD_LIST_FROM_FILE` method of the `Keyword` object is called. As will be discussed, this method returns a Python `set` object containing all the elements that are present in the given file. After this, if the keyword `SPLIT` is not in the arguments list, then, for each `Item` object in *L*, if the lowered version of its text is in the `set` object returned from the `LOAD_LIST_FROM_FILE` method, then a 1 is appended in the Boolean vector, otherwise, a 0 is appended. The reason to lower the text before performing the lookup operation will be explained when the implementation of the `LOAD_LIST_FROM_FILE` method will be discussed. Otherwise, if `SPLIT` is in the list of arguments, it means that all the words that are composing the text of an `Item` object have to be individually present in the `set` object. Hence, for each `Item` object in *L*, its text is lowered down and split using the space character as token, to create a list of words *W* that the considered `Item` is composed by. Moreover, also an empty list *T* is created, which will act as a mini Boolean vector, only valid for the considered `Item` object. Then, for each word in *W*, if it is present in the `set` object, a 1 is added in the list *T*, otherwise, a 0 is added. Finally, if all the elements in *T* for the considered `Item` object are 1s, then a 1 is added to the actual Boolean vector, otherwise, a 0 is added.

- an opening and closing square brackets are present in the second element: In this case, it means that the user has defined a hard-coded list. Since the syntax of an hard-coded list in ProbQL is the same as the syntax of a regular Python list, the `literal_eval` function of the Python `ast` library [67] is used to convert the string representation of the list into an actual Python `list` object P. Then, for each `Item` object in L, if its text is present in P, then a 1 is added to the actual Boolean vector, otherwise, a 0 is added.

Finally, the `IN` method returns the constructed Boolean vector.

LOAD_LIST_FROM_FILE. The `LOAD_LIST_FROM_FILE` construct takes only one input value, that is, the path of the `txt` file containing the list to be loaded. Then, this file is opened, and every line is considered to be an element of the list to be returned. To each element, two cleaning operations are performed: first, every space and `\n` characters are removed from the element start and end, then, the element is transformed in its lowered version. This second operation is performed in order to make a fair comparison when an item in main-list is searched in the list loaded from the `LOAD_LIST_FROM_FILE` construct. In fact, the intended way to use this construct is to search if the "concept" of an item in main-list is in the loaded list, disregarding any upper-case letters. Hence - as explained also in the `IN` construct implementation description - before the searching happens, also the item in main-list is transformed in its lowered version.

However, a regular Python `list` is not the correct data structure to use in this case, since the lookup happens in linear time. In this case, a `set` data structure would be more appropriate, since the loaded list doesn't need to contain repetitive elements, and the average time complexity for `set` lookup operations is constant, as they are implemented as hash-tables [68]. Hence, once each element in the `txt` file has been cleaned, it is inserted in a `set` data structure, and once the reading of the `txt` file is completed, that `set` is finally returned from the `LOAD_LIST_FROM_FILE` construct.

NOT. The `NOT` construct takes as input the instruction containing a P-Rule, a list L of `Item` objects, the path of the copied query file, the `context` attribute of the `Query` object and the `Query` object itself. If the `NOT` method of the `Keyword` object has been called, by design of the system it means that it is the first keyword contained in the given instruction. Also, the `NOT` construct is certainly followed by at least another keyword, since its whole purpose is to negate a certain condition expressed by a P-Rule. Hence, it is sufficient to isolate the part of the instruction that follows the `NOT` construct, and by negating the Boolean vector that is returned by calling the `process_condition` method of the `Query` object with the isolated instruction as input, a valid Boolean vector is defined for the `NOT` construct. Thus, after the `process_condition` method has returned a Boolean vector for the instruction that follows the `NOT` construct, every 1 in that vector is transformed to be a 0, and vice versa. Finally, the transformed Boolean vector is returned by the `NOT` keyword.

POS. The `POS` construct takes as input a list L of `Item` objects, and the entire list of arguments that has been created in the `process_condition` method by splitting the instruction line on the space character. The argument in first position will be the integer that specifies the position x that the item in L has to be assigned a Boolean value of 1. Hence, a Boolean vector containing all 0s of the same size as L is created. Subsequently, if $x < 0$ and $x \quad size(L)$ or $x > 0$ and $x < size(L)$,

then the element in the created Boolean vector in position *x* will be assigned a value of 1. Finally, the Boolean vector is returned by the POS construct.

MATCH. The MATCH construct takes as input a list *L* of Item objects, and the entire list of arguments that has been created in the process_condition method by splitting the instruction line on the space character. Then, an empty Boolean vector is created. In order to match a certain regular expression, the re library for Python [69] has been used, which is a standard library that contains interesting built-in functions to work with regular expressions. In fact, the actual algorithm of the MATCH construct works as follows. First, the compile function of the re library is used on the argument in first position of the input list of arguments - which is the user defined string in ProbQL containing the regular expression to match - to create a Pattern object. This intermediate operation before the actual matching is necessary for e□ciency reasons, because when a single regular expression is reused throughout the program (as is the case for the MATCH construct) the match is optimized if the regular expression is contained into a Pattern object, rather then using a string containing it. Then, the search function from the re library is applied for each item in *L*, which takes as input the text of the Item object in *L* and the Pattern object to match. The search function returns a Match object if the text of the Item object matches the given Pattern object, and returns None otherwise. Hence, if a Match object is returned, then a 1 is added to the Boolean vector, otherwise, a 0 is added. Please note that since this operation is performed for each item in *L*, the 0s and 1s are correctly inserted into the Boolean vector so to correspond to the correct positions of the evaluated items. Finally, the MATCH construct is returning the Boolean vector.

EXISTS. The EXISTS construct takes as input the instruction line containing the P-Rule, the path of the copied query file, a list *L* of Item objects and the Query object. First, the EXISTS construct creates an empty Boolean vector, represented by a regular Python list object. Then, if the given instruction contains an opening round parenthesis (meaning that a nested query will next be performed), Routine *R* - defined during the explanation of the IN construct implementation - is used to obtain a filtered list of Item objects that satisfied the P-Rules defined in the nested query. Then, if the length of the returned list is greater than 0, it means that the searched element actually exists in the searched place (e.g. after or before the considered Item object), hence, a 1 is appended in the Boolean vector, otherwise, a 0 is appended. Then, the counter defined in Routine *R* is incremented by 1. Because of its definition and purpose, the EXISTS construct must always contain a nested query having $this in its WITH statement, hence, if a round parenthesis was not found in the given instruction, an error is raised.

Finally, the EXISTS construct returns the created Boolean vector.

SUBSTRING. The SUBSTRING construct takes as input the instruction line containing the P-Rule, the path of the copied query file, a list *L* of Item objects and the Query object. First, the SUBSTRING construct creates an empty Boolean vector, represented by a regular Python list object. Then, if the given instruction contains an opening round parenthesis (meaning that a nested query will next be performed) and the keyword $this is not in the instruction, Routine *R* - defined during the explanation of the IN construct implementation - is used to obtain a filtered list of Item objects that satisfied the P-Rules defined in the nested query. However, the nested query (if any) of the SUBSTRING construct must always end with a 0 inside the square brackets after the ENDQUERY

keyword, in order to extract the first (and only) `Item` object of the filtered list, where its text will constitutes the created string. After this, if the text of the considered `Item` object is in the created string, then a 1 is appended in the Boolean vector, otherwise, a 0 is appended. Then, the counter defined in Routine *R* is incremented by 1. However, if `$this` was not present in the `WITH` statement, the `execute_query` method of the `Query` object is called only once - since the nested query has to be performed only once - giving True as its `last_subquery_execution` argument value. Then, for each `Item` object in *L*, if its text is present in the returned string, a 1 is appended in the Boolean vector, otherwise, a 0 is appended.

The `SUBSTRING` construct has also to deal with the case where a string is hard-coded in the `ProbQL` query. In this scenario, two situations have to be tackled (and every other case will raise an error):

- Items in *L* are substrings of a given string: This case is intercepted by checking the presence of the double quotation marks in the given instruction and the absence of the keyword `$this`. Thus, in this scenario, the hard-coded string is isolated, and it is considered to be the portion of text from the first double quotation mark ongoing. Subsequently, this string is stripped by the double quotation marks, to create a final string *S*. Then, for each `Item` object in *L*, if its text is present in *S*, a 1 is appended in the Boolean vector, otherwise, a 0 is appended.
- The given string is a substring of the items in *L*: This case is intercepted by checking the presence of the double quotation marks in the given instruction and the also the presence of the keyword `$this`. Also in this scenario, the hard-coded string *S* is isolated, and it's considered to be the portion of text from the first double quotation mark (excluded) until the last double quotation mark (excluded). Then, for each `Item` object in *L*, if *S* is present in its text, a 1 is appended in the Boolean vector, otherwise, a 0 is appended.

Finally, the constructed Boolean vector is returned by the `SUBSTRING` construct.

EXTRACT. The `EXTRACT` construct takes as input the instruction line, a list *L* of `Item` objects and the `Query` object. First, it isolates the position(s) of the item(s) that a user want to extract from *L*, which is considered to be the portion of text of the given instruction within square brackets. Second, if the isolated position contains only digits (hence a number *x*), then the item in position *x* of *L* is returned. Else, if a hyphen sign (-) is contained in the isolated position, it means that a range of items has to be extracted and returned from *L*. In this case, the position is split using the hyphen sign as token to create a list containing two elements: the starting position `sp` and the ending position `ep`. Hence, the items from `sp` (included) to `ep` (excluded) are extracted from *L* and returned. Otherwise, if a comma sign is found to be in the isolated position, then the isolated position is split using the comma as token to create a list of positions *P*. Hence, a new empty list *E* is created, and for each position *y* in *P*, the element in position *y* in *L* is inserted in *E*. Finally, *E* is returned. If none of the previous are True, an error is raised.

DEFINE_PROBABILITY. The `DEFINE_PROBABILITY` construct takes as input the instruction line, the main-list, the `Query` object and the `CHECKS` hash-map containing key-value pairs, with the computed Boolean vectors as values and the labels of the P-Rules that generated those Boolean

vector as keys. First, an empty hash-map *H* is created. The purpose of this hash-map is to contain key-value pairs, where the keys are the labels of each defined P-Rule, and the values are the corresponding weights that the user defined in the DEFINE_PROBABILITY construct of the ProbQL query. In fact, the probability function is isolated (by slicing the instruction line from the first space - excluded - ongoing) and is parsed in the following way. For each character, if the character is a dot or a digit, a string `weight` variable is incremented with the character. Otherwise, if the character is an alphabetic letter, a key-value pair corresponding to the current character as key, and the floating point version of the `weight` variable found by far as value is added in the hash-map *H*. Then, the `weight` variable is set to be empty, and the process starts again until each character in the isolated probability function has been processed. Subsequently, all the values in the hash-map *H* are summed up, and if the sum is not equal to 1, an error is raised, since the sum of all weights must adds up to 1. Then, the actual update of the probability scores of the `Item` objects contained in main-list happens in the following way. For each `index` (corresponding to a position) in main-list, and for each label in the list of keys of *H*, a `word-list` variable is computed, that is a list containing the text of the item corresponding to the position `index` in main-list that has been split using the space character as token. Then, if the length of `word-list` is less or equal than 3 (True only for words, bigrams and trigrams) and if any of the elements in `word-list` is one of the following punctuation signs: ( ) [ ] { } ? ! € £ $ % / , ; : " # . - + * ^ = then the probability score of the item in position `index` of main-list will not be updated. Otherwise, the probability score will be incremented by the value in *H* corresponding to the current label key (thus the weight of the considered P-Rule) multiplied by the Boolean score that the current items received for the current label, that is, the score in position `index` of the Boolean vector that is stored as a value inside the CHECKS hash-map corresponding to the current label key.

Once the probability scores of each item have been computed, for every pair of items in main-list, if they have the same probability score, but the text of one is a substring of the other, the probability of the minimal item is forced to be 0, keeping only the maximal one. Finally, the updated main-list is returned from the DEFINE_PROBABILITY construct.

SORT. The SORT construct takes as input the instruction line, a list *L* of `Item` objects and the `Query` object. First, it detects the sorting method (descending or ascending) by isolating everything that comes after a space character in the given instruction. Second, the Python `sort` method is called on *L*, by passing as its `key` attribute a lambda function that specifies to sort on the `probability` attribute of the `Item` objects contained in *L* and - if the sorting method is `descending` - the `reverse` attribute of `sort` will be given a value of True, meaning that *L* has to be sorted in descending order. Third, if *n* items in *L* have the same text, *n* 1 items are removed from *L*, keeping only the item with the highest probability attribute. This last operation is necessary because it can be possible that multiple items in *L* have the same text but di erent probabilities (due to their positioning in the original PDF, which directly reflects on how many P-Rules they satisfy), and it can happen that in the retrieved result, these items are possibly taking up for example, the first, second and third position of the sorted version of *L*, which logically is redundant. Hence, by removing those items that cannot be the correct metadata to extract since there exists another item with the same text, but a higher probability, this problem is avoided. Finally, the sorted (and filtered) version of *L* is returned by the SORT construct.

4.3.3 System Usage and Output

Once downloaded, this system requires Python (>3.8) in order to be used. Then, the creation of a virtual environment (venv) is necessary in order to correctly install the utilized libraries. Hence - on Unix-like systems - the following commands should create and activate a virtual environment (run these commands in the folder of the application):

```
>>> pip3 install virtualenv
>>> virtualenv venv
>>> source venv/bin/activate
```

Then, with the following command, all the necessary libraries will be installed in the created virtual environment:

```
>>> pip3 install -r requirements.txt
```

Once installed, this system can be executed via command-line. The main file to run is called `extractor.py`, and it requires 2 positional arguments, namely, the path of the PDF file to analyze (argument `-p`) and the path of the `ProbQL` query (argument `-q`). Hence, a basic execution of the system can start with the following command:

```
>>> python3 extractor.py -p ./pdf_path.pdf -q ./query_path.txt
```

By default, the Data Extractor Module will save the output data (contained into main-list returned by the `execute_query` method of the `Query` object) into a JSON file by using the `dump` method of the Python `json` library [70], and by indenting using 4 spaces. The created file will be saved in the same folder as the `extractor.py` file. However, the output folder path where the JSON file has to be saved can also be specified using the argument `-o`. The JSON file will have the following structure: first, an object named as the path of the PDF where the data have been extracted from, is created. This object has an array of other objects as its value, where each object corresponds to a piece of data that have been extracted from the considered PDF document. Each object in the array have 8 key-value pairs:

- `text`: The text corresponding to the extracted data. This is equal to the `text` attribute of the `Item` object.
- `starting_position`: The starting position with respect to the string created from the extracted PDF text. This is equal to the `start_pos` attribute of the `Item` object.
- `length`: Length of the text contained in the `text` attribute. This is equal to the `length` attribute of the `Item` object.
- `ending_position`: The ending position with respect to the string created from the extracted PDF text. This is equal to the `end_pos` attribute of the `Item` object.
- `probability`: The probability score of being the correct data. This is equal to the `probability` attribute of the `Item` object.
- `extraction_time`: Total time required to extract the data from the considered PDF document.

- num_checked_items: Number of candidate items of being the correct data to extract. This number is basically the length of main-list.

- pdf_pages: The number of pages that the PDF document used for the extraction task is composed by.

The objects are already sorted in decreasing order of probabilities. An example of a JSON file that has been produced by executing a query that extracts the two piece of data with the highest probability from a PDF file located at "./sample_data/abraxane-HAS.pdf" is as follows:


```
{
    "./sample_data/abraxane-HAS.pdf": [
        {
            "text": "ABRAXANE",
            "starting_position": 111,
            "length": 8,
            "ending_position": 118,
            "probability": 1.0,
            "extraction_time": 113.909,
            "num_checked_items": 7811,
            "pdf_pages": 10
        },
        {
            "text": "ABRAXANE 5",
            "starting_position": 111,
            "length": 10,
            "ending_position": 120,
            "probability": 0.65,
            "extraction_time": 113.909,
            "num_checked_items": 7811,
            "pdf_pages": 10
        }
    ]
}
```


As it is visible, this structure allows the union of multiple JSON files created by the extraction of different PDF documents, as the main keys are the paths of those PDFs. Since a single execution of the program to extract data from a PDF will produce a single JSON file, a user may want to combine all the produced JSON files into a single file, and this structure has been chosen in order to easily allow this operation.

Furthermore, there are other optional arguments that can be passed when executing the program, in order to better define the extraction settings. Table 1 provides a complete description of all the arguments that the system accepts.

Argument	Long name	Argument description
-h	–help	Show the help message and print all the arguments with their descriptions
-p	–pdf	Path of the PDF to analyze and extract the data from (required)
-q	–query	Path of the query that has to be executed (required)
-o	–output	Path of the output folder that will contain the extracted data (JSON file)
-l	–logs	Print the query logs (time required for executing each instruction)
-c	–check	Print a table items / boolean to see the boolean values given by each P-Rule to each item
-d	–display	Print the extracted data together with their probabilities
-u	–unsave	Don't save the JSON file containing the extracted data at the end of query execution
-s	–slow	Execute the query without pre-computation of common lists (disable fast method)

Table 1: *Arguments that can be specified via command-line execution of the program.*

Among all the arguments, only -p, -q and -o have to be followed by the correct desired path. All the other arguments can be used as they are, without inserting other optional information. Hence, an example of the execution that will provide a pdf path, a query path, an output folder, and that disables the optimized method, prints the obtained results as well as the query logs can be as follows

```
>>> python3 extractor.py -p ./pdf_path.pdf -q ./query_path.txt -o
./output_folder/ -s -d -l
```

Please note that by default the program will use the fast optimized method to extract the data, but when a `ProbQL` query contains multiple `TOKENIZEDBY` instructions, if at least one instruction is contained in a nested query that is using `$this` in its `WITH` statement and has a different token compared with the main `TOKENIZEDBY` instruction, the assumption behind the fast optimized method is not valid anymore, thus, it needs to be disabled by passing the argument `-s`.

4.3.4 System's Workflow

Now that the system has been described in its entirety, a reader should have a clear understanding of its working. Thus, Figure 5 summarizes the full workflow of the system. Also, as it was for Figure 2, even for Figure 5 the only inputs required by the user (PDF file and `txt` file containing the `ProbQL` query) are colored in red, the output file (JSON) is colored in orange, while the entry point is the cmd execution block.

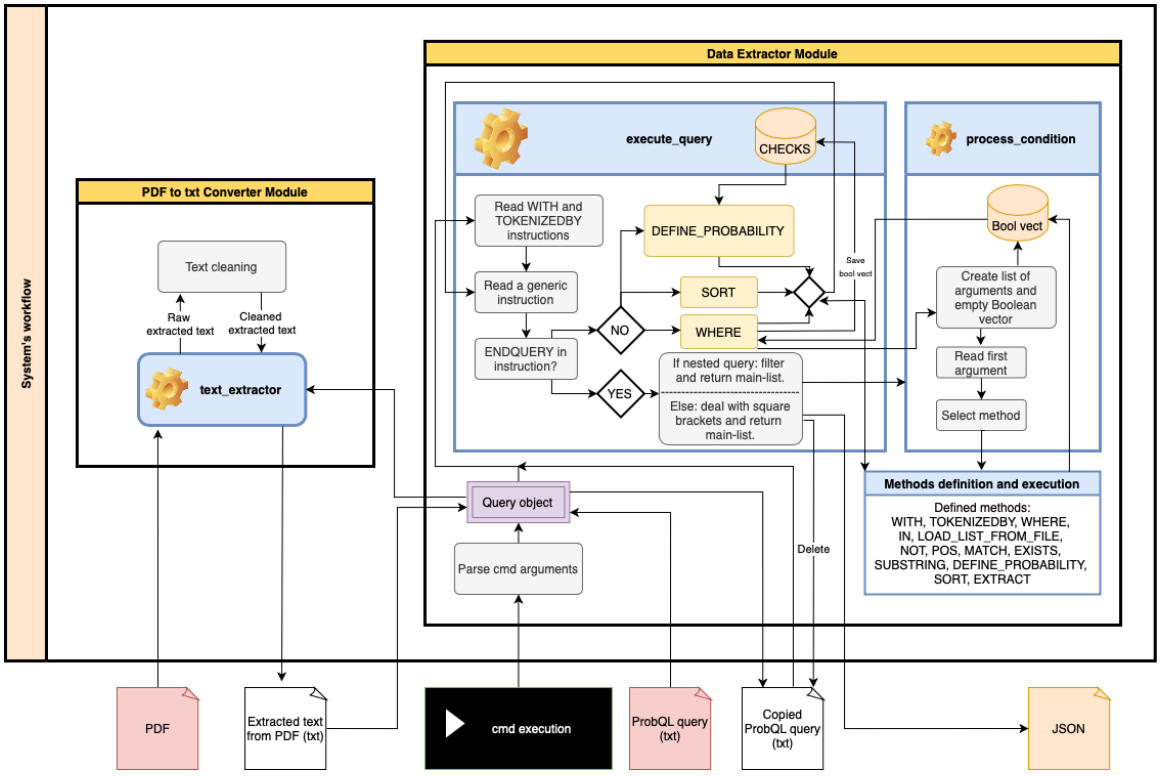

Figure 5: *Workflow of the developed system.*

4.3.5 Error Handling

A basic error handling system for ProbQL queries has also been implemented. When certain types of syntax errors are made by users writing the query, at execution time some kind of errors are raised, with the scope of automatically guiding the user by pointing out where the error has been made, so that the revising and correction phases can happen in a faster way. Whenever errors are raised, the execution of the program immediately stops, and the copied query file actually used to execute the query is deleted from the machine.

Every type of error has been given a code, from 1 to 10. The following list, maps the code error with its message. When *something* is used, the portion of the line that caused the error will be printed in the real program instead.

- Code 1: *something* is not a valid ProbQL keyword
- Code 2: Wrong syntax between square brackets: *something*
- Code 3: Wrong syntax in the SUBSTRING statement: *something*
- Code 4: After a WITH statement, every query must have a TOKENIZEDBY statement, but *something* was used instead
- Code 5: Every query should start with a WITH statement, but *something* was used instead
- Code 6: Wrong syntax in the EXISTS statement: *something*

- Code 7: The sum of the weights defined in `DEFINE_PROBABILITY` must be 1, not *something*
- Code 8: Wrong syntax in the `WITH` statement: *something*
- Code 9: Provide an output path to save the json file containing the data that will be extracted (use: `-o` or `–output`) (only if `-o` path has
  been passed empty and `-u` is not used)
- Code 10: Sorting method *something* is not defined. Possible sorting methods: `descending`, `ascending`

These errors are organized in the code in a `Error` object. This object only has one method called `throw_error`, which takes as input the code, the instruction that caused the error and the `Query` object. Based on the given code, this method will raise the proper error message. Hence, based on which if-then-else condition is not satisfied during the execution of an instruction, the `throw_error` method will be called using the correct code. Then, a user can refer to the language documentation described in Section 4.2 in order to understand how to properly solve the error.

4.4 Pharmaceutical Department Pipeline

The presented system is able to automatically extract some metadata from a PDF report, based on some specifications encoded in the so called P-Rules. Hence, it can be used as the core system to solve the problem explained in Chapter 3, regarding the automatic metadata extraction from medicine reimbursement reports. As previously mentioned, in order to properly work, this system requires standard and basic inputs (a `txt` file containing the `ProbQL` query and a PDF file) and produces a standard output (a JSON file containing the extracted data), thus, it's very easy to incorporate in a broader application, in order to completely automatize the medicine report analysis task. Indeed, as discussed in Section 1.1, an IE system is only an intermediate step included in broader applications. In this case, the presented IE system will be the core system that will extract the requested data from a PDF document, and the broader application in which can be integrated could work as follows. First, all the `txt` files containing the `ProbQL` queries that have to be executed on a single PDF document are stored in a local folder called F_1. Second, a seed file containing all the interesting websites to observe (e.g. HTA organizations websites) is provided to a web crawler application. Once the web crawler recognizes that a new PDF file is uploaded in a monitored section of the websites contained in the given seed file, it proceeds to download that file and save it into a local folder F_2. In the meantime, a Linux script that is always running, is monitoring that folder (or a sub-folder, if for each drug a specific folder is created) and once a new PDF file is added, it will iterate over the queries contained in F_1. For each iteration, the Linux script will activate and run the IE system presented in this work, giving as input the just added PDF file and one query file per iteration. Before executing the system, an output folder F_3 can also be specified, in order to have more control over the JSON files containing the extracted data that will be produced. Then, the JSON files can be re-organized at will, they can either be included in a relational database, in a graph database or even opened by applications like Excel or Access. In this way, if some queries are taking some time to extract the data, if everything is automated as explained, the extraction

velocity problem can be mitigated, since no inputs are required by a human user, meaning that the time of no one will be wasted waiting for the query to finish extracting the data.

Moreover, the Data Extractor Module is not the only way to start the execution. This module is in fact a wrapper around the actual model, since its actions are: parsing the arguments given by the command-line execution, creating the `Query` object, calling the `execute_query` method of the `Query` object and finally producing the JSON file (or other outputs, if requested through the command-line execution). However, if the final user has a little programming experience, they can create their own personalized wrapper that will perform these steps, and manipulate the data returned by the `execute_query` method directly in Python. In both cases (using a JSON file, or directly the Python list containing the extracted data), if post-processing operations are applied (e.g. sentence and word cleaning, standardize the format of the output data, studies of the distribution of probabilities of being the correct metadata, etc.) the quality of the retrieved data can further improve.

5 Experiments in the Pharmaceutical Sector

The system has been tested on the task of extracting 5 types of metadata from the medicine reimbursement reports compiled by HTA organizations. For each medicine, a PDF public report is compiled and published on the considered HTA's website. Medicines are sometimes reassessed by HTA organizations because new evidence is available or because the medicine will be used in a new indication. To test our system, 106 initial assessment reports published on the HAS website (the French HTA organization) have been considered. For reproducibility reasons, the list of the medicines that these reports are referring to can be found in Appendix B.

All the reports are written in English (since the proposed queries have been designed having the English language as a reference) and for each medicine (hence for each report) the 5 metadata to extract are the following:

- Medicine brand name: The name given to the medicine by the pharmaceutical company that markets it.
- Medicine generic name: The "official" name of the medicine, based on its active ingredient.
- Indication: The use of a medicine for the treatment of a particular disease.
- Reimbursement class: The reimbursement class in which the medicine has been classified.
- Assessment date: The date on which the medicine has finished of being assessed and therefore the PDF report has been produced.

To accomplish this task, 5 different queries have been written, one for each data to extract. Then, the extracted data have been compared to a golden standard (ground truth) that the Pharmaceutical Department of the Utrecht University has manually extracted and collected, in order to compute, for each query, the appropriate performance metrics. In the IE field, four are the most common metrics, namely: precision, recall, accuracy and F1-score. These metrics are commonly defined as follows:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN} \tag{3}$$

$$F1 \quad score = 2 \quad \frac{Precision \quad Recall}{Precision + Recall} \tag{4}$$

Where TP is the number of documents whose data have been correctly extracted, FN is the number of documents having data to extract, but nothing was extracted, FP is the number of documents where incorrect data have been extracted and TN is the number of documents that do not contain the data to extract, and nothing was indeed extracted [71]. However, due to the nature of our task, all the documents contain all the 5 data to extract (meaning that TN will always be 0), plus, the system will always extract some data from the documents, meaning that FN will always be 0 as well. This results in the recall of our system always being 100% for every extraction task - which doesn't provide any useful information - and in turn, implies that the F1-score will always be the same as the precision metric. Additionally, since TN and FN are both 0, the precision metric is the same as the accuracy metric. Hence, the only metric of interest for the evaluation of this system is the accuracy, that can be expressed - for each data to extract - as the fraction of documents whose data have been correctly extracted over all the documents.

Together with the golden standard, the Pharmaceutical Department has also provided some domain knowledge (which will be shared in each section describing the considered metadata extraction) that they are currently using to identify the correct data to extract from a document. This knowledge, in addition to a detailed analysis of some possible document layouts, have been encoded in the P-Rules of the queries.

All the experiments have been performed on a MacBook Pro having a 3.1 GHz Intel Core i5 dual-core processor and 8GB RAM.

5.1 Medicine Brand Name

It has been noticed that the medicine brand name data is probably located at the beginning of the document (is always present on the first page) but can also be located in multiple sections. Most of the times is a single word, but can also be composed by two words. Most of the time can be found in upper-case. Most of the time can be found in bold. Sometimes is a frequently repeated word throughout the document, and is probably a non-English word. Finally, sometimes is followed by a number and the word “mg”.

Query. The query that encodes this domain knowledge in the P-Rules is as follows:

```
(WITH 0 LEN
TOKENIZEDBY "\bigrams" AND " "
WHERE
A: SUBSTRING (
     WITH 0 LEN
     TOKENIZEDBY "\p"
     WHERE
     A: POS 0
     ENDQUERY)[0]
B: IN _FREQLIST 15
```

```
C: IN _BOLDLIST
D: NOT IN LOAD_LIST_FROM_FILE "./english_dictionary.txt" SPLIT
E: IN _CAPSLOCKEDLIST
F: EXISTS (
    WITH $this.END_POS LEN
    TOKENIZEDBY " "
    WHERE
    A: MATCH "mg"
    B: POS 1
    ENDQUERY)
DEFINE_PROBABILITY 0.1A + 0.1B + 0.1C + 0.3D + 0.15E + 0.25F
SORT "descending"
ENDQUERY)[0]
```

First, the Main-list definition part starts by selecting the entire length of the converted PDF file using the `WITH` statement. Since the data can either be a single word or a bigram, the `TOKENIZEDBY` construct has been used to split the converted document to produce a list (main-list) of bigrams (first) and words (second). The bigrams' list has been produced first because in case a word *W* has received the same probability of a bigram *B* (maybe because *W* is contained in *B*), it is sure that the bigram *B* will be retrieved first, being ahead compare to the word *W*.

Then, with the `WHERE` clause, the P-Rules definition part starts. P-Rule A states that the correct data should be in the first page of the document. In fact, in the defined nested query, the entire converted document has been split based on the \p character, and in its P-Rules definition part, the item in first position (hence, all the contents in the first page) has been assigned a value of 1. Being a nested query, the produced list is filtered to keep only the items having a conjunction of values resulting in 1, hence, a list with a single element is returned by the nested query of P-Rule A. From that list, using the 0 in the square brackets, the element in first position is extracted, and with the `SUBSTRING` construct it is possible to check whether each item in main-list is a substring of the extracted item, and assigning 1 if that is the case, or 0 otherwise. P-Rule B is used to check if the considered item in main-list has been repeated at least 15 times throughout the document, where 15 has been chosen as the frequency threshold based on a manual investigation performed on the documents. P-Rule C states that the correct data should be in bold, while P-Rule D yields that the correct data should not be contained in a user-loaded list, which in this case contains all the words in the English dictionary. Since main-list also contains bigrams, the keyword `SPLIT` is also passed as argument of `LOAD_LIST_FROM_FILE`, to ensure that only a bigram whose both words are not in the English dictionary will be assigned a value of 1. P-Rule E states that the correct data should be in upper-case, and finally, P-Rule F state that the second word after the considered data should be the word "mg". To perform this check, the `WITH` clause of the nested query is selecting the portion of text from the ending position of the considered data (hence, the first element of the produced list would be the first word after the considered data) until the end of the document. Then, the document is split to obtain a list of words, and in this list, a 1 is assigned to each word that matches the word "mg" and to the word in position 1. Being a nested query, the produced list is filtered, and the only case where the returned list will not be empty, is the case where both the conditions are satisfied, namely, a word that matches the word "mg" is found in position 1 (second position).

If that is the case, a 1 is assigned to the considered data in main-list.

Then, the Additional-context definition part starts. By using the DEFINE_PROBABILITY keyword, it is possible to define the weights that allow a better control over the importance of each P-Rule. These weights have been manually assigned based on domain knowledge, and it can be seen that - for example - not being in the English dictionary is three times more important than being in the first page of the document. The best strategy (at this stage) to assign the correct weights to each P-Rule is to give a higher weight to the P-Rule that is probably better at separating the correct data from all the noise of the other incorrect data. In this way, it's easier to design a query that can be used across documents having di□erent layouts: if only in few of them the desired data is satisfying a certain property, a P-Rule that expresses that property can be expressed and assigned a low weight, so that in the majority of cases that P-Rule doesn't a□ect that much the overall probability score given to each item. However, in the documents for which that property is satisfied, that weight acts as a further discriminant to better identify the correct data to extract, increasing its probability of being retrieved. Finally, main-list is sorted in ascending order of probabilities and the 0 in the square brackets will extract the item in first position, namely, the item having the highest probability of being the correct data to extract, given the defined P-Rules.

Extraction Results. Amongst 106 total PDF documents, the medicine brand name data was correctly extracted from 90 of them, resulting in a 84.9% accuracy. The 16 medicines for which the extracted data are wrong are listed in Table 2, together with the extracted data, the probability it was assigned, the time required for the extraction (in seconds), the number of items that main-list was composed by and the number of pages that the PDF document was composed by.

Ground Truth	Extracted	Prob	Time [s]	n° items	Pages
Alunbrig	C RANSPARENCY	0.65	2.3	955	2
Besremi	C RANSPARENCY	0.65	8.9	2421	3
Cometriq	cabozantinib	0.75	7.9	2283	2
Cotellic	BRAF V600	0.65	4.9	1655	2
Iclusig	T315I	0.75	12.7	2943	2
Imfinzi	PD-L1 ,â•	0.65	9.1	2507	2
Keytruda	the pembrolizumab	0.65	7.9	2267	2
Lorviqua	ALK TKI	0.65	2.3	1055	2
Mekinist	BRAF V600	0.65	6.5	2085	2
Mektovi	The encorafenib	0.75	4.9	1701	2
Polivy	DLBCL	0.75	1651.2	29199	29
Tagrisso	EGFR T790M	0.65	5.8	1773	2
Talzenna	0.25 and	0.65	8	2279	4
Tecartus	CD3+	0.65	24.6	4111	5
Venclyxto	CLL	0.8	18.00	3517	3
Zaltrap	irinotecan	0.85	162.8	10171	14

Table 2: *Wrong results for the medicine brand name extraction.*

Hence, for all the medicines listed in Appendix B that are not in Table 2, the extraction of the medicine brand name was correctly performed. An accuracy of 84.9% is very promising, considering

that this system hasn't reached yet its maximum potential, since future works with the objective of refining its extraction accuracy are possible, as explained in Section 6.2.

5.2 Medicine Generic Name

It has been noticed that the medicine generic name data is probably located in the first page of the document, but can also be present in multiple sections. Most of the times is a single word, but sometimes can be a bigram as well. It is frequently repeated throughout the entire document, but with less frequency than the medicine brand name data. Finally, sometimes is preceded by the bigram "Active ingredient".

Query. The query that encodes this domain knowledge in the P-Rules is as follows:

```
(WITH 0 LEN
TOKENIZEDBY "\bigrams" AND " "
WHERE
A: SUBSTRING (
    WITH 0 LEN
    TOKENIZEDBY "\p"
    WHERE
    A: POS 0
    ENDQUERY)[0]
B: IN _FREQLIST 7
C: IN LOAD_LIST_FROM_FILE "./generic_names_list.txt"
D: EXISTS (
    WITH 0 $this.START_POS
    TOKENIZEDBY "\bigrams"
    WHERE
    A: MATCH "Active\singredient"
    B: POS -2
    ENDQUERY)
DEFINE_PROBABILITY 0.05A + 0.1B + 0.7C + 0.15D
SORT "descending"
ENDQUERY)[0]
```

As the query for the medicine brand name data, in the Main-list definition part, we consider the entire length of the document and tokenize it to produce a list of bigrams and words.

Then, in the P-Rules definition part, P-Rule A and P-Rule B are the same as the medicine brand name query, with the only difference that now the frequency threshold has been set to 7 rather than 15, to express that the medicine generic name data is less frequent than the medicine brand name data. Then, since the collected domain knowledge seems to be not complete enough to encode a set of P-Rules of good quality - that is, a set of P-Rules able to isolate the correct data - a list of medicine generic names has been constructed and provided to P-Rule C, which states that the correct data should be in that list. This strategy is perfectly working at a condition that the user knows in advance all the forms that the data could appear. Moreover, the provided list can be

dynamically modified in order to include a new form of the data, or to have a better control over the possible retrieved data. In this case, several complete lists of medicine generic names do exist, and by using one of them, it is easy to build a very good P-Rule. In fact, a 0.7 weight factor has been given to P-Rule C. Finally, P-Rule D is expressing that immediately before the desired data, the bigram "Active ingredient" could be present. Please note that the keyword POS, even though the bigram is located immediately before the desired data, is not using -1 as argument - expressing the fact that a 1 should be given to the last element of the nested main-list - but is using -2, to give a 1 to the second last element of the nested main-list. Although in the PDF document this is not the case, in the converted version a dot has been added between "Active ingredient" and the medicine generic name, and this is visible by printing the output using the -c argument, as explained in Section 4.3.3.

Finally, in the Additional-context definition part, the weights have been defined, main-list is sorted in decreasing order of probabilities and the item in first position is then extracted.

Extraction Results. Amongst 106 total PDF documents, the medicine generic name data was correctly extracted from 83 of them, resulting in a 78.3% accuracy. The 23 medicines for which the extracted data are wrong are listed in Table 3.

Medicine	Ground Truth	Extracted	Prob	Time [s]	n° items	Pages
Afinitor	everolimus	sunitinib	0.8	62.4	5783	9
Arzerra	ofatumumab	alemtuzumab	0.8	148.2	8601	10
Beromun	tasonermin	melphalan	0.85	161	8659	11
Cyramza	ramucirumab	paclitaxel	0.85	7.1	1905	2
Darzalex	daratumumab	lenalidomide	0.85	12.1	2591	2
Farydak	panobinostat	bortezomib	0.85	5.7	1723	2
Gazyvaro	obinutuzumab	rituximab	0.85	10.5	2375	2
Imnovid	pomalidomide	lenalidomide	0.85	735.8	17643	20
Iressa	gefitinib	docetaxel	0.8	151.6	8605	11
Kyprolis	carfilzomib	lenalidomide	0.85	5.1	1643	2
Lutathera	lutetium oxodotreotide	everolimus	0.75	8.5	2127	2
Mektovi	binimetinib	encorafenib	0.85	5.6	1701	2
Oncaspar	pegaspargase	asparaginase	0.85	8.3	2031	2
Pixuvri	pixantrone dimaleate	rituximab	0.8	267.4	11255	14
Sprycel	dasatinib	imatinib	0.8	279.3	10935	14
Tarceva	erlotinib	docetaxel	0.8	55.2	5333	9
Tasigna	nilotinib	imatinib	0.8	107.4	7235	11
Tecartus	autologous	ibrutinib	0.8	32.1	4111	5
Vargatef	nintedanib	docetaxel	0.85	9.3	2221	2
Vectibix	panitumumab	irinotecan	0.8	85.2	6585	9
Votrient	pazopanib	sunitinib	0.8	66.4	5857	8
Xofigo	radium dichloride	abiraterone	0.75	9.8	2311	2
Xtandi	enzalutamide	docetaxel	0.85	380.1	13397	16

Table 3: *Wrong results for the medicine generic name extraction.*

Again, and as will be for the remaining queries, for all the medicines listed in Appendix B that are not in Table 3, the extraction of the medicine generic name was correctly performed. In this case, multiple medicine generic names were mentioned in the same document (mainly for comparison reasons with the considered medicine), hence, even though the obtained accuracy is not low, more domain knowledge can be used to express new P-Rules that are able to better discriminate when a medicine generic name is mentioned for comparison reasons or is the actual data to extract.

5.3 Indication

It has been noticed that the indication data is a sentence that usually contains (or even starts with) the words “treatment of”. The sentence containing the indication data usually also contains either the words “is indicated for”, “is indicated in”, “is indicated as” or “is indicated with”, sometimes together with “treatment of” and sometimes alone. In some documents, the sentence containing the indication data starts with the words “Main points” or “Key points”. Finally, in some documents, the word “Indication” can either be the first word of the sentence containing the indication data, or can be the sentence before (a sentence with a single word, being the header of the Indication section).

Query. The query that encodes this domain knowledge in the P-Rules is as follows:

```
(WITH 0 LEN
TOKENIZEDBY "."
WHERE
A: MATCH "(T|t)reatment\sof\s"
B: MATCH "is\sindicated\s(for|in|as|with)\s"
C: MATCH "(Main|Key)\spoints\s"
D: MATCH "^Indication\s"
E: EXISTS (
    WITH 0 $this.START_POS
    TOKENIZEDBY "."
    WHERE
    A: MATCH "^Indication$"
    B: POS -1
    ENDQUERY)
DEFINE_PROBABILITY 0.4A + 0.1B + 0.25C + 0.15D + 0.1E
SORT "descending"
ENDQUERY)[0]
```

In the Main-list definition part, the whole length of the document has been considered in the `WITH` statement, while the `TOKENIZEDBY` instruction has created a list containing sentences.

Then, in the P-Rules definition part, P-Rules from A to D are trying to match a regular expression directly on the items in main-list. P-Rule A will assign 1 to the items in main-list having either “Treatment of ” or “treatment of ” within. P-Rule B assigns 1 to the items in main-list containing either “is indicated for”, “is indicated in”, “is indicated as” or “is indicated with”. P-Rule C is assigning 1 to the items containing “Main points ”, and P-Rule D assigns 1 to the items that start with the “Indication ” word. Finally, P-Rule E is selecting through the `WITH` statement the portion of text

comprised from the starting position (0) until the starting position (excluded) of the considered `$this` item. Then, that portion of text is split to create a nested main-list of sentences. Lastly, in the nested P-Rules definition part, P-Rule A is assigning 1 to the items of the nested main-list that contain the word "Indication" while P-Rule B is assigning a 1 to the last item of that list. Hence, a 1 will be assigned to the item `$this` if the item before is composed by the word "Indication". Please note that in this query, the `TOKENIZEDBY` instruction contained within the nested query - that has `$this` in its `WITH` statement - is using the exact same token as the outer (main) `TOKENIZEDBY` instruction, therefore, the optimized method of pre-computing the common used lists can be employed in this query. It is important to note that this query expresses the core difference between this system and other proposed systems in the literature, being that it is possible deconstruct an otherwise difficult to express regular expression, and encode the single parts in simpler regular expressions. Then, through the probability architecture of the system, the scores of each item are recomposed to obtain a final score. In this way, dealing with different document layouts (remarkably, the most difficult challenge in rule-based IE) is much easier compared to other rule-base systems, also improving the expressiveness of the language, and consequently, the extraction accuracy.

In the Additional-context definition part, the weights are assigned following the usual procedure, main-list is ordered by decreasing order of probabilities and the item in first position is retrieved.

Extraction Results. Amongst 106 total PDF documents, the indication data was correctly extracted from 88 of them, resulting in a 83% accuracy. The 18 medicines for which the extracted data are wrong are listed in Table 4, but this time without the "Ground Truth" and "Extracted" columns, due to the impossibility to clearly show two entire sentences for each medicine.

Medicine	Prob	Time [s]	n° items	Pages
Adcetris	0.8	4.5	410	20
Atriance	0.7	1.3	207	9
Caprelsa	0.7	3	390	13
Ceplene	0.7	1.9	321	8
Halaven	0.7	1.8	294	10
Iclusig	0.7	0.6	72	2
Javlor	0.7	1.4	227	8
Kadcyla	0.8	10.9	840	25
Mepact	0.7	1.5	235	9
Perjeta	0.7	3.2	384	15
Pixuvri	0.7	3	457	14
Rozlytrek	0.7	0.8	49	4
Sprycel	0.8	2.6	319	14
Tarceva	0.7	1.4	249	9
Thalidomide Pharmion	0.8	1.8	316	10
Tyverb	0.7	2.3	312	10
Yondelis	0.7	1.1	166	7
Zydelig	0.9	0.6	89	2

Table 4: *Wrong results for the indication extraction.*

In this case, 15 out of 18 wrong results (Adcetris, Atriance, Caprelsa, Ceplene, Halaven, Javlor, Kadcyla, Mepact, Pixuvri, Sprycel, Tarceva, Thalidomide Pharmion, Tyverb, Yondelis, Zydelig) contained an indication composed by two sentences, however, the system has only extracted the first sentence, failing at also extracting the second. This situation is classified as incomplete data extraction, hence, incorrect. If this limitation will be solved in the future, the extraction accuracy of the indication data would potentially increase from 83% to 97.2%. Then, for the 3 remaining medicines (Iclusig, Perjeta and Rozlytrek) a wrong sentence has been extracted.

5.4 Reimbursement Class

It has been noticed that the reimbursement class data is a bigram, where the first word is either “IACB”, “IAB”, “CAV”, “ASMR”, “level” or “grade” and the second word is always a roman number from one (I) to five (V).

Query. The query that encodes this domain knowledge in the P-Rules is as follows:

```
(WITH 0 LEN
TOKENIZEDBY "\bigrams"
WHERE
A: MATCH "(IACB|IAB|CAV|ASMR|level|grade)\s(V|IV|III|II|I)"
DEFINE_PROBABILITY 1A
SORT "descending"
ENDQUERY)[0]
```

In the Main-list definition part, the entire length of the document is selected by the `WITH` statement, and split using to `TOKENIZEDBY` construct to obtain a main-list of bigrams.

Then, in the P-Rules definition part, P-Rule A is sufficient to encode the provided domain knowledge in a single regular expression to match. The first part of the regular expression (before the first \s) is matching the first word, while everything after the \s is matching the roman number, from five (V) to one (I). The reason why the roman numbers are expressed from five to one (rather than from one to five) is to add a further level of safeness, since the regular expression is matched from left to right. Hence, if we were to write the numbers from one to five, a I could be matched when the actual number to match was a II, and the regular expression will stop to evaluate further tokens. Instead, by expressing the numbers from five to one, the regular expression is first trying to match a V, then, if there are no matches, tries with a IV, and so on. This entire procedure is not strictly necessary, since the data where the regular expression is applied is a bigram, but it’s better to have this extra level of robustness since it doesn’t require extra resources to run.

Finally, in the Additional-context definition part, a weight of 1 is given to P-Rule A - since obviously is the only P-Rule within the query - then, main-list is sorted in decreasing order of probability and the item in first position is retrieved.

Extraction Results. Amongst 106 total PDF documents, the reimbursement class data was correctly extracted from 90 of them, resulting in a 84.9% accuracy. The 16 medicines for which the extracted data are wrong are listed in Table 5.

Medicine	Ground Truth	Extracted	Prob	Time [s]	n° items	Pages
Besremi	n/a	Haematology Sectors	0	1	1210	3
Fotivda	n/a	ONCOLOGY New	0	0.6	916	2
Iclusig	III	CAV V	1	0.9	1471	2
Inlyta	IV	IAB Improvement	1	13.4	7776	16
Lartruvo	n/a	ONCOLOGY New	0	0.7	961	2
Mepact	n/a	The legally	0	3.1	3097	9
Nerlynx	n/a	Breast cancer	0	0.6	740	3
Polivy	n/a	CAV V	1	42.5	14599	29
Removab	V	The legally	0	2.3	2548	7
Rozlytrek	n/a	CAV V	1	1.3	1520	4
Teysuno	n/a	IAB V	1	2.4	2771	7
Vargatef	n/a	ONCOLOGY New	0	0.8	1110	2
Venclyxto	V	HAEMATOLOGY New	0	1.2	1758	3
Votrient	n/a	The legally	0	2.8	2928	8
Zelboraf	III	IAB IV	1	6.1	4815	12
Zytiga	III	IAB IV	1	4.9	4267	11

Table 5: *Wrong results for the reimbursement class extraction.*

For this data, for some medicines a n/a (not applicable) should have been extracted. However, the n/a was never explicitly mentioned in the document, making it impossible to extract. It would have been possible to extract a similar wording if when a n/a should have been extracted, the documents explicitly states something similar, however, no such pattern was found. In 7 out of 10 n/a cases, the regular expression specified in the query didn't match anything, since the probability of the returned data is 0. Hence, to solve the n/a problem, for this domain is more e□ective to apply a post-processing procedure that replaces the extracted data with a n/a if the probability of the extracted data is 0. This procedure will never overwrite a possibly correct data (having probability 1) since it will change only the data having 0 probability of being correct. At worst, if the ground truth was not n/a, the extracted data was already certainly wrong. By using this procedure, the reimbursement class of 7 more documents would have been correctly extracted, meaning that the extraction accuracy would increase from 84.9% to 91.5%.

5.5 Assessment Date

It has been noticed that the assessment date data is always located in the first page of the document, and is always the first mentioned date. Sometimes it is in format "dd month yyyy" while sometimes it is in format "month yyyy" (hence, the day is not stated). When the day is expressed and is less than 10, it may be expressed either as a single digit or double digits, having a 0 in front.

Query. The query that encodes this domain knowledge in the P-Rules is as follows:

```
(WITH 0 LEN
TOKENIZEDBY "\trigrams" AND "\bigrams"
WHERE
```

```
A: MATCH "(^\d{1,2}?\s\w{3,9}?\s\d{4}?$|^\w{3,9}?\s\d{4}?$)"
B: SUBSTRING (
    WITH 0 LEN
    TOKENIZEDBY "\p"
    WHERE
    A: POS 0
    ENDQUERY) [0]
DEFINE_PROBABILITY 0.9A + 0.1B
SORT "descending"
ENDQUERY) [0]
```

In the Main-list definition part, the whole length of the document has been considered in the `WITH` statement, and a list of bigrams and trigrams is created using the `TOKENIZEDBY` construct.

Then, in the P-Rules definition part, P-Rule A assigns 1 to all the items in main-list which satisfy the provided regular expression. The regular expression is composed as follows. The outer structure is an OR choice, in the format of (*xjy*). The *x* expression starts with a ^ symbol and ends with a $ symbol, meaning that to match the expression contained within the symbols ^ and $, a perfect match has to be made. The inner expression is responsible for matching the pattern “dd month yyyy” by searching for a sequence of 1 or 2 digits, then a space, then a word composed by minimum 3 and maximum 9 characters - corresponding to the shortest length month (May) and to the longest one (September) - then a space, and finally a number composed by 4 digits. Even the *y* expression starts with a ^ symbol and ends with a $ symbol, as the *x* expression. The inner expression is matching the pattern “month yyyy”, and works the same as the *x* expression, but without the part expressing the day. Finally, the question marks after each piece of expression mean that the behavior of that piece should be greedy, that is, finding that piece of data as few times as possible in the given expression. Although considering the design of the system the question mark is unnecessary, it makes the regular expression more robust. Then, P-Rules B assigns 1 to all the items that are present in the first page of the document. Please note that in the `TOKENIZEDBY` instruction, the first token is `\trigrams` so that if a trigram has the same probability score as a bigram, the trigram is retrieved. In fact, consider the scenario where the assessment date to retrieve is for example 01 February 1995. In main-list, there will be a trigram having as text “01 February 1995” and a bigram having as text “February 1995”. In this case, both items will be given the same probability score by both P-Rules, however, by making main-list with trigrams first and then the bigrams, the data with more information - being the assessment date containing the day - is correctly retrieved.

Finally, in the Additional-context definition part, the highest weight has been given to P-Rule A, main-list is sorted in decreasing order of probabilities, and the item in first position is retrieved.

Extraction Results. Amongst 106 total PDF documents, the assessment date data was correctly extracted from 105 of them, resulting in a 99.1% accuracy. The only medicine for which the extracted data is wrong is listed in Table 6.

Medicine	Ground Truth	Extracted	Prob	Time [s]	n° items	Pages
Beromun	01 October 2014	10 October 2013	1	11.4	8657	11

Table 6: *Wrong results for the assessment date extraction.*

For this data, the extraction accuracy is close to 100%, showing the robustness of the assessment date query. However, for the Beromun medicine, the date was stated in the document as "1st October 2014" - having the "st" after the 1 - which was not considered in the possible cases to design the P-Rules because never found in the inspected documents.

5.6 Extraction Times Analysis

Since the extraction time has been individuated as a critical factor, an analysis has been performed with the objective of understanding the extent of this problem. For a document, the extraction time crucially depends on the number of items that main-list is composed of, since the P-Rules are applied over each of these items. Hence, together with the extracted data, the JSON file produced by the extraction is also storing the number of items that were inserted in main-list for that execution, and the total extraction time required. For each query, it is very insightful to plot the extraction times of each execution as a function of the number of items stored in main-list, as visible in Figure 6.

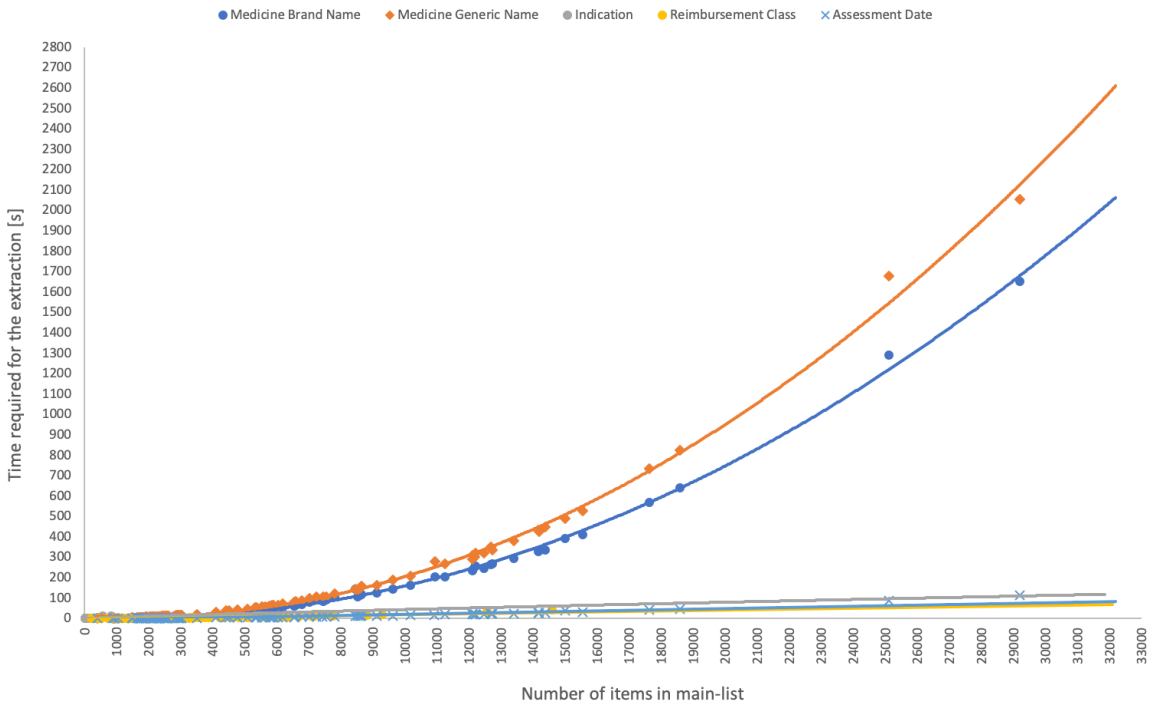

Figure 6: *Time required for the extraction as a function of the number of items in main-list.*

In Figure 6, for each document, the number of items that were inserted in main-list and the total time for extraction are plotted. Thus, each point represents a document, and a same document is considered 5 times, once for each query. Hence, having 106 total documents and 5 queries, 106 5 = 530 points are pictured in the figure. Then, for each query, a trend-line has been drawn using the regression method - linear, power and polynomial - by choosing the best regression model based on visual inspection. Furthermore, when the aforementioned 5 data have to be extracted from a new document, by knowing how many items main-list of that document will be composed of, for each query, it is possible to predict how much time the extraction will take.

For the medicine brand name and the medicine generic name queries, a polynomial regression model having degree 2 has been chosen as the model that best fits the data, obtaining a R^2 value of 99.77% and 99.68% respectively. In fact, due to P-Rule F - for the medicine brand name query - and P-Rule D - for the medicine generic name query - the extraction time has become quadratic in the number of items in main-list. The resulting equations of the trend-lines are as follows:

$$\textit{Medicine brand name} : y = 0{:}000003x^2 \quad 0{:}0087x + 15{:}12 \tag{5}$$

$$\textit{Medicine generic name} : y = 0{:}000002x^2 \quad 0{:}0076x + 14{:}489 \tag{6}$$

Furthermore, both nested queries have `$this` in their `WITH` statement and the tokens of the nested `TOKENIZEDBY` instructions are not the same as the tokens used in the main `TOKENIZEDBY` instruction, thus, the optimized solution has been disabled for both queries. The difference in time is due to the fact that in the nested query of the medicine generic name query, the token used by the `TOKENIZEDBY` instruction is `\bigrams`, while the token used in the nested query of the medicine brand name query is a space character, to produce a list of words. Considering the implementation of the system, and considering that the optimized method has been disabled, to produce a list of bigrams, the system first produces a list of words, and then runs an algorithm to concatenate words to produce bigrams. Since this concatenation algorithm runs every time a nested query is performed, this explains the difference in time between the two queries. However, generally, both queries were pretty fast for almost all the documents, which are concentrated towards the bottom left part of the graph. If on the y-axis we consider 2 minutes as the point where the curves are getting steeper, we obtain - for the medicine brand name query - a count of 83 documents that have completed the extraction procedure in under 2 minutes, while 23 are above 2 minutes, and - for the medicine generic name query - a count of 78 documents that have completed the extraction procedure in under 2 minutes, while 28 are above 2 minutes. For the medicine brand name query, the minimum obtained value is 0.8 s (for 407 checked items, medicine: Foscan), while the maximum obtained value is 1651.2 s (roughly 28 minutes, for 29199 checked items, medicine: Polivy), with a mean value of 105.9 s, a median value of 13.8 s and a mode value of 5.8 s. For the medicine generic name query, the minimum obtained value is 0.5 s (for 407 checked items, medicine: Foscan), while the maximum obtained value is 2054.7 s (roughly 34 minutes, for 29199 checked items, medicine: Polivy), with a mean value of 134.4 s, a median value of 17.2 s and a mode value of 8.3 s.

On the contrary, a linear regression model has been chosen for the reimbursement class and assessment date queries, obtaining a R^2 value of 89.01% and 87.44%, respectively. The resulting equations of the trend-lines are as follows:

$$\textit{Reimbursement class} : y = 0{:}0023x \quad 2{:}6089 \tag{7}$$

$$\textit{Assessment date} : y = 0{:}0029x \quad 7{:}5467 \tag{8}$$

For the reimbursement class query, the minimum obtained value is 0.2 s (for 203 checked items, medicine: Foscan), while the maximum obtained value is 42.5 s (for 14599 checked items, medicine: Polivy), with a mean value of 4 s, a median value of 1.2 s and a mode value of 0.6 s. For the

assessment date, the minimum obtained value is 0.1 s (for 405 checked items, medicine: Foscan), while the maximum obtained value is 113 s (for 29197 checked items, medicine: Polivy), with a mean value of 9.2 s, a median value of 2 s and a mode value of 0.8 s. In this case, the resulting extraction times are more than acceptable, being rather low.

Finally, a power model has been chosen for the indication data, obtaining a R^2 value of 86.05%. The resulting equation of the trend-line is as follows:

$$Indication : y = 0{:}0179x^{0{:}8489} \tag{9}$$

However, due to the fact that all the dots are condensed towards the bottom left of the graph - since the documents have been split in sentences, meaning that the number of items in main-list is extremely low if compared to other queries - for the indication data this model may not be appropriate considering that the maximum value on the x axis is 866, resulting in a lack of data. For the indication query, the minimum obtained value is 0.1 s (for 19 checked items, medicine: Foscan), while the maximum obtained value is 14.6 s (for 580 checked items, medicine: Giotrif), with a mean value of 1.7 s, a median value of 0.8 s and a mode value of 0.5 s. Even for this query, the extraction times are very low, hence, they are acceptable and don't constitute a problem for the system.

5.7 Metrics Summary

Table 7 summarizes for each query all the results described in the previous sections.

	MBN	MGN	I	RC	AD
Documents extraction OK	90	83	88	90	105
Documents extraction WRONG	16	23	18	16	1
Accuracy	84.9%	78.3%	83%	84.9%	99.1%
Min extraction time [s]	0.8	0.5	0.1	0.2	0.1
Max extraction time [s]	1651.2	2054.7	14.6	42.5	113
Mean extraction time [s]	105.9	134.4	1.7	4	9.2
Median extraction time [s]	13.8	17.2	0.8	1.2	2
Mode extraction time [s]	5.8	8.3	0.5	0.6	0.8
Min number of items	407	407	19	203	405
Max number of items	29199	29199	866	14599	29197
Mean number of items	5825	5825	193	2912	5823
Median number of items	3029	3029	78	1514	3027
Mode number of items	12113	12113	70	6056	12111

Table 7: *Summary of all the obtained results.*

In Table 7, MBN stands for medicine brand name, MGN stands for medicine generic name, I stands for indication, RC stands for reimbursement class and AD stands for assessment date. It can be noticed that the obtained average extraction accuracy across the 5 queries is about 86%.

6 Discussion

The main research challenge (together with all its four sub-challenges) discussed in Section 1.2 has been tackled and solved.

The obtained results are very promising, indicating that the path of creating a probability based query language to perform the IE task not only is possible, but it is effective. Indeed, the obtained extraction accuracies are aligned with the accuracies described in the literature for similar tasks, however, the greatest advantage of using ProbQL is its general nature, meaning that this same tool can be used in different domains, without being restricted to one in particular. Furthermore, some improvements are possible in order to further increase the extraction accuracies, but due to time constraints, they have not been implemented, but only discussed in Section 6.2.

6.1 Limitations

In its current state, the system has achieved a high level in terms of quality of extracted data. However, there still are limitations that prevent a further improvement.

(1) A first limitation is that the Item objects having the same text, although they basically refer to the same concept, they are treated as different. This can be a problem in the following scenario. Suppose that some P-Rules are designed in order to extract a specific piece of data (e.g. a word), and suppose that this piece of data is repeated multiple times throughout the PDF document. The designing process of the rules, should usually consists of an inspection of multiple documents having different layouts, where the desired data is present. Then, some patterns have to be manually recognized and encoded in the P-Rules, in order to have a valid method able to successfully discriminate from all the data in the document, the one that is desired to be retrieved. To do that, the P-Rules should assign the highest probability to that specific piece of data. However, if a P-Rule *A* is saying that the desired data is for example in the first page, and another P-Rule *B* is saying that two words after the desired data there is the word “mg”, if in the first page there is an instance of the desired data but it never has the word “mg” after two words, but in the second page there is an instance that satisfies P-Rule *B*, even though the desired data is the same entity (no matter its position in the document, or the position of other data relatively to the desired one) the probabilities are split between the two. Hence, supposing that P-Rule *A* gives a reward of 0.15 and P-Rule *B* gives a reward of 0.3, the entity should have a probability score of 0.45, but in the current state of the program, the instance of the desired data in the first page will be given a score of 0.15, while the instance of the desired data in the second page will be given a score of 0.3, even though they refer to the same concept.

(2) A second limitation is speed, mainly when the query contains a nested query that have to

be repeated multiple times. As previously discussed, the system has been implemented in Python, which it's definitely not the best performing language in terms of execution speed. An optimization approach that is able to cut down the execution times has been proposed, however, its limitation lies in the assumption that has been made in order to be able to exploit the benefits in velocity.

(3) Finally, the third limitation is that when the same data can assume di□erent forms (e.g. bigram and trigram, as for the person's names) the quality of the retrieved data is strongly dependent on the quality of the defined P-Rules, which have to be carefully designed by keeping in mind that the same P-Rule should work for every form that the data is expected to appear. However, when the same data to extract could for example take either the form of a sentence or the form of two sentences, in the current state of the program is not possible to extract both sentences, but only one will be retrieved.

6.2 Future Directions

A straightforward way to solve limitation (1) - mentioned in Section 6.1 - could be the following procedure. When a piece of data *d* is rewarded with a certain probability score from a P-Rule, then the same reward should also be given to all the data *d* that have not received that reward from the considered P-Rule. However, this procedure has not been implemented, since there exists a much better and clean method that can be utilized in order to solve the aforementioned problem, that comes "for free" with an update that could be made in the future, that is, the integration of the program as a whole with a Named Entity Recognition (NER) system. In fact, not only this problem can be easily solved by making each `Item` object pointing to a corresponding entity object (hence, all the `Item` objects that share the same text, are now pointing to the same entity), but all the specific ontologies from the most di□erent domains could be used to express more specific P-Rules, obtaining the advantage of retrieving data with even more accuracy. In fact, if every piece of data that has been tokenized from the PDF is mapped to an existing entity, it will be much easier to express more specific P-Rules by making use of some kind of ontology. Furthermore, NER systems are often integrated with Part-of-Speech (POS) tagging systems, which can be in turn integrated in `ProbQL`. For example, adopting a POS tagging system to tag each tokenized item in main-list can be useful, since it allows to express some P-Rules that are using these tags, e.g., if one is searching for a data that is suspected to be an adjective, a P-Rule that gives a score to each item in main-list having the ADJ tag can be expressed.

Regarding other future directions, an integration with a machine learning approach can also be an interesting one, in order to exploit the benefits of both worlds, namely, rule-based data extraction and machine learning based data extraction. For example, NLP techniques such as considering each tokenized `Item` object as a feature in order to apply a learning approach to give a score to each feature and combining the results of both approaches seems a viable solution. Even more, also including techniques such as computing the *mutual information* score between each item in main-list seems a valid approach to improve the extraction accuracy.

Furthermore, some minor improvements are also possible. For example, the weights contained in the `DEFINE_PROBABILITY` construct are now manually decided, based on domain knowledge of the final user. However, weight tuning is a classic problem in the machine learning world, that have produced lots of methods and solutions (e.g. cross-validation) over the time, that are capable of

performing an optimal weight tuning to optimize some sort of metrics (usually, accuracy).

More constructs can also be identified and included in the grammar of the language, in order to improve its expressiveness, and other optimization methods can be identified to improve the speed performances of the system, especially when a nested query contains `$this` in its `WITH` statement.

Finally, the semantics of `ProbQL` has been defined only in an informal manner, thus, defining a formal operational semantics (e.g. in form of an abstract machine) can also be possible, obtaining the benefit of better reasoning on `ProbQL` queries and the capability of proving possible theorems.

7 Conclusion

In recent years, Information Extraction has become an extremely important field due to the vast amount of raw data being produced at an ever-increasing rate. However, our ability to collect these raw data is far greater than our ability to process and reorganize them, giving them a structure. Nowadays, most of these raw data are in fact unstructured - such as PDF documents, or text written in natural language - making processes like data mining, pattern recognition and, more generally, structured database construction, virtually impossible. Regarding the extraction of information from texts, some techniques have been proposed in the literature, but in most of them, a part which was tailored on the specific problem they are trying to solve has been included, therefore lacking of generality. Others that are general enough - i.e. the development of programming languages to perform the IE task - seems to be much focused on solving the problem with extensive usage of regular expressions, therefore lacking of expressiveness.

In this Thesis, a novel programming language - ProbQL - has been proposed, which is a rule-based query language through which it is possible to write queries able to extract information from a PDF document or a text file. The main idea of this language is to split the given text into a list of items, and then score each item with a probability value of being the correct data to extract, based on some rules defined by the final user, describing some properties that the desired data is thought to have. A set of useful constructs has been implemented in the language, and a proper grammar together with syntax rules have been defined in order to write a valid query. Then, the language has been tested on the task of extracting 5 types of data from PDF documents, obtaining an average extraction accuracy of 86% (with a minimum of 78.3% and a maximum of 99.1%).

Unstructured data are notoriously difficult to interact with, due to their intrinsic uncertainty nature. This means that the same information could be expressed in different ways or located in different positions based on the type of PDF document (or text) at hand. Thus, even for the same extraction task, there could exist documents having different structures and layouts. Hence, often times a final user doesn't know in advance which type of structure the document where they are trying to extract the data from will have. This situation has a similar underlying concept of the quantum mechanics theory - Heisenberg uncertainty principle - which deals with particles positions. In fact, also particles have an intrinsic uncertainty nature, and to know a particle position (as a data in a document) the *probability* of finding that particle in a certain position is instead considered. We believe that switching the paradigm from developing increasingly efficient techniques that are however tailored on a specific problem or domain to developing a *probability* based technique, which can perfectly deal with the uncertainty of a piece of data being there, can lead to the design of a general programming language that will be the standard - as SQL is for extracting data from structured databases - for doing IE on unstructured data, and ProbQL is a first step in this direction.

Bibliography

[1] V. Shubhangi, B. Erick, E. Stephen, D. Melissa, N. David, and E. Bern, 2020 market guide for text analytics, 2020.

[2] L. Chiticariu, Y. Li, and F. Reiss, Rule-based information extraction is dead! long live rule-based information extraction systems!, EMNLP 2013 - 2013 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference, vol. October, pp. 827 832, 01 2013.

[3] N. Sangeeth and R. Rejimoan, An intelligent system for information extraction from relational database using hmm, in 2015 International Conference on Soft Computing Techniques and Implementations (ICSCTI) , pp. 14 17, 2015.

[4] S. M. Thede and M. P. Harper, A second-order hidden markov model for part-of-speech tagging, in Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics on Computational Linguistics, ACL '99, (USA), p. 175 182, Association for Computational Linguistics, 1999.

[5] F. Suchanek, G. M. Kasneci, and G. M. Weikum, Yago: A Core of Semantic Knowledge Unifying WordNet and Wikipedia, in 16th international conference on World Wide Web, Proceedings of the 16th international conference on World Wide Web, (Ban , Canada), pp. 697 697, May 2007.

[6] C. Fellbaum, WordNet: An Electronic Lexical Database. Bradford Books, 1998.

[7] M. Wick, Geonames ontology. Available at http://www.geonames.org/.

[8] J. Ho art, F. M. Suchanek, K. Berberich, and G. Weikum, Yago2: A spatially and temporally enhanced knowledge base from wikipedia, Arti cial Intelligence , vol. 194, pp. 28 61, 2013. Arti cial Intelligence, Wikipedia and Semi-Structured Resources.

[9] D. Vrande£i¢ and M. Krötzsch, Wikidata: A free collaborative knowledgebase, Commun. ACM, vol. 57, p. 78 85, sep 2014. Available at https://www.wikidata.org/wiki/Wikidata:Main_Page.

[10] F. Mahdisoltani, J. A. Biega, and F. M. Suchanek, Yago3: A knowledge base from multilingual wikipedias, in CIDR , 2015.

[11] R. V. Guha, D. Brickley, and S. Macbeth, Schema.org: Evolution of structured data on the web, Commun. ACM, vol. 59, p. 44 51, jan 2016.

[12] T. Pellissier Tanon, G. Weikum, and F. Suchanek, Yago 4: A reason-able knowledge base, in The Semantic Web (A. Harth, S. Kirrane, A.-C. Ngonga Ngomo, H. Paulheim, A. Rula, A. L. Gentile, P. Haase, and M. Cochez, eds.), (Cham), pp. 583 596, Springer International Publishing, 2020.

[13] G. Zaman, H. Mahdin, K. Hussain, and A. Rahman, Information extraction from semi and unstructured data sources: A systematic literature review, ICIC Express Letters, vol. 14, pp. 593 603, 06 2020.

[14] D. Hindle, Noun classi cation from predicate-argument structures, in Proceedings of the 28th Annual Meeting on Association for Computational Linguistics, ACL '90, (USA), p. 268 275, Association for Computational Linguistics, 1990.

[15] A. Akbik, D. Blythe, and R. Vollgraf, Contextual string embeddings for sequence labeling, in Proceedings of the 27th International Conference on Computational Linguistics, (Santa Fe, New Mexico, USA), pp. 1638 1649, Association for Computational Linguistics, Aug. 2018.

[16] D. Nadeau and S. Sekine, A survey of named entity recognition and classi cation, Lingvisticae Investigationes, vol. 30, pp. 3 26, Jan. 2007.

[17] T. Poibeau and L. Kosseim, Proper Name Extraction from Non-Journalistic Texts, pp. 144 157. Leiden, The Netherlands: Brill, 2001.

[18] W. Zaremba, I. Sutskever, and O. Vinyals, Recurrent neural network regularization, CoRR, vol. abs/1409.2329, 2014.

[19] I. Sutskever, J. Martens, and G. Hinton, Generating text with recurrent neural networks, pp. 1017 1024, 01 2011.

[20] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, Deep contextualized word representations, in Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), (New Orleans, Louisiana), pp. 2227 2237, Association for Computational Linguistics, June 2018.

[21] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, Neural architectures for named entity recognition, CoRR, vol. abs/1603.01360, 2016.

[22] D. M. Bikel, S. Miller, R. Schwartz, and R. Weischedel, Nymble: a high-performance learning name- nder, in Fifth Conference on Applied Natural Language Processing, (Washington, DC, USA), pp. 194 201, Association for Computational Linguistics, Mar. 1997.

[23] S. Sekine, Description of the Japanese NE system used for MET-2, in Seventh Message Understanding Conference (MUC-7): Proceedings of a Conference Held in Fairfax, Virginia, April 29 - May 1, 1998, 1998.

[24] M. Asahara and Y. Matsumoto, Japanese named entity extraction with redundant morphological analysis, in Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics, pp. 8 15, 2003.

[25] D. Nadeau, P. D. Turney, and S. Matwin, Unsupervised named-entity recognition: Generating gazetteers and resolving ambiguity, in Advances in Arti cial Intelligence (L. Lamontagne and M. Marchand, eds.), (Berlin, Heidelberg), pp. 266 277, Springer Berlin Heidelberg, 2006.

[26] M. Pasca, D. Lin, J. Bigham, A. Lifchits, and A. Jain, Organizing and searching the world wide web of facts - step one: the one-million fact extraction challenge, in Proceedings of the 21st National Conference on Arti cial Intelligence (AAAI-06) , (Boston, Massachusetts), pp. 1400 1405, 2006.

[27] A. Alhelbawy and R. Gaizauskas, Collective named entity disambiguation using graph ranking and clique partitioning approaches, in Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers, (Dublin, Ireland), pp. 1544 1555, Dublin City University and Association for Computational Linguistics, Aug. 2014.

[28] T. Eftimov, B. Korou²i¢ Seljak, and P. Koro²ec, A rule-based named-entity recognition method for knowledge extraction of evidence-based dietary recommendations, PLOS ONE, vol. 12, pp. 1 32, 06 2017.

[29] Apache, Apache opennlp maxent sentence detector. Documentation available at https://opennlp.apache.org/docs/1.5.3/manual/opennlp.htmltools.sentdetect.

[30] Apache, Apache opennlp maxent chunker. Documentation available at https://opennlp.apache.org/docs/1.5.3/manual/opennlp.htmltools.chunker.

[31] M. Palmer, D. Gildea, and P. Kingsbury, The Proposition Bank: An Annotated Corpus of Semantic Roles, Computational Linguistics, vol. 31, pp. 71 106, 03 2005.

[32] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, Building a large annotated corpus of english: The penn treebank, Comput. Linguist., vol. 19, p. 313 330, jun 1993.

[33] M. Palmer, J. Rosenzweig, and S. Cotton, Automatic predicate argument analysis of the Penn TreeBank, in Proceedings of the First International Conference on Human Language Technology Research, 2001.

[34] V. Nebot and R. Berlanga, Exploiting semantic annotations for open information extraction: An experience in the biomedical domain, Knowledge and Information Systems, vol. 38, 02 2014.

[35] A. Fader, S. Soderland, and O. Etzioni, Identifying relations for open information extraction, in Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing (Edinburgh, Scotland, UK.), pp. 1535 1545, Association for Computational Linguistics, July 2011.

[36] Mausam, M. Schmitz, S. Soderland, R. Bart, and O. Etzioni, Open language learning for information extraction, in Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, (Jeju Island, Korea), pp. 523 534, Association for Computational Linguistics, July 2012.

[37] P. Palaga, L. Nguyen, U. Leser, and J. Hakenberg, High-performance information extraction with alibaba, vol. 360, pp. 1140 1143, 01 2009.

[38] NLM, Pubmed. Available at https://pubmed.ncbi.nlm.nih.gov.

[39] V. I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals., Soviet Physics Doklady vol. 10, pp. 707 710, feb 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[40] O. Bodenreider, The uni ed medical language system (umls): integrating biomedical terminology., Nucleic Acids Research, vol. 32, pp. 267 270, 2004.

[41] C. Ahlers, M. Fiszman, D. Demner-Fushman, F.-M. Lang, and T. Rind esch, Extracting semantic predications from medline citations for pharmacogenomics, Paci c Symposium on Biocomputing. Paci c Symposium on Biocomputing, vol. 12, pp. 209 20, 02 2007.

[42] A. Aronson, E ective mapping of biomedical text to the umls metathesaurus: The metamap program, Proceedings / AMIA ... Annual Symposium. AMIA Symposium , vol. 2001, pp. 17 21, 02 2001.

[43] I. Alimova and E. Tutubalina, Multiple features for clinical relation extraction: A machine learning approach, Journal of Biomedical Informatics , vol. 103, p. 103382, 2020.

[44] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, in Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers) (Minneapolis, Minnesota), pp. 4171 4186, Association for Computational Linguistics, June 2019.

[45] Z. Guo and H. Jin, A rule-based framework of metadata extraction from scienti c papers, in 2011 10th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, pp. 400 404, 2011.

[46] A. M. Hashmi, M. T. Afzal, and S. u. Rehman, Rule based approach to extract metadata from scienti c pdf documents, in 2020 5th International Conference on Innovative Technologies in Intelligent Systems and Industrial Applications (CITISIA) , pp. 1 4, 2020.

[47] Free leconverter. Available at https://www.free leconvert.com.

[48] A. Bartoli, G. Davanzo, A. D. Lorenzo, E. Medvet, and E. Sorio, Automatic synthesis of regular expressions from examples, Computer, vol. 47, pp. 72 80, dec 2014.

[49] L. Chiticariu, V. Chu, S. Dasgupta, T. W. Goetz, C. T. H. Ho, R. Krishnamurthy, A. Lang, Y. Li, B. Liu, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu, The systemt ide: an integrated development environment for information extraction rules, in SIGMOD '11 , 2011.

[50] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan, Declarative information extraction using datalog with embedded extraction predicates., pp. 1033 1044, 01 2007.

[51] D. FERRUCCI and A. LALLY, Uima: an architectural approach to unstructured information processing in the corporate research environment, Natural Language Engineering, vol. 10, no. 3-4, p. 327 348, 2004.

[52] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan, Gate: A framework and graphical development environment for robust nlp tools and applications, 07 2002.

[53] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren, Spanners: a formal framework for information extraction, in Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013 (R. Hull and W. Fan, eds.), pp. 37 48, ACM, 2013.

[54] W.-F. Hsiao, T.-M. Chang, and E. Thomas, Extracting bibliographical data for pdf documents with hmm and external resources, Program: electronic library and information systems, vol. 48, pp. 293 313, 07 2014.

[55] Apache, Apache pdfbox. Documentation available at https://pdfbox.apache.org/blog/.

[56] H. Yang, C. A. Aguirre, M. F. De La Torre, D. Christensen, L. Bobadilla, E. Davich, J. Roth, L. Luo, Y. Theis, A. Lam, T. Y.-J. Han, D. Buttler, and W. H. Hsu, Pipelines for procedural information extraction from scienti c literature: Towards recipes using machine learning and data science, in 2019 International Conference on Document Analysis and Recognition Workshops (ICDARW) , vol. 2, pp. 41 46, 2019.

[57] M. De La Torre, C. Aguirre, B. Anshutz, and W. Hsu, Matesc: Metadata-analytic text extractor and section classi er for scienti c publications, pp. 261 267, Jan. 2018. IC3K 2018 - Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management ; Conference date: 01-01-2018.

[58] L. Hawizy, D. Jessop, N. Adams, and P. Murray-Rust, Chemicaltagger: A tool for semantic text-mining in chemistry, Journal of cheminformatics, vol. 3, p. 17, 05 2011.

[59] N. Chomsky, Syntactic structures, 1957.

[60] H. Bast and C. Korzen, A benchmark and evaluation for text extraction from pdf, in 2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL) , pp. 1 10, 2017.

[61] C. Korzen, Icecite. Available at https://github.com/ckorzen/icecite/.

[62] R. Berg, Pdfextract. Available at https://github.com/oyvindberg/PDFExtract/.

[63] Y. Shinyama, Pdfminer. Available at https://github.com/euske/pdfminer/.

[64] J. Singer-Vine, Pdfplumber. Available at https://github.com/jsvine/pdfplumber/.

[65] R. A. Vreman, A. K. Mantel-Teeuwisse, A. M. Hövels, H. G. Leufkens, and W. G. Goettsch, Di erences in health technology assessment recommendations among european jurisdictions: The role of practice variations, Value in Health, vol. 23, no. 1, pp. 10 16, 2020.

[66] S. Marlowet al., Haskell 2010 language report, Available online http://www. haskell. org/(May 2011), 2010.

[67] G. Van Rossum, The Python Library Reference, release 3.10.5 Python Software Foundation, 2022. Available at https://docs.python.org/3/library/ast.html.

[68] Python, Python data structure time complexities. Available at https://wiki.python.org/moin/TimeComplexity/.

[69] G. Van Rossum, The Python Library Reference, release 3.8.2 Python Software Foundation, 2020. Available at https://docs.python.org/3/library/re.html.

[70] G. Van Rossum, The Python Library Reference, release 3.10.6 Python Software Foundation, 2022. Available at https://docs.python.org/3/library/json.html.

[71] B. Ojokoh, O. Adewale, and F. Samuel oluwole, Automated document metadata extraction, J. Information Science, vol. 35, pp. 563 570, 09 2009.

A Examples of construct usages

A.1 WITH

```
context :: "Hi, I 'm Daniele Di Grandi"

>>> WITH 0 LEN
"Hi, I 'm Daniele Di Grandi"

>>> WITH 1 LEN
"i, I 'm Daniele Di Grandi"

>>> WITH 1 23
"i, I 'm Daniele Di Grand"

>>> WITH $this.START_POS LEN (assume that $this are words)
"Hi, I 'm Daniele Di Grandi"
"I 'm Daniele Di Grandi"
"Daniele Di Grandi"
"Di Grandi"
"Grandi"

>>> WITH 0 $this.END_POS (assume that $this are words)
"Hi"
"Hi, I 'm"
"Hi, I 'm Daniele"
"Hi, I 'm Daniele Di"
"Hi, I 'm Daniele Di Grandi"

>>> WITH $this.END_POS LEN (assume that $this are words)
"I 'm Daniele Di Grandi"
"Daniele Di Grandi"
"Di Grandi"
"Grandi"
```

A.2 TOKENIZEDBY

```
context :: "Hi, I 'm Daniele Di Grandi. \p Hello."

>>> TOKENIZEDBY " "
["Hi", ",", "I  'm", "Daniele", "Di", "Grandi", "\p", "Hello"]

>>> TOKENIZEDBY "."
["Hi, I 'm Daniele Di Grandi", "\p", "Hello"]

>>> TOKENIZEDBY "\bigrams"
["Hi ,", ", I 'm", "I 'm Daniele", "Daniele Di", "Di Grandi", "Grandi \p",
    "\p Hello"]

>>> TOKENIZEDBY "\trigrams"
["Hi , I 'm", ", I 'm Daniele", "I 'm Daniele Di", "Daniele Di Grandi", "Di
    Grandi \p", "Grandi \p Hello"]

>>> TOKENIZEDBY "\p"
["Hi, I 'm Daniele Di Grandi.", "Hello."]

>>> TOKENIZEDBY "Daniele"
["Hi, I 'm", "Di Grandi. \p Hello"]

>>> TOKENIZEDBY "\bigrams" AND "\trigrams"
["Hi ,", ", I 'm", "I 'm Daniele", "Daniele Di", "Di Grandi", "Grandi \p",
    "\p Hello", "Hi , I  'm", ", I 'm Daniele", "I 'm Daniele Di", "Daniele
    Di Grandi", "Di Grandi \p", "Grandi \p Hello"]

>>> TOKENIZEDBY "\trigrams" AND "\bigrams"
["Hi , I 'm", ", I 'm Daniele", "I 'm Daniele Di", "Daniele Di Grandi", "Di
    Grandi \p", "Grandi \p Hello", "Hi ,", ", I  'm", "I 'm Daniele", "
   Daniele Di", "Di Grandi", "Grandi \p", "\p Hello"]

>>> TOKENIZEDBY " " AND "Daniele" AND "."
["Hi", ",", "I  'm", "Daniele", "Di", "Grandi", "\p", "Hello", "Hi, I   'm",
   "Di Grandi. \p Hello", "Hi, I  'm Daniele Di Grandi", "\p", "Hello"]
```

A.3 IN

```
main=list  :: ["Hi", ",", "I  'm", "Daniele", "Di", "Grandi", "\p", "Hi", "
   Daniele Alice"]
```

```
>>> A: IN (
        WITH 0 LEN
        TOKENIZEDBY " "
        WHERE
        A: POS 0
        ENDQUERY)
[1, 0, 0, 0, 0, 0, 0, 0]

>>> A: IN LOAD_LIST_FROM_FILE "path_containing_personal_names"
[0, 0, 0, 1, 0, 0, 0, 0, 0]

>>> A: IN LOAD_LIST_FROM_FILE "path_containing_personal_names" SPLIT
[0, 0, 0, 1, 0, 0, 0, 0, 1]

>>> A: IN _FREQLIST 2
[1, 0, 0, 0, 0, 0, 0, 1, 0]

>>> A: IN [".", ",", ";", ":"]
[0, 1, 0, 0, 0, 0, 0, 0, 0]
```

A.4 LOAD_LIST_FROM_FILE

```
main=list :: ["Hi", ",", "I 'm", "Daniele", "Di", "Grandi", "\p", "Hi", "
    Daniele Alice"]

>>> A: IN LOAD_LIST_FROM_FILE "path_containing_personal_names"
[0, 0, 0, 1, 0, 0, 0, 0, 0]

>>> A: IN LOAD_LIST_FROM_FILE "path_containing_personal_names" SPLIT
[0, 0, 0, 1, 0, 0, 0, 0, 1]

>>> A: NOT IN LOAD_LIST_FROM_FILE "path_containing_personal_names"
[1, 1, 1, 0, 1, 1, 1, 1, 1]

>>> A: NOT IN LOAD_LIST_FROM_FILE "path_containing_personal_names" SPLIT
[1, 1, 1, 0, 1, 1, 1, 1, 0]
```

A.5 NOT

```
See examples of LOAD_LIST_FROM_FILE.
```

A.6 POS

```
main=list :: ["Hi", ",", "I 'm", "Daniele", "Di", "Grandi", "\p", "Hi"]

>>> A: POS 0
[1, 0, 0, 0, 0, 0, 0, 0]

>>> A: POS 2
[0, 0, 1, 0, 0, 0, 0, 0]

>>> A: POS =1
[0, 0, 0, 0, 0, 0, 0, 1]

>>> A: POS =2
[0, 0, 0, 0, 0, 0, 1, 0]
```

A.7 MATCH

```
main=list :: ["Hi", ",", "I 'm", "Daniele", "Di", "Grandi", "\p", "Hi"]

>>> A: MATCH "Hi"
[1, 0, 0, 0, 0, 0, 0, 1]

>>> A: MATCH "Hello"
[0, 0, 0, 0, 0, 0, 0, 0]

>>> A: MATCH "I"
[0, 0, 1, 0, 0, 0, 0, 0]

main=list :: ["February 1995", "1995 Hello", "Hello everybody"]

>>> A: MATCH "\w{3,9}?\s\d{4}?"
[1, 0, 0]
```

A.8 EXISTS

```
context :: "SUNITINIB BIOGARAN 37,5 mg"

Suppose to: TOKENIZEDBY " " AND "\bigrams"
```

```
main=list :: ["SUNITINIB", "BIOGARAN", "37,5", "mg", "SUNITINIB BIOGARAN
    ", "BIOGARAN 37,5", "37,5 mg"]

>>> A: EXISTS (
        WITH $this.END_POS LEN
        TOKENIZEDBY " "
        WHERE
        A: MATCH "mg"
        B: POS 1
        ENDQUERY)
[0, 1, 0, 0, 1, 0, 0]

context :: "30 $ Active ingredient sunitinib"

Suppose to: TOKENIZEDBY " "

main=list :: ["30", "$", "Active", "ingredient", "sunitinib"]

>>> A: EXISTS (
        WITH 0 $this.END_POS
        TOKENIZEDBY "\bigrams"
        WHERE
        A: MATCH "Active\singredient"
        B: POS =2
        ENDQUERY)
[0, 0, 0, 0, 1]

>>> A: EXISTS (
        WITH $this.START_POS LEN
        TOKENIZEDBY " "
        WHERE
        A: MATCH " $"
        B: POS 1
        ENDQUERY)
[1, 0, 0, 0, 0]
```

A.9 SUBSTRING

```
context :: "Hi, I 'm Daniele Di Grandi. \p Hello and Hi to everyone."

Suppose to: TOKENIZEDBY " "
```

```
main-list :: ["Hi", ",", "I'm", "Daniele", "Di", "Grandi", ".", "\p", "
    Hello", "and", "Hi", "to", "everyone", "."]

>>> A: SUBSTRING (
        WITH 0 LEN
        TOKENIZEDBY "\p"
        WHERE
        A: POS 0
        ENDQUERY)[0]
[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0]

>>> A: SUBSTRING "Hello to anyone"
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0]

>>> A: SUBSTRING "an" $this
[0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0]
```

A.10 SQUARE BRACKETS

```
main-list :: ["Hi", ",", "I'm", "Daniele", "Di", "Grandi", ".", "\p", "
    Hello", "and", "Hi", "to", "everyone", "."]

>>> ENDQUERY)[0]
"Hi"

>>> ENDQUERY)[-1]
"."

>>> ENDQUERY)[0-4]
"Hi"
","
"I'm"
"Daniele"

>>> ENDQUERY)[2-3]
"I'm"

>>> ENDQUERY)[0, 4, 6]
"Hi"
"Di"
"."
```

```
>>> ENDQUERY)[0, -1]
"Hi"
"."
```

A.11 DEFINE_PROBABILITY

See example provided when DEFINE_PROBABILITY has been described in Section 4.2.5.

A.12 SORT

Consider the second element of the tuple as the probability value of the first element being the correct metadata:

```
main-list :: [("SUNITINIB", 0.7), ("BIOGARAN", 0), ("37,5", 0.4), ("mg",
    0.5), ("SUNITINIB BIOGARAN", 1), ("BIOGARAN 37,5", 0.2), ("37,5 mg"
   , 0.5)]

>>> SORT "descending"
[("SUNITINIB BIOGARAN", 1), ("SUNITINIB", 0.7), ("mg", 0.5), ("37,5 mg",
    0.5), ("37,5", 0.4), ("BIOGARAN 37,5", 0.2), ("BIOGARAN", 0)]

>>> SORT "ascending"
[("BIOGARAN", 0), ("BIOGARAN 37,5", 0.2), ("37,5", 0.4), ("mg", 0.5), ("
   37,5 mg", 0.5), ("SUNITINIB", 0.7), ("SUNITINIB BIOGARAN", 1)]
```

B List of Medicines (brand names) Considered in the Experiments

Abraxane	Ceplene	Imnovid	Lynparza	Rubraca	Vitrakvi
Adcetris	Cometriq	Inlyta	Mekinist	Sarclisa	Vizimpro
Afinitor	Cotellic	Iressa	Mektovi	Sprycel	Votrient
Alunbrig	Cyramza	Jakavi	Mepact	Stivarga	Xalkori
Arzerra	Dacogen	Javlor	Nerlynx	Tafinlar	Xaluprine
Atriance	Darzalex	Jevtana	Nexavar	Tagrisso	Xofigo
Avastin	Erivedge	Kadcyla	Ninlaro	Talzenna	Xospata
Ayvakyt	Evoltra	Keytruda	Nubeqa	Tarceva	Xtandi
Bavencio	Farydak	Kisqali	Odomzo	Tasigna	Yervoy
Beromun	Firmagon	Kymriah	Oncaspar	Tecartus	Yondelis
Besponsa	Foscan	Kyprolis	Onivyde	Teysuno	Zaltrap
Besremi	Fotivda	Lartruvo	Opdivo	Thalidomide Pharmion	Zejula
Blenrep	Gazyvaro	Ledaga	Perjeta	Torisel	Zelboraf
Blincyto	Giotrif	Lenvima	Phesgo	Tyverb	Zydelig
Bosulif	Halaven	Libtayo	Pixuvri	Vargatef	Zykadia
Braftovi	Iclusig	Lonsurf	Polivy	Vectibix	Zytiga
Cabometyx	Imbruvica	Lorviqua	Removab	Venclyxto	
Caprelsa	Imfinzi	Lutathera	Rozlytrek	Vidaza	