DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE COURSE IN
Computer Engineering

Reproducibility and Generalization of a Relation Extraction System for Gene-Disease Associations

Supervisor: Prof. Gianmaria Silvello

Graduate Student: Laura Menotti

Co-supervisor: Dr. Stefano Marchesin

ACADEMIC YEAR 2021 – 2022

Date of graduation October 10, 2022

Abstract

Biomedical literature is a rich source of information on Gene-Disease Associations (GDAs) that could help physicians in assessing clinical decisions and improve patient care. GDAs are publicly available in databases containing relationships between gene/miRNA expression and related diseases such as specific types of cancer. Most of these resources, such as DisGeNET, miR2Disease and BioXpress, include also manually curated data from publications. Human annotations are expensive and cannot scale to the huge amount of data available in scientific literature (e.g., biomedical abstracts). Therefore, developing automated tools to identify GDAs is getting traction in the community. Such systems employ Relation Extraction (RE) techniques to extract information on gene/microRNA expression in diseases from text. Once an automated text-mining tool has been developed, it can be tested on human annotated data or it can be compared to state-of-the-art systems.

In this work we reproduce DEXTER, a system to automatically extract Gene-Disease Associations (GDAs) from biomedical abstracts. The goal is to provide a benchmark for future works regarding Relation Extraction (RE), enabling researchers to test and compare their results.

The implemented version of DEXTER is available in the following git repository: https://github.com/mntlra/DEXTER.

Contents

Abstract **i**

1 Introduction **1**

2 Background **7**

2.1 Named Entity Recognition and Entity Linking 8

2.2 Relation Extraction 16

2.3 Biomedical Relation Extraction 23

3 Original System **31**

3.1 System Architecture 31

3.2 Comparison Patterns 38

3.3 Results 41

4 SpaCy and Text Processing **45**

4.1 Input Handling 47

4.2 Text Processing Module 49

5 RE Module **55**

5.1 Rules Definition 58

5.1.1 TypeA Patterns 59

5.1.2 TypeB Patterns 71

5.2 Module Overview 78

6 Information Extraction **83**

6.1 Entity Detection and Phrase Typing 83

6.2 Argument Filtering and Extraction 86

7 Results and Discussion 95

7.1 Materials and Methods 95

7.2 Results 98

8 Conclusion 107

References 110

Acknowledgments 120

Chapter 1

Introduction

Discovering the relationships between genes and diseases is getting traction in the community since Gene-Disease Associations (GDAs) can help physicians in assessing clinical decisions and improve patient care. In fact, genetics plays a role in all diseases, for example it can determine if an individual is more likely to develop a specific type of cancer or on the contrary it could render the same individual less susceptible to another disease. For this reason, understanding genetics and its variation in the human population is crucial to understand disease processes thus provide the foundation for curative therapies, beneficial treatments and preventative measures [1]. An example of such information is the correlation between some variations of the BRCA1/2 gene and breast cancer. In fact, patients that present certain variations of the BRCA1/2 gene have an increased risk of developing several cancers, specifically breast and ovarian cancer. This type of information can lead physicians to perform suitable prevention measures, such as more frequent screenings or more advanced tests. To provide easy access to information, GDAs are publicly available in databases containing relationships between gene/miRNA expression and related diseases. These resources comprise DisGeNET [2], miR2Disease [3] and BioXpress [4, 5]. DisGeNET is one of the largest platforms of human gene-disease associations currently available, where data comes from different types of source databases containing both expert curated resources and results from text mining tools. On the other hand, miR2Disease is completely human curated and it aims to provide microRNA-disease relationships. Most of these resources include manually curated

data from publications that are expensive to obtain and cannot scale to the exponential increase of biomedical literature. For this reason, developing automated tools to identify GDAs is getting traction in the community [6]. Such systems employ Relation Extraction (RE) techniques to extract information on gene/microRNA expression in diseases from text.

Relation Extraction (RE) is an Information Extraction (IE) task where we extract relational triples from natural language text which can automatically populate a database or a Knowledge Base (KB). This task has risen in popularity mainly for two reasons. Firstly, a huge proportion of of human knowledge can be expressed in the form of Resource Description Framework (RDF) triple, that is a tuple of entity-entity-relation, and a vast collection of such triples allows for automatic creation of a KB [7]. Secondly, there are abundant applications for KBs containing RDF triples, for example *Freebase* which is the foundation of the Google Knowledge Graph. Therefore, RE finds relational facts from plain text. For example, consider as input sentence *"Paris is the capital of France."*, in this case an RE model would output the triple *(Paris, France, capital_of)* where *"capital_of"* is the relation between the two entities *"Paris"* and *"France"*. RE requires two additional tasks to correctly extract relations between entities: Named Entity Recognition (NER) and Entity Linking. NER aims to recognize portions of text that correspond to specific semantic types that could be Person (PER), Location (LOC), etc. Consider the sentence we introduced above, in this case an NER model would identify both *"Paris"* and *"France"* as entities of type Location (LOC). This task can be both a standalone tool for IE and it is crucial for many applications in NLP such as RE, Text Understanding and Question and Answering (Q&A). Many approaches has been employed to solve this task, including rule-based models and unsupervised techniques, however the dominant approach applies Deep Learning. Entity Linking is the task of linking entity mentions with their corresponding entries in a KB [7]. This tool can facilitate many different tasks such as KB population, Q&A and information integration. In the context of RE, if we develop a system to extend a database, it is required to have a mapping between entities newly detected and the existing entries in the database of reference before the new relations are inserted [8]. Typically, Entity Linking is preceded by a NER module, where named entities are

identified.

Once an automated text-mining tool that extracts relations from text has been developed, it can be tested on human annotated data or it can be compared to state-of-the-art systems. In this context, the state of the art is DEXTER, a rule-based system that extracts gene/microRNA expressions in diseases from biomedical abstracts [9]. It has been published in *Database: The Journal of Biological Databases and Curation* in 2018 and has attracted 11 citations so far. To the best of our knowledge it is the most recent system for RE on Gene-Cancer Associations. DEXTER takes as input biomedical abstracts from PubMed and extract relevant information such as the correlation between genes or miRNAs and diseases. The system also classifies sentences as TypeA or TypeB based on the number of entities found. In particular, TypeA sentences are comparative phrases where gene expression is contrasted between two different samples or conditions while in TypeB sentences there is no explicit comparison. One of the main objectives of the original paper is to create a tool to automatically extend expression databases like BioXpress [4, 5], that contains gene/miRNA expressions associated with cancer. DEXTER is developed both in Python and Java and it mainly employs the Stanford CoreNLP toolkit [10]. The system is based on several modules, each dealing with a different part of the computation. In the first step, namely *Text Processing module*, the title and text of a medline abstract are tokenized and the abstract is split into sentences. Subsequently, each sentence is parsed in order to produce a Standard Dependency Graph (SDG) [11], that provides a tree-like representation of grammatical relations between words in a sentence. In the *Relation Extraction module*, Semgrex patterns are used to extract relevant components by matching relations in the SDG returned by the previous module. The parsed sentence is also taken as input from the *Entity Detection and Phrase Typing module* that extract gene and disease mentions from the sentence using annotations from PubTator [12]. Finally, both the output form the Relation Extraction and the previous module are taken as input by the *Argument Filtering and Extraction module* where we check that the extracted components respect the task type, i.e. the compared aspect is a gene or miRNA and the compared entities are diseases or samples and we extract the relevant information that can be used to populate a database like BioXpress [4, 5].

Unfortunately, DEXTER's source code is not publicly available hence researchers rely only on data provided by the authors to evaluate newly developed systems.

In this work we reproduce DEXTER, so that we can provide a benchmark for future works regarding RE, enabling researchers to test their newly developed systems on a reliable baseline. Our version of DEXTER is publicly available in a GitHub repository[1]. We decided to develop the system as an end-to-end application that takes as input biomedical abstracts and returns relevant information on the expressed gene/microRNA and the associated disease. We preserved the overall block architecture and we made some changes in each block to enable a seamless integration of the different modules. Our system is entirely developed in Python, therefore we could no longer use the Stanford CoreNLP toolkit, which supports Java. Instead, we used the SpaCy library [13], which is an open-source library written in Python that provides some utility functions to process text efficiently and effectively. Spacy was released in 2016 by Matthew Honnibal, who wanted to provide small companies an accurate and fast pipeline for NLP tasks that is suitable for production use [14]. SpaCy comes with some trained pipelines that provide support for several standard components, i.e. SpaCy tokenizer, parser, NER and tagger, but it also allows for custom components definition. We added a custom component to the SpaCy pipeline to expand entity mentions using hand-craft rules which will be useful when extracting information from the components by the RE module. We used data provided by the authors in BioXpress[2] as ground truth to assess the accuracy of our system. In particular, we evaluated our system performance based on the percentage of parsed sentences and the correctness of the results in terms of correct gene expression level and correct sentence type.

The rest of this work is organized as follows. Chapter 2 provides an overview of some tasks that are employed in the development of DEXTER. In particular, we describe the task of Relation Extraction (RE) in general and provide some definition and methods of two related task, namely Named Entity Recognition (NER) and Entity Linking. Finally we focus on Biomedical Relation Extraction and the main

[1]https://github.com/mntlra/DEXTER
[2]https://hive.biochemistry.gwu.edu/bioxpress

methods that have been used to solve such task. Chapter 3 describes the original system that was developed in [9], with particular interest in the system architecture and its objectives. Chapters 4, 5 and 6 focus on the implemented version of DEXTER. In particular, Chapter 4 focuses on the input handling step and the *Text Processing* module, Chapter 5 describes the *Relation Extraction module* and the dependency parsing task in general while Chapter 6 focuses on the last two modules, namely *Entity Detection and Phrase Typing* and *Argument Filtering and Extraction* modules. In Chapter 7 we evaluate our implementation of DEXTER using data provided by the authors of the original paper in BioXpress. We focus our evaluation in terms of how many sentences we are able to parse and the correctness of our output. Then we also perform an error analysis where we compare the dependency parser we are using and the one used in the original paper and we also investigate the impact of PubTator in our system's performance. Finally, in Chapter 8 we conclude our work with some final remarks.

Chapter 2

Background

This chapter presents an overview of the task of relation extraction. We describe the task in general and provide some useful definitions of related tasks. Next we discuss some previous works on RE in general and finally we focus on the application of relation extraction in the biomedical domain, namely Biomedical Relation Extraction (BioRE). Part of this chapter is based on the lecture about Relation Extraction with distant supervision from the Stanford course of Natural Language Understanding held in 2019[1], that is available on YouTube, and on some slides from the lecture about Information Extraction from the course of Natural Language Processing held at the University of Washington in 2013[2].

Relation Extraction (RE) is an Information Extraction (IE) task where we extract relational triples from natural language text which can automatically populate a database or a Knowledge Base (KB). This task is getting traction in the community mainly for two reasons. Firstly, a huge proportion of human knowledge can be expressed in the form of Resource Description Framework (RDF) triple, that is a tuple of entity-entity-relation [7]. A vast collection of triples allows for the automatic creation of KBs. Secondly, there are abundant applications of KBs like Freebase, that is the foundation of the Google Knowledge Graph. Therefore, one aim of RE is to find relational facts from a plain text, organizing unstructured information into structured information. For example, from the sentence *"SpaceX was founded by Elon*

[1] https://web.stanford.edu/class/cs224u/2019/
[2] https://courses.cs.washington.edu/courses/cse517/13wi/

Musk in 2002" an RE model should output the triple *(Elon Musk, SpaceX, founded_by)*. RE usually requires two additional tasks as a pre-processing step: Named Entity Recognition (NER) and Entity Linking.

2.1 Named Entity Recognition and Entity Linking

As we said before, RE aims to extract relations between entities. In order to do so we firstly need to detect the named entities mentioned in the input text. Such operation is usually performed by a *Named Entity Recognition (NER)* system. Subsequently, we may want to associated each named entity with its representation in an ontology or knowledge base. This task is defined as *Entity Linking*. Therefore, NER and Entity Linking often acts as a pre-processing step in an RE system. In this section we describe these two tasks and provide a small survey of the main techniques used to develop NER models and Entity Linking systems.

Named Entity Recognition

Named Entity Recognition (NER) is the task of finding spans of text that corresponds to specific semantic types that could be person (PER), location (LOC) or organization (ORG). NER acts both as a tool for IE and as a useful pre-processing step in lots of Natural Language Processing (NLP) applications such as sentiment analysis, text understanding, question answering and KB creation [7]. Each span of text is called Named Entity (NE) and we generally divide NEs into two categories. *Generic NEs* usually refers to general concepts that can be applicable to a great variety of domain, such as person and location. On the other hand we have *domain-specific NEs* that, as the name suggests, are specific for a domain. Examples of domain-specific NEs are proteins, enzymes, genes and diseases in the biomedical domain.

We use the formal definition of the NER problem reported in [15]. Given a sequence of tokens $s = <w_1,..,w_N>$, NER systems output a list of triples $<I_s,I_e,t>$, where each triple is a NE detected in the text. The first two elements of the triple are the start ($I_s \in [1, N]$) and the end ($I_e \in [1, N]$) indexes of the span of text corresponding to the mention. The last element t is the entity type, which usually belongs to a

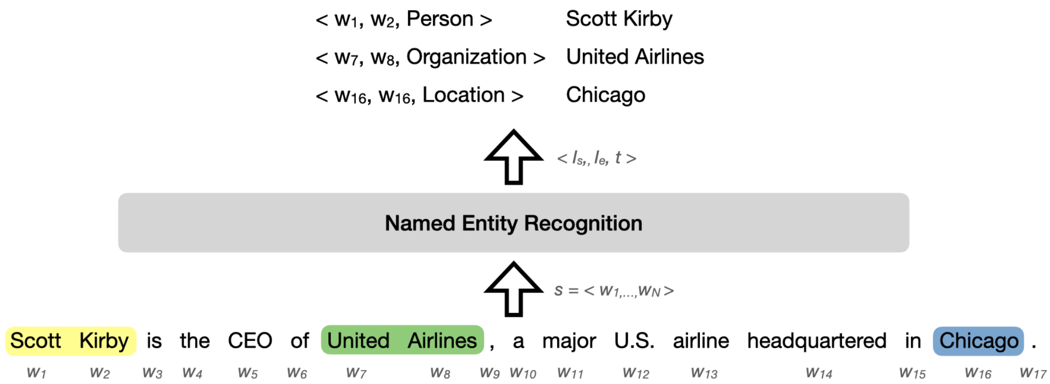


Figure 2.1: Example of NER task. We identified three mentions belonging to three different categories. We highlighted each NEs in the text with a different colour.

predefined set of categories. Figure 2.1 shows an example of the NER task. Here we extracted three NEs, namely the person *Scott Kirby* which spans between w_1 and w_2, the organization *United Airlines* identified in tokens w_7 and w_8 and the location *Chicago* that starts and ends with token w_{16}. As we just said, the set of categories is predefined in advance hence we can distinguish between *coarse-grained NER* and *fine-grained NER*. In the first case, we have a small set of categories and we output one category per NE. On the other hand, in the second task the set of entity types is much larger and we allow for multiple types assignment.

NER systems are usually evaluated by comparing the results against human annotations, hence high quality tagged corpora are critical for assessing the systems performance. Evaluation involves the usual metrics such as Precision, Recall and balanced F-Score. Precision measures the ability of the system to present only correct results while Recall assess whether the system recognizes all entities in a corpus. The balanced F-score is the harmonic mean of precision and recall. We recall the definition of Precision and Recall based on True Positive (TP), False Postive (FP) and False Negative (FN).

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN}$$

NER models need both to identity the entity boundaries and the entity types, therefore we can perform two kinds of evaluation: *exact-match* or *relaxed match*. In the first case we consider correct, i.e. as a TP, each NE where both boundaries and type match

ground truth. For this reason, FP are the entities that are recognized by the NER model but do not match the ground truth. This can either be due to wrong span boundaries or wrong entity type or both. On the other hand, FN are missed entities, that are NEs annotated in the ground truth that are not recognized by the model. In the second case we relax the previous assumption and we consider correct each entity assigned to its correct type where there is an overlap between ground truth boundaries and the recognized span. If we consider the example in Figure 2.1, we would consider correct output <w_1, w_5, *Person*> because it overlaps with the ground truth boundaries, that are <w_1, w_2>.

There are four main techniques applied in NER: Rule-based approaches, Unsupervised learning models, Feature-based supervised learning methods and Deep-learning based techniques. *Rule-based NER* relies on hand-crafted rules based on domain specific features and syntactic patterns. These models work effectively when lexicon is exhaustive, however they do not generalize well due to the specificity of the rules and often suffers from low recall.

With the advent of machine learning techniques several models have been proposed for NER task employing both supervised and unsupervised learning techinques. Supervised learning techniques use a labelled datasets to train a model to solve classification or regression tasks. Such models can be trained to learn features about the data, in this case we refer to *Feature-based models*, or they could employ deep learning techniques to learn multiple levels of data representation [16], in this case we talk about *Deep-learning techniques*. The main challenge for supervised systems comes from the need of big annotated datasets for training, which remains time consuming and expensive. To mitigate this limitation, unsupervised learning methods have been proposed.

Unsupervised learning models are trained on unlabelled data and they solve tasks like association, dimensionality reduction and clustering, that is a common approach used for NER. Clustering techniques aims to group together unlabelled data based on their similarities. Clustering-based NER systems resort to context similarity and infer mentions of NEs from text using lexical resources and patterns and statistics computed on a large corpus. Zhang and Elhadad [17] proposed an unsupervised

biomedical NER system that employs terminologies, shallow syntactic knowledge (e.g., noun phrase chunking), and corpus statistics (e.g., inverse document frequency and context vectors). Experiments demonstrate that the model is effective and can be generalized.

As far as *supervised learning* is concerned, NER can be solved as a multi-class classification problem or as a sequence labelling task, where the input is a sequence of words and the output is a sequence of labels. Given annotated data, features are extracted to represent training data. Such features can be word-level features (e.g., case and POS tag), list lookup features (e.g., Wikipedia gazetteer) or document and corpus features. Based on these features, we can apply several machine learning algorithms such as Hidden Markov Models (HMM), Maximum Entropy Models and Conditional Random Fields (CRF).

Another supervised approach employs *Deep Learning (DL)*, where models automatically learn hidden features from data without the need of a feature extractor as for the previous methods. Deep learning is making major advances in many tasks, such as speech recognition, visual object recognition and many other domains such as drug discovery and genomics [16]. DL models are composed of multiple processing layers that transform the representation at one level into a representation at a slightly abstract level. Each layer is typically an artificial neural network that consists of the forward pass and the backward pass. The former computes transformations on the inputs from the previous layer and passes results to a non-linear module before outputting them. The latter computes the gradient of an objective function with respect to the weights of a multilayer stack of modules, which is used to trained the multilayer architecture [16]. DL-based NER models become dominant and improve the state-of-the-art, mainly because of three major benefits of applying DL techniques to NER. Firstly, by means of the non-linear transformation DL-based models are able to learn complex and intricate features from data. Secondly, feature-based approaches require domain expertise and a great amount of engineering skill to design features. Since DL-based methods automatically learn useful representations from raw data, these models saves significant effort on NER feature design. Lastly, we can use possibly complex NER systems since models can be trained in an end-to-end paradigm

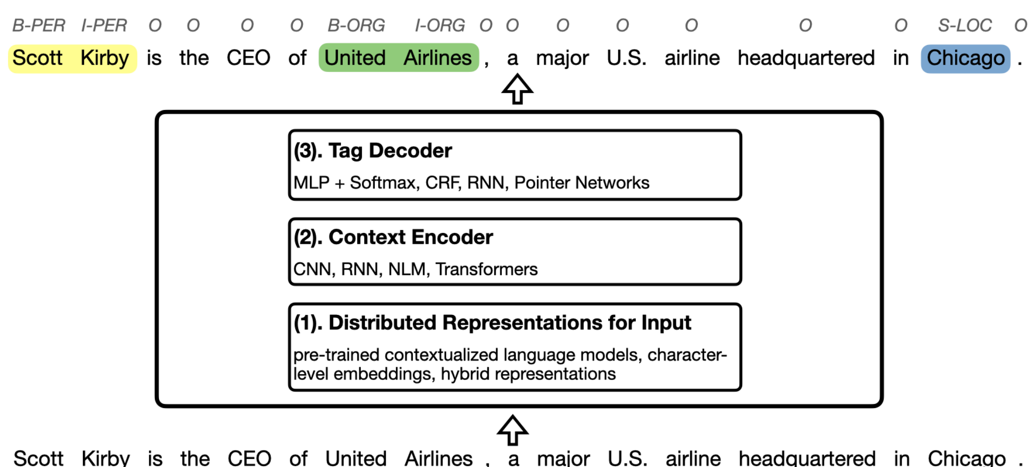


Figure 2.2: DL-based NER general architecture. We also provide an example of the input and the output of the system, using the same sentence as in Figure 2.1. NEs are highlighted with different colours.

by gradient descent. The DL-based NER model general architecture described in [15] comprises three modules, as shown in Figure 2.2. *Distributed representation for input* represents words in vectors where each dimension represents a feature. The most widely used distributed representations are word-level representations (typically a pre-trained language model), character-level representations (extracted using CNN-based or RNN-based architectures) and hybrid representations, where additional information (e.g. linguistic dependencies or lexical similarity) is incorporated. *Context encoder* captures the context dependencies considering the full input sequence. Several architectures are widely used, comprising Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Recursive Neural Networks and Neural Language Models (NLM). Transformer-based architectures, such as pre-trained language model embeddings using Transformer, are faster and achieve more effective performances. Since the success of a NER system heavily depends on its input representation integrating or fine-tuning pre-trained language model embeddings is becoming a new paradigm for neural NER. *Tag Decoder* predicts tags for each token in the input sentence based on the context representation of the previous module. Each token is predicted with a tag indicating the position of the token with respect to the considered named entities. Such tags are B-(begin), I-(inside), E-(end), S-(singleton) or O-(outside). Four architectures are available to carry out the task: Multi-Layer Perceptron (MLP) + Softmax, CRF, RNN and

Pointer Networks. The most common choice for tag decoder is CRF, that is a random field globally conditioned on the observed sequence, which achieves state-of-the-art performance.

On the whole, what architecture to choose is data and domain task dependent. As far as data is concerned, end users could consider training models with RNNs from scratch or fine-tuning contextualized language models when a large dataset is available. Otherwise, unsupervised methods, i.e. transfer learning, is preferable. On the other hand, many pre-trained off-the-shelf models are available for newswires domain whereas for specific domains, for example medical NER, fine-tuning general-purpose contextualized language models with domain-specific data works effectively.

Entity Linking

Entity Linking is the task of associating a mention in text with its representation in an ontology or KB [7]. This task is employed in many different tasks such as knowledge base population, question answering and information integration. For the sake of our work, if we develop an RE system to extend a database, it is required to have a mapping between entities detected and the entries in a database before the new relations are inserted [18]. The main challenges for entity linking are name variations and entity ambiguity. *Name variations* refers to the different forms an entity could appear in a text. For example, *"University of Padua"* could also be mentioned as its abbreviation, i.e. *"unipd"*, or the named entity *"New York"* could be referred to as *"Big Apple"*. It is important that an entity linking system identifies the correct mapping for mentions of various forms. On the other hand, the same mention could represent different named entities. For example, the named entity *"Apple"* could refer to the American technology company founded by Steve Jobs or to a fruit. Therefore, an entity linking system must be able to disambiguate the mentions in a text and identify the correct mapping. For this reason, this task is also called *Named Entity Disambiguation (NED)*. Typically, the task of entity linking is preceded by a NER module, where we identify the named entities in the text.

We use the task definition and general architecture of an entity linking system presented in [18]. Given a KB containing a set of entries E and a natural language text

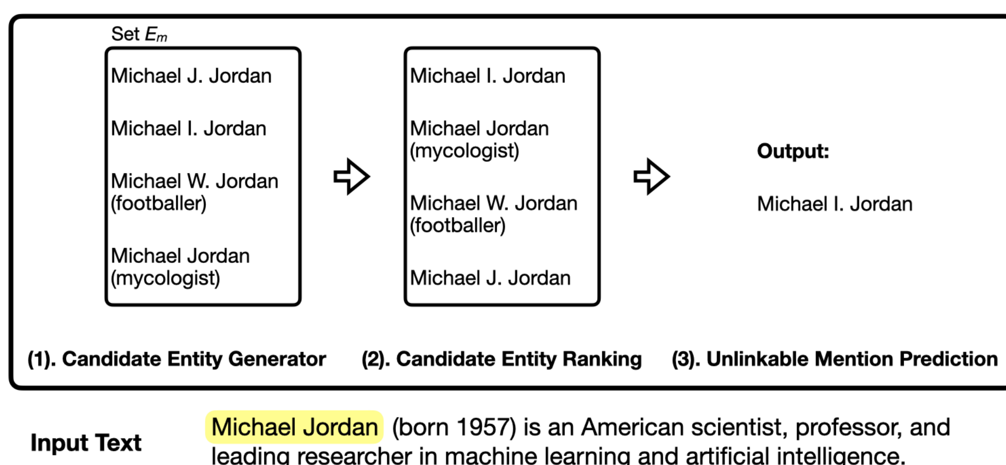


Figure 2.3: Example of a typical Entity Linking system. We highlighted in yellow the considered entity mention in the input text and we provide the example computation of the three modules for the entity mention *"Michael Jordan"*. In the last module we output the name of the corresponding entity in the KB.

where we identified a set of entity mentions M, the goal of an entity linking system is to return for each mention $m \in M$ the corresponding entity $e \in E$ in the KB. If the knowledge base does not contain the entity e for mention m we usually label the mention as *NIL*, i.e. unlinkable mentions. A typical entity linking system consist of three modules: Candidate Entity Generation, Candidate Entity Ranking and Unlinkable Mention Prediction. An example of an entity linking system architecture is shown in Figure 2.3.

In the *Candidate Entity Generation* module, for each mention the entity linking system aims to return a set E_m which contains all possible entities that the mention could refer to. To achieve this goal, approaches are mainly based on string comparison between the text mention and the name of the entity inside the KB of reference. The main approach employs *Name dictionary* based techniques, where we build an offline name dictionary D comprising various names and their possible mappings. Specifically, dictionary D is a <*key, value*> mapping where for each $k \in$ *key* the *value* column is a set of entities in the reference KB that could be mentioned under the name k. The dictionary can be constructed via features from Wikipedia or exploiting query click logs and web documents to find entity synonyms. To match entity mentions against the dictionary keys, we can use exact matching or some common rules. For

example, we can check if the entity name share several common words with the mention or if it is contained or contains the mention. For each entity mention $m \in M$, if m matches any key in the dictionary, either by exact match or by some rules, the entities contained in the *value* set corresponding to the matched key are added to the candidate entity set E_m. Entity mentions could be acronyms or part of their full name hence one category of entity linking systems use some expansion techniques to identify expanded variations. Such techniques include heuristic-base methods or supervised learning models. Finally, candidates can be retrieved by exploiting web search engines, such as Google, hence leveraging the whole web information.

The *Candidate Entity Ranking* module ranks the entities in E_m based on the most likely link for mention m, hence the first entity after re-ranking will be the most likely entity for mention m. Techniques employed for this task can be categorized as *supervised* or *unsupervised ranking*. The first set of methods trains models using annotated data and learns how to map the proper entity to each mention. We recall annotated training sets are usually manually curated hence it can problematic to provide a training set big enough to train an accurate model. Models based on supervised learning usually employ Binary Classification methods, Learning To Rank, Probabilistic models, Graph Based approaches or model combination. On the other hand, *unsupervised methods* do not need annotated data therefore they can be less labor-intensive and costly. Such methods comprise Vector Space Models (VSM) and Information Retrieval (IR) based models. Candidate Entity Ranking methods can also be categorized based on whether ranking is performed indipendently for each mention (i.e., *Indipendent Ranking*), if it considers the whole document for ranking each mention (i.e., *Collective Ranking*) or if it takes into account other documents (i.e., *Collaborative Ranking*). Ranking is usually based on some features that can be extracted from the mention itself or from the context in which the mention appears.

In the *Unlinkable Mention Prediction* module we validate whether the top-ranked entity identified in the previous step is the target entity. If so, we return the entity otherwise we return *NIL*. Many methods assume that the KB contains all the mappings hence they ignore the unlinkable entity problem. In this case, if the candidate set E_m contains some elements the first element is returned as the linked entity without

any further checks, otherwise we return *NIL*. On the contrary, other methods predict the unlinkable entity mention through a *NIL* threshold. In this case each entity is assigned a score and if such score is lower than the threshold τ, which can be learned from training data, then the entity is not assigned to the mention and we return *NIL*. There are other supervised models that predict the unlinkable entity mention treating it as a binary classification problem. Some other Entity Linking systems, instead, incorporate the unlinkable mention prediction in the previous module.

As far as evaluation is concerned, we use the standard performance measures such as Precision, Recall, Accuracy and F1-score. In particular, the Precision considers all entity mentions that *are linked* by the system and determines how many of them are correct. On the other hand, Recall takes into account all entity mentions that *should be linked* and counts how many of them are correctly linked by the system. The two measures are used together to determine the F1-score, which we already defined when we described the NER task. To sum up, there are many methods for each module used in the Entity Linking task however it is still unclear which one works best. This is due to the fact that Entity Linking systems performance highly depends on the dataset and domain we are considering.

2.2 Relation Extraction

In this section we provide a brief description of RE in general and the main approaches used to achieve this task, with particular attention to *Deep Neural Network (DNN)* methods such as *Distant Supervision*.

As we already mentioned, RE aims to extract relational triples from natural language text so that we can convert unstructured data, such as text, into structured data, e.g. tables or RDF triples. This task allows to automatically populate KBs thus avoiding manual curation that can be time-consuming and laborious when dealing with large amount of data. More precisely, we extract two entities from a sentence that are linked together by a relation, that belongs to a set of predefined relations. For example if we consider the sentence *"John Smith founded HCC S.P.A."*, a RE model identifies the entities *John Smith* and *HCC S.P.A* and the relation *founded_by* from a

pre-defined set. Therefore, the model will output the triple *(HCC S.PA., John Smith, founded_by)*. It is clear that to extract this type of information from natural language we need to recognize the entities mentioned in the text and check if their type matches the relation of interest requirements. In fact, considering the previous example, the relation *founded_by* is expressed between an entity of type *Person (PER)* and one of type *Company/Organization (ORG)*. Such subtask can be achieved by means of some NER methods together with Entity Linking, that were both described in section 2.1. RE can be used in a large number of tasks, for example in Question Answering (QA) or Text Mining, and in a variety of domains including finance, biology and medicine [8]. We will provide a detailed description of RE in the biomedical domain, namely BioRE, in Section 2.3. There are five main approaches for RE: hand-built patterns, semi-supervised learning, unsupervised learning, supervised learning and distant supervision.

The first technique that was largely used in 60s and 70s consists on writing patterns that express the relation we are trying to identify. For example, we can search over a large corpus for occurrences of the pattern *"X was founded by Y"* for relation *founded_by* and extract the (X, Y) pairs from sentences that match the pattern. If we consider sentence *"SpaceX was founded by Elon Musk."*, the (X, Y) pair we retrieve is *(SpaceX, Elon Musk)*. Instead of using basic patterns like the one above, one can define patterns as Regular Expressions (regex) to integrate minor variations in the sentence text. DEXTER, the system we are reproducing in this work, is based on Semgrex Patterns that are patterns for matching nodes and edges of a dependency graph. For this reason, the system we are reproducing falls into the *rule-based* category. The problem with this approach is that patterns are very specific and do not generalize well, hence this methods results in having high precision but low recall.

After the advent of Machine Learning (ML), RE task has been reformulated as a classification task so that *Supervised Learning* could be employed through some standard techniques that, unfortunately, require lots of labelled data. In order to mitigate this issue, *Semi-Supervised* methods and distant supervision have been used, since they require smaller labelled datasets. Another set of methods that do not require any labelled data exploits *Unsupervised Learning*, where RE is usually treated

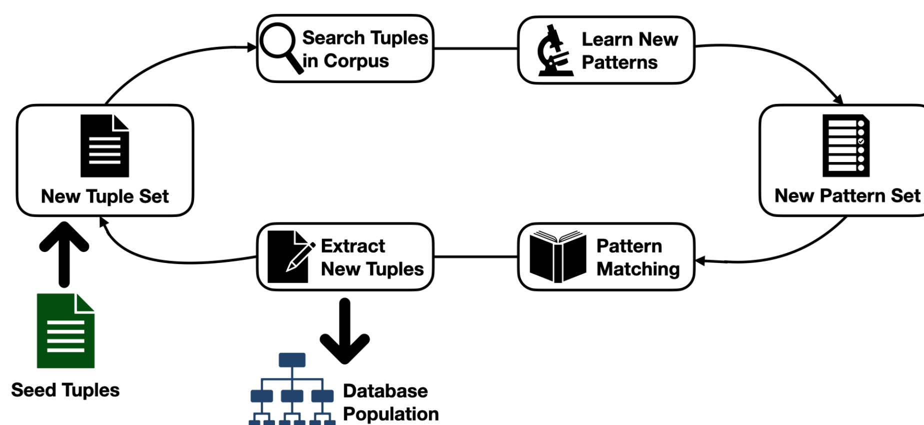


Figure 2.4: Semi-Supervised Relation Extraction using bootstrapping. We start by taking some seed tuples and iterate over the blocks depicted in the figure. At each iteration the new extracted tuples are added to the database of reference.

as a clustering problem.

In *Semi-Supervised* methods the main idea is to find some seed tuples with high confidence and use a bootstrapping algorithm to extract patterns with the above-mentioned tuples from an unlabeled corpus. We show how semi-supervised RE works in Figure 2.4. In particular, bootstrapping algorithms take the entities in the seed tuple and find sentences that comprise both entities. Then, from all such sentences we extract the context to learn general patterns that can be used to search for additional tuples [7]. Some models employing bootstrapping techniques are DIPRE (Dual Iterative Pattern Relation Expansion) [19], that retrieves (author, book) pairs, Snowball [20], KnowItAll [21] and TextRunner [22]. Contrarily to supervised ML methods, these methods do not require large datasets of labelled data that can be expensive to obtain. However, they are sensitive to the original set of seeds, in fact we can only identify relations that are present in the seeds and such models usually have lots of parameters to be tuned.

As we mentioned before, *unsupervised methods* do not need any labelled data nor any list of relation. This task is often referred to as *Open Information Extraction (Open IE)*. A model based on this approach was proposed in [23] and it was based on the assumption that context information of different entities pair conveying the same relation are similar. Therefore, in this model authors extracted an entity pair and its context and then cluster entity pairs according to the context. Finally, they labelled

each cluster of entity pairs with the suited relation type. Another model proposed in 2011 is *ReVerb* [24], where relations are extracted by checking each verb in the sentence and finding noun phrases that satisfy some lexical constraints based on a dictionary D, which is populated during the offline run of the algorithm. Then, a classifier assigns a confidence value to each relation triples retrieved in the previous steps. This confidence classifier is trained by hand labelling 1000 random web sentences. The advantage of methods employing unsupervised techniques is that they can handle large number of relations without pre-define them. However, most of these models focus on relations conveyed by verbs, hence we lose nominal relations. In addition, the clustering results generated by such models are generally broad [8].

Evaluating Semi-Supervised and Unsupervised methods is not easy since there is no ground truth comprising labelled data. For these methods we generally approximate precision by drawing a random sample of relations from the output and manually check the accuracy of each extracted relation [8].

Supervised methods consider RE as a multi-class classification problem. These models have much better performance, especially on recall, but they rely on manually annotated data. Supervised approaches can be divided into two types: *feature based* and *kernel based*. The former learns how to extract relations based on some syntactic features, including the types of the two entities and the path in the parse tree containing the two entities, and semantic features, that include the path between the two entities in the dependency parse [25]. On the other hand, kernel approaches are based on the definition of a kernel function that computes the similarity score between the two entities. Feature-based methods performances depend on the defined feature set hence without a proper feature selection it is difficult to improve a model. On the hand, kernel-based approaches heavily depends on the performance of the defined kernel function. Supervised RE can be evaluated as any other ML task exploiting Supervised Learning, i.e. computing performance measures on the test set.

Distant Supervision is another supervised method that does not require a lot of annotated data. The main idea is to use relational triples from existing KBs to automatically extract relations from the corpus. In other words, when a sentence and the KB of reference refer to the same entity pair, this approach marks the two entities

in the sentence with the same relation found in the KB. For example, if a sentence contains the entities *Steve Jobs* and *Apple Inc.* and in the KB we have a triple *(Steve Jobs, Apple Inc., founded_by)*, then the system will output the relation *founded_by* for the two entities. Distant Supervision can be used to produce a larger training dataset thus it is usually used before training a learning algorithm. This approach overcome the problem of having a labelled dataset large enough to train an effective model, however it suffers from two major drawbacks. Firstly, not all sentences where the same entity pair appears will convey the same relation. As a consequence, we inject noisy examples in the dataset we are building hence it is more difficult to train a learning algorithm on this data. However, this disadvantage is outweighed by the benefit of having much more data, hence the model still performs better than other approaches. Lastly, it can only be used to extend existing KBs because it relies on existing triples hence this strategy works only for relations for which there is at least a triple in the KB.

Relation Extraction using Deep Neural Networks (DNNs)

Recently, DNNs have been employed in RE resulting in improving the performance of the overall task. In fact, DNN-based models can automatically learn features from text instead of manually design them and they can also achieve better results with fewer features. We restrict our description to DNN-based Supervised RE and DNN-based Distant Supervision RE.

DNN-based RE framework usually include four components: data sets, sentence representation, feature extraction and classifier. Firstly, supervised methods uses manually annotated data that have high accuracy and low noise but are smaller in size. On the other hand, distant supervision employ data created by aligning entities detected in the corpus with a KB of reference. In this case the data set we use has low accuracy, high noise but it is bigger in size and can be applied to different domains. Overall, in RE tasks constructing training triples takes a crucial part. In any NLP task, *sentence representation* is important to obtain good results. In particular, words and sentences are usually represented as vectors that can be sparse or dense based on the method we use to obtain them. In DNN-based RE models, sentence representation

comprises word representations that are the combination of word vectors, that can be obtained using methods like word2vec or GloVe, and positional embeddings, that express the positional relation between words. The *feature extraction* component handles the extraction of high-level features from the sentence representation. In fact, with the annotated data sets, these methods can train a feature extractor to extract useful information that can then be used to detect relations among entities. Finally, using the features extracted by the previous components and a set of predefined relations, the *classifier* is trained to output relations between entity pairs.

Supervised methods employing DNNs can be divide into two sub-categories based on the evolution of the model structure: structure oriented and semantic oriented. The former improves the feature extraction component by changing the structure of the model while the latter improves the semantic representation of text by focusing on the internal association of text [8]. Several structures can be employed to design a DNN-based supervised model, comprising CNNs, RNNs or Long Short-Term Memory (LSTM) and mix-structures. CNN-based models have achieved excellent results. About structure-oriented architectures, the first RE model based on CNN was proposed by Liu et al. in [26], where the sentence was split into a series of word vectors that was fed to a CNN and a softmax output layer to output the classification probability. To improve the feature selection, later models introduced multiple filters and max-pooling modules for CNN while PE method was employed to extract information with fewer NLP toolkits. CNN-based models use filters yet fixed size filter CNNs ignore global features and focus only on the local ones. For this reason, multiple window size filter CNNs were used in later architectures. If we consider semantic oriented models, most architectures employ attention mechanism, in particular Multi-Level Attention CNNs considers both the semantic information at the word level and sentence level and allows for end-to-end training on task-specific data. However, CNN-based methods rarely consider global features and time sequence information, especially for long-distance dependency between entity pairs. RNN\LSTM models can alleviate these issues. In fact, to learn relations within a long text one can use a bi-directional RNN architecture that combine the output of each hidden layer and represent features at the sentence level. Then,

at the end of the computation, the model does a max-pooling operation to pick up some trigger word features for prediction. The problem with RNN is that it suffers from gradient explosion thus LSTM models were proposed to solve this problem by using the gate mechanism. Using bi-directional long short-term memory networks (BLSTM) to obtain lexical features and using word embedding as input features was enough to achieve state-of-the art results. Overall, these methods improved the performance yet they still have some issues. In fact, they introduced a large number of external artificial features and they have no effective feature filter mechanism. As far as semantic oriented architectures are concerned, an effective model employs the attention mechanism in BLSTM in order to give more weight to the important features extracted by the model. In this case, some models use random weights for features while others consider prior knowledge to determine such weights. Finally, there are several works where these models were combined and they perform better than any other single model. For structure-oriented architectures models were simply combined, e.g. CNN+RNN or BLSTM-CNN, while for semantic-oriented structures models were combined together using attention mechanism. To conclude, both structures improve the RE task in different aspects. In fact, structure-oriented architectures improve feature extraction while semantic-oriented focus on the semantic representation of the sentence. Overall, semantic-oriented models look more effective than structure-oriented ones.

The main obstacle of this type of methods lies on their need for annotated training corpora, that are often insufficient. To solve this problem, *distant supervision* was employed in RE tasks. As we already mentioned, distant supervision is based on the assumption that if two entities are linked together with a certain relation in the KB of reference then whenever we found such entities in a sentence we assume they express the same relation. Since this assumption is too strong and results in adding a lot of noise in the training data two more assumptions were added. The first assumption leads to more accurate results and state that among all sentences that mention the two entities of interest, at least one of them conveys the relation between the two entities we found in the KB. The other assumption, instead, considers more sentence features and affirms that we can found a relation either explicitly or

implicitly from all sentences that mention the two considered entities. Based on these assumptions, research on distant supervision methods follows two research directions. The first one are *Encoder-based methods* where we focus on the sentence encoder and we aim to optimize the RE model and performance. The other one instead focus on *de-noise algorithms* to improve the quality of the data sets. This last direction can be divided into three sub-categories where we focus on different aspect of the task. The first are *representation-based methods* that make use of features in the bag of sentences mentioning the same pair of entities both at word-level and sentence-level. In this case we focus on enhancing the representation of the input data. The second sub-category comprises *knowledge-based methods* where we introduce external knowledge to get additional information about the pair of entities. Lastly, *plug-and-play-based methods* were developed to directly construct a component to reduce the noise of datasets. Overall, distant supervision achieves excellent results in RE tasks, however the problem of noisy data still remains.

In conclusion, both supervised and distant supervision methods improved the performance of RE tasks since they are both beneficial to RE in different aspects. In fact, supervised methods are better suited for tasks related to specific domains while if we want to develop a general-domain RE model distant supervision works better. Nevertheless, there are still many open challenges of RE mainly concerning transfer learning, relationship framework, overlapped multiple relations, relationship reasoning and different presentation of data sets.

2.3 Biomedical Relation Extraction

RE has been widely applied to the biomedical domain since traditional methods, i.e. human annotations, struggle to keep up with the exponential growth of biomedical literature. In fact, RE can be applied in several tasks related to the medical domain such as extracting drug-gene relationships, Gene-Disease Associations (GDA), drug-effect relations etc. As we said before, RE aims to extract relations from unstructured text so that information can then be transformed into a structured form and can be accessed more easily. This can be very useful in many medical applications. For

example, in the radiology oncology domain, extracting relevant information about pre-exisiting conditions or clinical history embedded in clinical notes as free text can help physicians provide better treatment and speed up diagnosis [27]. As we already discussed in previous sections, NER plays an important role in RE tasks hence significant research on Clinical NER has been carried out in the past years. The goal here is to extract entities from clinical texts. Some terms of interest can be diseases, location, medical procedures, genes or drugs. We now provide a brief description of the main methods used to solve BioRE tasks. These methods are very specific to the particular dataset and task we are trying to solve hence it is difficult to assess which method works best since most of them do not generalize well.

Rule-Based models extract relations based on some hand-written rules or patterns. There are several works employ this type of technique however it is not very used nowadays since they have been outperformed by ML-based models. Segura-Bedmar et al. (2011) [28] developed a linguistic hybrid rule-based method to extract drug-disease interaction from clinical texts however the model did not perform particularly well and suffers from a very low recall. Li et al. (2015) [29] instead developed a system to identify medical discrepancies in drugs by matching drug names with their attributes and achieved some good results. Finally, Mahendran et al. (2021) [30] developed a system to perform adverse drug event extraction that finds the adverse drug effects and was able to achieve good performance in terms of macro-average F1-score. Overall, rule-base approaches performance highly depends on how the rules are defined.

Supervised Learning has been widely used in RE as it can achieve excellent results. These methods solve RE as a classification problem and they learn some features from annotated training data to predict whether there is a relation between two entities or not. As we discussed in Section 2.2, these methods need training data annotated by domain experts and this is a big limitation. We can divide these methods into traditional ML and DL-based models and Language Model (LM)-based methods. *Traditional models* heavily depend on well-defined features hence they depend on the feature extraction process. These methods employ algorithms such as Support Vector Machines (SVM), Linear Regression, K-Nearest Neighbour (KNN), node2vec

or they may employ DL architectures like CNN, RNN or autoencoders. SVM-based approach was first used by Swampillai et al. (2011) [31] that extracted relations between entities in different sentences and obtained good results. Sahu et al. (2016) [32] instead developed a decent system using domain-invariant CNN on multiple features with variuos filter length combination. Finally, Kim et al. (2018) [33] built a system to detect relations in the type2 diabetes pathway by using node2vec to learn the features from texts in the networks and outperformed previous works in the same pathway [27]. Overall, ML or DL-based methods have very different performance based on the training data hence there is no model that works best for BioRE tasks in general.

LM-Based approaches have been used in a large number of NLP tasks and they achieve state-of-the-art results. Language models represent features using contextual information of the input sentence therefore RE models based on this technique add a classifier on top of the language model to predict the probability of the relation between a pair of entities. Bidirectional Encoder Representations from Transformers (BERT) [34] introduced by Google in 2019 is the most popular language model by far. Language Models are usually pre-trained and one can adapt them to the specific task by means of fine-tuning. For biomedical clinical texts, there are two BERT-based models trained on domain-specific data, namely BioBERT [35] and ClinicalBERT [36]. The main difference between the two models is that BioBERT is trained on biomedical PubMed corpus while ClinicalBERT is trained on a biomedical corpus and on clinical notes and discharge summaries. Overall, language models achieve better performances than any other models on clinical RE tasks [27]. Wei et al. (2020) [37] investigated how to apply BERT to clinical RE tasks and established that fine-tuning BERT outperformed previous state-of-the-art RE systems in two shared tasks, namely 2018 n2c2 data set and 2010 i2b2 data set. BERT, BioBERT and ClinicalBERT were also used by Mahendran et al. (2021) [30] for extracting adverse drug events. The performance of such models were compared to a rule-based approach based on co-location information and a deep learning approach utilizing CNNs confirming that contextualized LM-based models outperform all other models and obtain state-of-the-art performances.

Unsupervised Learning allows for developing models without the need of annotated data, hence this technique is less expensive and is preferred in some cases. In clinical RE, unsupervised-based approaches usually view RE as a clustering problem however this is the least popular approach since without proper annotations this task is more ambiguous which results in decreased accuracy [27]. Two works using unsupervised learning to extract relations from biomedical texts were proposed by Quan et al. in 2014 [38] and Alicante et al. in 2016 [39]. The former used a clustering algorithm to extract protein-protein interactions and gene-suicide associations while the latter used k-means and applied hierarchical clustering to solve BioRE tasks from Italian clinical records.

To sum up, clinical NER tends to employ DL techniques and it has shown brilliant results. On the other, clinical RE performance depends on the specific task and on the data available however the main research direction is focused on ML and DL approaches and using LMs. Very little research is using unsupervised learning techniques due to the uncertainty in the results generated by such models. The main issue about ML and DL models is due to the limited data available hence LMs like BERT is vastly used and achieve the best results in extracting relations from clinical texts.

Datasets

As we just said, the main issue related to employ ML and DL techniques in BioRE tasks is the need of large amount of annotated data to use to train the models. Therefore, datasets play a crucial role in developing BioRE models. In this section we present two large-scale datasets containing annotations of GDAs, namely CoMAGC [40] and TBGA [6].

CoMAGC is an hand-labelled corpus proposed by Lee et al. (2013) [40] for the development of automated systems to extract gene-cancer relations. The corpus consists of 821 annotation units from 408 PubMed abstracts limited to prostate cancer, breast cancer and ovarian cancer. In particular, there are 310 annotations related to prostate cancer, 255 related to breast cancer and 256 to ovarian cancer. Each annotation unit comprises four semantically orthogonal concepts, this means that the value of a

concept can be identified not knowing the values of the other concepts. Such values express information about a gene’s expression level change in a cell or tissue which is followed by changes in the cancerous properties of the cell. The first concept is *Change in Gene Expression* (CGE) which express whether the level of a gene is increased or decreased. Then, *Change in Cell State* (CCS) is extracted to capture the way the cell changes together with the gene level change. This concept takes values like *’normal->cancer’*, *’cancer->normal’* and vice versa or *’normal->normal’*, *’cancer->cancer’*. *Proposition Type* (PT) captures whether the causality between gene expression change and cell property change is claimed in the sentence (namely *’causality’*) or not (value *’observation’*). Finally, *Initial Gene Expression Level* (IGE) convey the initial expression level of a gene before the change in its expression level. This concept takes value like *’up-regulated’*, etc., meaning that the considered gene initial expression is up-regulated with respect to its level in a non-cancerous cell, hence we are assuming the expression level of a gene in a normal cell is maintained somewhere. These four concepts allow to understand how a gene changes, how a cancer changes and the causality between the gene and the cancer. In addition, genes are classified into three classes by means of 10 inference rules. These classes are: *oncogene*, when the gene strengthens cancerous properties of cells, *tumor suppressor gene*, when it weakens cancerous properties and *biomarker*, when it shows an altered level in cancerous cells yet it is not identified as either of the above classes. As we said before, CoMAGC is an hand-labelled corpus and it only comprises gene-cancer associations related to prostate, breast and ovarian cancers. Moreover, hand-labelling data is an expensive and time-consuming task hence data sets constructed with this approach are limited in size.

To address these limitations, Marchesin and Silvello (2022) [6] introduced TBGA, a large-scale, semi-automatically annotated data set for GDA extraction based on DisGeNET [2]. The above-mentioned data set has been generated from more than 700K publications and consists of over 200K instances and 100K gene-disease pairs, therefore it is two orders of magnitude larger than current available data sets for GDA extraction such as CoMAGC [40]. Each instance comprises information about the sentence from which the GDA was extracted, the corresponding GDA and information about it. On the other hand, differently from fully distantly-supervised BioRE data

sets, e.g., BioRel [41], TBGA contains expert-curated data hence it represents a more accurate benchmark for BioRE. TBGA was created following four steps, namely data acquisition, data cleaning, distant supervision and data set generation. In the *data acquisition* phase, authors acquired data from DisGeNET [2], that is a database that collects data on genotype-phenotype relationships from several resources. The version of DisGeNET used in [6] comprises 1,134,942 GDAs involving 21,671 genes and 30,170 diseases. Then, a *data cleaning* step is performed where data acquired from DisGeNET are restricted and cleaned to better suite TBGA's objective. In the third step, authors used *Distant Supervision* to create false GDAs that are required to effectively train RE models. Finally, the set of true and false instances obtained in the previous steps are used to generate TBGA. The *creation of the data set* is performed by transforming DisGeNET association types into TBGA relations through a normalization process. Then true instances are divided into training, validation and test sets based on some resource category defined in [6], e.g. validation and test sets only comprise curated data. Then the number of true instances among the data set relations is balanced and a large number of false instances are included into training, validation and test sets to make the data set sparse. Finally, GDAs that occur both in the training and validation or test sets are removed from the training set to avoid to introduce bias. Overall, TBGA has been evaluated on state-of-the-art models for GDA extraction and it proved to be a well-suited data set for GDAs extraction.

In this chapter we presented the task of RE together with some sub-tasks that are required to develop an effective model. We first provided an overview of NER and Entity Linking tasks where we concluded that there is no best model that works for every task since the choice of the model depends on the specific domain. However, DL techniques have shown promising results, especially in NER. About RE in general, we provided an overview of the task in general and discussed the main methods that have been employed to solve it. Overall, we concluded that supervised and distant supervision methods based on DNNs are beneficial to RE yet the main issue lies in the need for annotated data. Finally, we described the task of BioRE, providing some related work based on different techniques. Overall, we discovered unsupervised learning is not well suited for the task and research is now focusing on DL and LMs.

To conclude the chapter, we highlighted the need for challenging datasets for RE tasks and described two datasets that are specifically for GDA extraction, that is the task the system we are reproducing, i.e. DEXTER, solves.

Chapter 3

Original System

In this chapter we introduce Disease-Expression Relation Extraction from Text (DEXTER), a text-mining tool that extract information on gene and miroRNA expression in diseases from biomedical literature [9]. This work has been published in *Database: The Journal of Biological Databases and Curation* and has attracted 11 citations so far. To the best of our knowledge it is the most recent system for RE on Gene-Cancer Associations. This chapter is structured as follows: section 3.1 presents the system architecture and its objectives, section 3.2 provides a detailed description of patterns used for extracting the relations and section 3.3 discusses the system performance reported in the original paper.

3.1 System Architecture

DEXTER is a system that extract GDA starting from biomedical abstracts[9]. This work comes from the need to develop automated tools to identify GDAs since most expression databases include human annotated data, which are expensive and cannot scale to the huge amount of data available in scientific literature. Such resources comprise DisGeNET [2], miR2Disease [3] and BioXpress [4, 5]. DisGeNET is one of the largest platforms of human gene-disease associations currently available, where data comes from different types of source databases containing both expert curated resources and results from text mining tools. On the other hand, miR2Disease is completely human curated and it aims to provide microRNA-disease relationships.

The original paper mainly focuses on BioXpress, in fact one of the objective is to extend the above-mentioned database. BioXpress contains only genes associated with cancer and reports differential expression of genes manually extracted from publications. For these reasons, BioXpress adds some additional constraint on data that must be considered hence the output of the system must be filtered.

In the original paper authors focus only on a particular set of sentences that usually fit into two broad categories called TypeA and TypeB sentences. TypeA sentences are typically comparative, i.e. a gene's expression is typically contrasted between two different samples or conditions. Such sentences are common in biomedical literature because experiments usually compare two different scenarios. In Example 3.1, in sentence #1 *Nucleolin expression* is compared between non-small cell lung carcinoma (NSCLC) tissues and normal lung tissues. This sentence is of interest to BioXpress since the comparison is between tumor and non-tumor-tissues, however such sentences are only a subset of TypeA. In fact, sentence #2 from Example 3.1 is of TypeA but *miR-210 expression* is compared between two stages of cancer.

Example 3.1:

#1 Nucleolin expression was higher in NSCLC tissues than adjacent normal lung tissues. *(TypeA sentence)*

#2 Higher miR-210 expression was found in metastatic tumors compared to primary tumors. *(TypeA sentence)*

On the other hand, in TypeB sentences there is no explicit comparison. As we can see in Example 3.2, TypeB sentences still provide the expression level of a gene (*miR-630*), in a particular sample (*NSCLC tissues and cell lines*), but such expression is not compared to another scenario. We can assess from the expression level, i.e. *down-regulated*, that there is some kind of implicit comparison between samples however we have no information about where it is compared to normal tissues or a different stage of cancer.

Example 3.2: Our results showed that miR-630 was significantly down-regulated in NSCLC tissues and cell lines. *(TypeB sentence)*

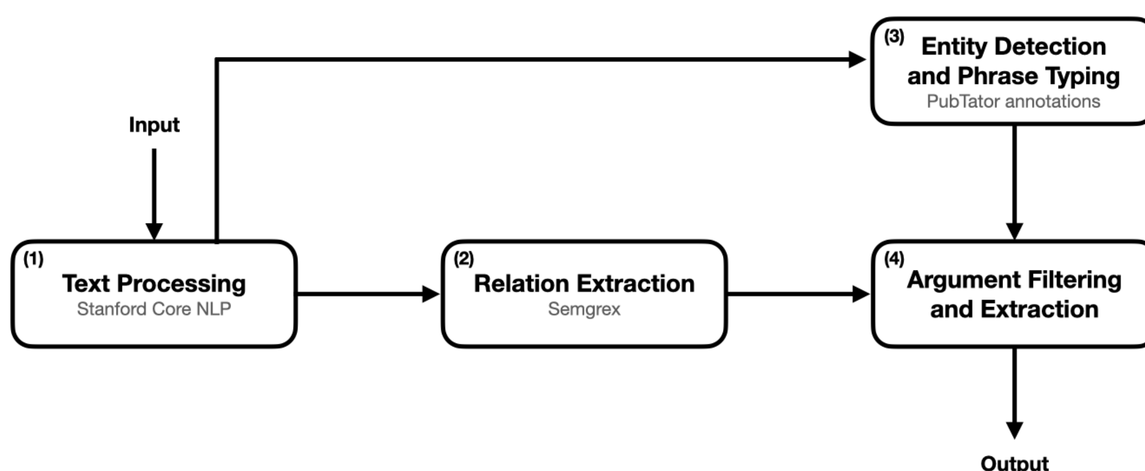


Figure 3.1: DEXTER modules defined in the original paper. Arrows indicates the direction of the computation, providing an overview of the input of each block.

Among biomedical literature regarding gene expression in diseases one can also find a third category called TypeC sentences. In such sentences gene's expression level and related diseases are still reported, however there is no explicit association between the gene and the disease.

> Example 3: Over-expression of C1GALT1 enhanced breast cancer cell growth, migration and invasion *in vitro* as well as tumor growth *in vivo*. *(TypeC sentence)*

In Example 3, we see that *when* C1GALT1 is over-expressed in breast cancer, cell-growth is enhanced. However, there is no explicit association between the gene and breast cancer, meaning there is no information about whether C1GALT1 is typically over-expressed in breast cancer or not. DEXTER does not extract information from such sentences.

DEXTER is an expert system whose rules are based on the syntactic structure of TypeA and TypeB sentences. The system is developed mainly in Python and Java and it is based on several modules as shown in Figure 3.1, each dealing with a different part of the computation.

Text Processing Module

In the first step, the title and text of a medline abstract are tokenized and the abstract is split into sentences. Such operations are performed using the Stanford CoreNLP

toolkit [10]. CoreNLP enables users to derive linguistic annotations for text by running a series of NLP annotators on the text, including token and sentence boundaries, parts of speech, named entities, dependency and constituency parses. DEXTER uses patterns based on lemmas, part-of-speech tags and dependencies among words therefore using this library is essential for achieving the objectives of the original paper. Subsequently, sentences that do not contain predefined trigger words are discarded and not processed any further. Such terms are available in one of the two supplementary files provided as appendixes of the original paper and they are used to detect sentences that might contain expressions and comparative relations. The extraction of relations is based on parsing, therefore authors in [9] use the Charniak-Johnson parser [42, 43] with David McClosky's adaptation to the biomedical domain [44] to obtain syntactic parse trees for each sentence. Charniak-Johnson parser is a statistical parser that returns the best parse tree for each sentence using a regularized MaxEnt reranker on the 50-best parses. The list of best parses is computed using a 50-best parsing algorithm based on a coarse-to-fine parser, that relies on a probabilistic generative model [42]. Parsers are usually trained and tested on a specific domain, hence the performance of these systems degrades on sentences drawn from different domains. David McClosky's adaptation to the biomedical domain has been developed to generalize parsers to a wide variety of domains and increase parsing accuracy [44]. Each tree is further converted into a Standard Dependency Graph (SDG) [11], that provides a representation of grammatical relations between words in a sentence. This operation can be performed using the Stanford conversion tool [10, 11] which convert parse trees into syntactic dependency graphs. Additionally, during the conversion we use the '*CCProcessed*' option which allows to properly treat sentences involving conjunctions and relative clauses by collapsing and propagating dependencies.

Relation Extraction Module

In the RE module, semgrex patterns are used to extract relevant components containing the gene expressed, its level and the associated disease. In typeA sentences, the RE module extracts the aspect being expressed, the level of expression of such aspect and the entities being compared, that can be two samples of tissue or two disease

states. In TypeB sentences the components extracted are basically the same, the only difference is that there is only one entity where the gene/miRNA is expressed. A different terminology is used in the original paper to identify TypeA and TypeB components. In particular, in TypeA sentences the aspect being compared is reffered to as *Compared Aspect (CA)*, the scale of the comparison is called *Scale Indicator (SI)* and the entities being compared are defined as *Compared Entity (CE)*. On the other hand, in TypeB sentences the expression level of an entity is called *Level Indicator (LI)*, the entity being expressed is defined as *Expressed Aspect (EA)* and the disease state where the entity is found is reffered to as *Expressed location (EL)*.

Patterns are written in Semgrex, which is also part of the Stanford NLP toolkit and it allows the matching of nodes and edges in a dependency graph. In particular, nodes in a SDG are the tokens corresponding to sentence words and patterns are written to check a set of attributes for each token. Attributes comprise lemmas, part-of-speech tags and dependency labels between the considered node and already-matched nodes. A node is represented by a set of attributes contained by curly braces. Attributes are either plain strings or regular expressions blocked off by "/". For example, *'{word:/(higher|lower|high|low)/}'* match any node representing one of the words between round brackets. On the other hand, relations are defined by a symbol representing the type of relationship and a string or regular expression representing the value of the relationship. Semgrex documentation[1] provides a full list of node relations and their symbols. The relation we will use the most is '>', for example *'A >subj B'* indicates that node A is the head of a dependency labelled *nsubj* (i.e., nominal subject), whose dependent is node B. We see an example of such patterns in Figure 3.2. To better understand how Semgrex patterns work, we apply the pattern in Figure 3.2 to the sentence in Example 3.3, whose dependencies are shown in Figure 3.3.

> Example 3.3: TMEM48 expression level was higher in NSCLC tissues than that in the adjacent normal tissues.

In this case we can see that node N0 is identified in token *higher*, node N1 is token *level*, node N2 is token *tissues* and node N3 is token *tissues*. Semgrex patterns only return

[1]https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/semgraph/semgrex/SemgrexPattern.html

```
Cond_1 : {word:/(higher|lower|high|low)/}=N0
Cond_2 : {}=N0 >nsubj {}=N1
Cond_3 : [{}=N0 | {}=N1] >/nmod:in/ {}=N2
Cond_4 : {}=N2 !> /case/ {word:than}
Cond_5 : {}=N0 >/nmod:in/ ({}=N3 > /case/ {word:than})
```

Figure 3.2: Example of a Semgrex pattern provided in a supplementary file of the original paper.

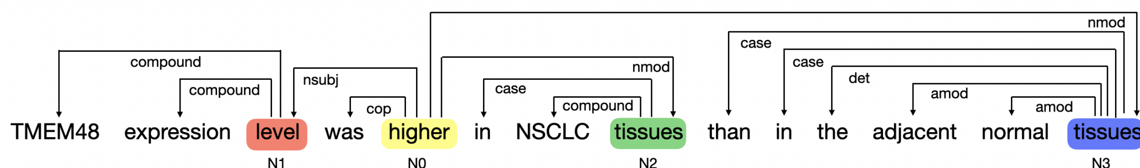


Figure 3.3: Example sentence dependencies obtained using https://corenlp.run/. We highlighted the matched nodes with different colors.

head words hence we extract the spans corresponding to each component by following all outgoing edges from the matched tokens. Consider our previous example, we extract *higher* as Scale Indicator (SI), *TMEM48 expression level* as Compared Aspect (CA), *in NSCLC tissues* as Compared Entity 1 (CE1) and *than in the adjacent normal* as Compared Entity 2 (CE2). All comparison patterns used to extract the components in the RE module are available in a supplementary file of the original paper. A full description of the development of comparison patterns is available in section 3.2.

Entity Detection and Phrase Typing Module

The entity detection and phrase typing module takes sentences parsed by the text processing module and determine if there are terms referring to entities of type gene/miRNA, expression or disease/disease-sample. Pre-computed annotations from PubTator [12], an automatic system providing annotations of biomedical concepts, are employed to detect genes and diseases mentions in the abstracts. On the other hand, microRNA mentions are detected using regular expressions based on the naming convention described in miRBase [45]. Finally, to determine whether a phrase is of type *"Expression"* its terms are checked against a list of expression triggers such as "expression", "level", "over-expression", etc. The same strategy has been adopted to detect disease-sample phrase type. In this case triggers can be "tissues", "cells",

"patients", "samples", etc. A full list of such triggers is provided in the supplementary files of the original paper.

Argument Filtering and Extraction Module

The argument filtering and extraction module consists of two main steps: verify if the tokens matched by the RE module are of the right type, i.e. contains the required mentions, and extract the relevant information such as the gene/miRNA expressed and the associated disease. To perform both tasks this module takes the results of the RE module and the mentions identified in the previous block as input. For TypeA sentences, the compared aspect must be of type expression, while the two compared entities need to be of type disease/disease-sample. Similarly, for typeB sentences the expressed aspect and the expression level must be respectively of type expression and disease/disease-sample. If the checks succeed, relevant information are extracted from the results computed by the RE module. This includes the expressed gene, its level and the associated disease. The gene/miRNA expressed is extracted from the compared aspect/expressed aspect argument using the mentions extracted by the entity detection and phrase typing module. If the system detects a gene expression it also returns the corresponding NCBI Gene ID downloaded from PubTator together with the gene mentions. Expression level information is captured by the RE module in the scale indicator/level indicator argument, therefore the level is normalized to high or low by matching the argument against a list of triggers available in the supplementary files. Some examples of triggers for *"high"* expression are "over-expressed", "increased", etc, while terms like "low", "decreased", "down-regulated" refers to *"low"* expression level. Extracting the associated disease requires some extra care. Disease mentions downloaded from PubTator are searched in the compared entities/expression location components, however in some cases the entities contain only generic diseases such as "tumor", "cancer" or "disease". In these cases, the associated disease is extracted from the abstract by looking for disease mentions in the title or in the first sentence. Alternatively the associated disease can be inferred from sentences in the 'methods' part of the abstract or from sentences describing the experimental set-up. Such sentences are detected by checking if they

contain certain *"investigation triggers"* such as "investigated", "examined", "analyzed", etc or *"analyzed triggers"* such as "tested", "collected", "explored", etc. PubTator returns disease mentions normalized to MEDIC IDs, however such IDs are mapped to DOIDs to allow an easy integration in BioXpress.

Finally, we set a frame-to-reference flag to distinguish sentences containing comparisons. This is needed to extend BioXpress database since it requires direct comparison between a tumor tissue and a control sample. For TypeA sentences, we set the flag to *Control* if the CEs contain words such as "control", "normal", "healthy", "adjacent", etc. Otherwise we set it to *Not-Control*. For TypeB sentences instead, we can still convey implicit comparison to control. In this case we check if the LI is "over/under-expressed", "increased", "decreased", "elevated", "reduced", etc. In such cases, we assume the comparison reference is to normal samples and we set the flag to *Control_Implicit*, otherwise we set it to *none*.

3.2 Comparison Patterns

In order to extract GDAs, DEXTER uses Semgrex patterns based on syntactic dependencies and part-of-speech tags. These patterns convey TypeA, i.e. Comparative, and TypeB sentence structures.

TypeA Patterns

To develop patterns for TypeA sentences authors rely on a previous work [46] where they identify structures in biomedical text. As we said before, comparison sentences contains two *Compared Entities (CE)* and an aspect on which the entities are being compared called *Compared Aspect (CA)*. There are two more parts indicating the comparison. One is a word that indicates the scale of the comparison, i.e. *Scale Indicator (SI)* and the other separates the two compared entities and it is referred to as *Entity Separator (ES)*. Such word is usually a comparison word and can be *than*, *versus*, *vs.* or *compared*. In Gupta et al. [46] authors categorized comparative structures into four classes following Jindal and Liu [47]. They consider structures belonging to two of these classes: *Non-Equal Gradable* and *Equative* structures. The

first provides differential relations between two entities (i.e. higher, lower) while the other indicates equal relation between the entities (i.e. X as high/low as Y). The other two classes that are not considered are *Superlative* sentences, where one entity is ranked over all the others (i.e. X is greater than all the others) and *Non-Gradable* structures, where there is no explicit comparison between two entities. The main idea behind the pattern construction is to identify SI and ES words and then extract the other components using dependencies from them.

As we said above, in [46] authors build patterns for Non-Equal Gradable and Equative sentences however, since DEXTER extracts differential relations, we only focus on Non-Equal Gradable structures. In this class we find different syntactic structure based on three part-of-speech (POS) tags of the SI: Comparative Adjective (JJR), Comparative Adverb (RBR) and Verbs. Patterns provided as appendix of the original paper are based on structures where the SI is either an adjective or a verb hence we do not describe the comparative adverb case. The most frequent case for SI is a *comparative adjective* (JJR) such as higher, lower etc. We can distinguish two categories where the SI plays two different roles. In the first case the adjective serves as the predicate of the comparison, therefore it involves copular structures as shown in Figure 3.4a. Here the CA is typically the subject of the adjective hence we follow the *nsubj* edge to return the head of the component. Then we use the *nmod* edges to retrieve the two CEs. In particular, in Figure 3.4a the two edges are *nmod:in* and *nmod:than*. We can also verify that the two extracted CEs are separated by an ES, in Figure 3.4a such word is *'than'*. In the second category the comparative adjective modifies the CA, as shown in Figure 3.4b. In this case the Noun Phrase (NP) containing the SI is connected to a verb that serves as predicate of the comparison. In order to extract the CA we can use the *amod* edge from the SI or the *nsubj* or *nsubjpass* edge from the verb. The CEs are extracted as in the previous category but here they depend on the verb instead of the SI. In particular, in Figure 3.4b the CEs are extracted following two edges *nmod:in*. Note that the same patterns can be apply even if the SI is an adjective (JJ) instead of a comparative adjective.

On the other hand, the SI could be a verb such as "increased" or "decreased" hence in this case the verb serves as the predicate of the comparison. We can distinguish

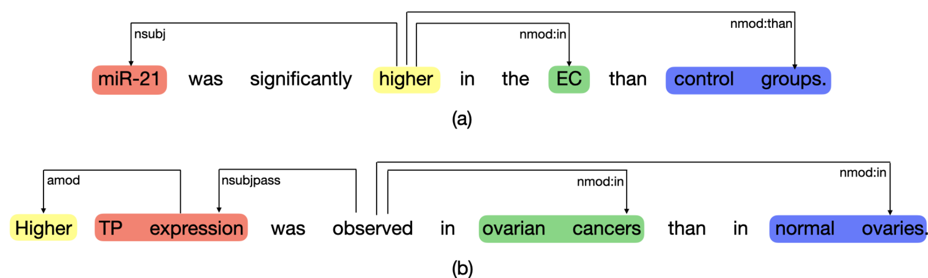


Figure 3.4: Comparative Adjective example. For the sake of simplicity, we only display the mentioned dependencies. We highlighted the components with different colors, in particular the SI is highlighted in yellow, the CA in red and the two compared entities in green and blue.

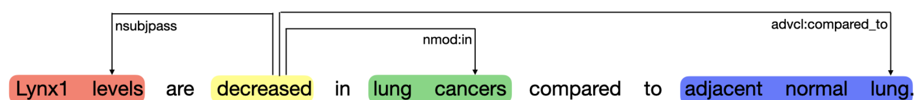


Figure 3.5: Verb example (passive form). Dependencies and components are shown as in Figure 3.4.

two categories based on the voice of the verb (i.e. passive or active) where the CA is extracted in two different ways. In fact, in the passive form we follow the *nsubjpass* edge to the determine the CA while in the active form the CA is the direct object of the verb. As far as CEs are concerned we extract them as we did before. We provide an example of this structure in Figure 3.5. In this case, the SI is *decreased* and we extract CA following the edge *nsubjpass* while the two entities are identified by the edges *nmod:in* and *advcl:compared_to*. There are some cases in which a verb in the past participle tense (VBN) can be used as an adjective that modifies a noun, as shown in Example 3.4. We treat such cases as the second category of the Comparative Adjectives case.

> Example 3.4: Decreased miR-149 expression was observed in HCC tissues compared to peritumoral tissues.

TypeB Patterns

In TypeB sentences the compared aspect is expressed in only one entity, hence there is no comparison between two samples. As explained before, in this case we are intrested in extracting the *Expressed Aspect (EA)* that is the entity being expressed, the *Expressed Location (EL)* that is the sample were we found such expression and the *Level Indicator (LI)* that is the level of expression. In this type of sentences the LI can

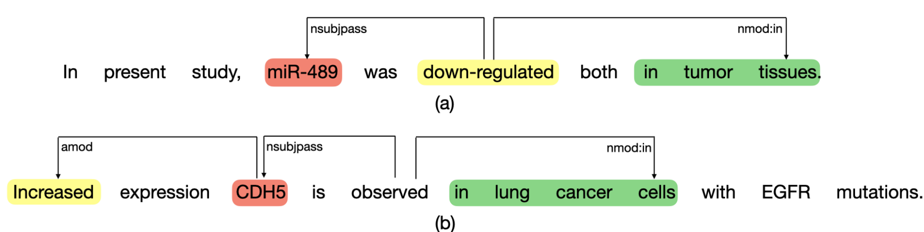


Figure 3.6: TypeB example sentences. For the sake of simplicity, we only display the mentioned dependencies. We highlighted the components with different colors, in particular LI is highlighted in yellow, EA in red and the EL in green.

be either the main predicate of the sentence or it can be attached as a noun modifier. We show an example of the first category in Figure 3.6a, where the main predicate of the sentence is the verb *down-regulated*, which is the LI. In this case we extract the EA, i.e. *miR-489*, by following the edge *nsubjpass* while the EL, which is *in tumor tissues*, is retrieved using the edge *nmod:in*. On the other hand, Figure 3.6b illustrates an example of the second category, where the LI *increased* is attached as a modifier to the EA. In particular the LI depends on the EA and it can be extracted following the edge *amod*. The main predicate is the verb *observed* therefore we extract the EA, that is *CDH5*, by following the edge *nsubjpass* while the EL (i.e., *in lung cancer cells*) is returned using the edge *nmod:in* that depends on the verb.

3.3 Results

The main objective of the development of DEXTER was to extend the BioXpress Database. For this reason, the system was originally tested on results relevant to BioXpress and on its ability to extract GDAs without any limitation. Both evaluations compare DEXTER's output with manually annotated data.

We recall that DEXTER output can extend BioXpress if the disease is cancer and there is a comparison between cancer and normal samples. To evaluate the ability of the system to extend the literature-based portion on BioXpress authors in [9] consider three use cases and a large set of PubMed abstracts related to cancer.

All the results for the three use cases are reported in Table 3.1. In the first set of abstract, authors focused on lung cancer in order to address the case in which a

Table 3.1
Results reported in the original paper for the three use cases. Note that we are filtering out results that are not appropriate for BioXpress, hence we are only keeping entries where the disease is cancer and there is a comparison with a control sample.

	# of input abstracts	# of abstracts extracted		# of entries		# of expressed genes	
		TypeA	TypeB	TypeA	TypeB	TypeA	TypeB
Lung Cancer	88 431	742	1 448	985	2019	642	1383
GTs	27 516	90	180	106	236	42	73
microRNAs	28 067	1 650	3575	2 522	6 437	477	721

researcher wants to study a particular cancer. As a consequence, DEXTER was run on 88 431 abstracts retrieved from PubMed and only results where the extracted cancer was one of the lung cancer DOIDs were kept. In total DEXTER extracted TypeA information from 742 abstracts and TypeB information from 1 448 abstracts for a total of 642 and 1 383 gene extracted respectively from TypeA and TypeB expressions. The second use case focuses on a group of genes called *glycosyltransferases (GT)*, whose disfunction or deregulation is responsible for several diseases, including different types of cancer. In this case DEXTER computed results for 27 516 abstracts. After filtering out relations that are not relevant for BioXpress, the system returned TypeA and TypeB information from 90 and 180 abstracts for a total of 45 genes in 34 cancers and 73 genes in 52 cancers respectively. Finally, the last use case regards microRNA-related abstracts. In this case the number of total abstract was 28 067 and DEXTER returned BioXpress-appropriate TypeA information from 1 650 abstract for a total of 477 microRNAs in 114 cancers and TypeB information from 3 575 abstracts comprising 721 microRNAs in 157 cancers.

Finally, in order to test the robustness and scalability of DEXTER, authors ran the system on 1 750 928 abstracts extracted 24 416 unique gene-cancer types.

The second evaluation uses standard measures of precision and recall both on BioXpress-appropriate data and on results obtained without the limitations imposed by the database guidelines. For the first set of abstracts authors selected 100 abstracts related to GTs and 100 abstracts related to microRNAs while for the second group they randomly selecte 100 abstracts both among GTs and microRNA related abstracts. Evaluation results are shown in Table 3.2. An entry is a true positive (TP) only when every component of the output matched human annotations, hence when even if just

Table 3.2
Results reported in the original paper for the second evaluation. Row *BioXpress* indicates results for the output filtered according to BioXpress guidelines while row *Unfiltered* corresponds to DEXTER's output without any limitations.

	True Positive	False Positive	False Negative	Precision	Recall	F-score
BioXpress	77	5	15	93.90	83.69	88.51
Unfiltered	126	13	41	90.06	74.56	81.81

one component between expressed gene/microRNA, expression level and associated disease is different from the ground-truth the entry is marked as False Negative (FN) or False Positive (FP).

Overall the system achieved pretty good results with a precision of 94% in the BioXpress-based data and 90% in the output without limitations. The other measures are all above 80% except for the recall of the second set of abstract that is just 74.5%, which is common in rule-based methods due to the variety of sentence structures. Most errors are due to mis-parsing, errors in detecting the associated disease and lack of patterns.

Chapter 4

SpaCy and Text Processing

In this chapter we describe the implementation of DEXTER, providing a description of how we implemented the modules described in the previous chapter. We also highlight the differences between our version and the original system and explain the main differences. In particular, we focus on the input handling step and the *Text Processing* module.

We recall that the original version of DEXTER is developed both in Java and Python thus each block runs separately from the others. Our goal is to implement an end-to-end system publicly available and easy to reproduce. For this reason, we decided to develop DEXTER as an end-to-end application that takes as input biomedical abstracts and returns relevant information on the expressed gene/microRNA and the associated disease. We preserved the overall block architecture and we made some changes in each block to enable a seamless integration of the different modules. Our system is entirely developed in Python therefore it was not possible to use Stanford CoreNLP toolkit [10] since it is written in Java. CoreNLP developers published Stanza [48], that is a Python NLP package that includes an interface to the CoreNLP Java package. However, we decided not to use it since, at the time of writing, it does not offer support for dependency parsing. Instead, we use the SpaCy library [13], which is an open-source library written in Python that provides some utility functions to process text efficiently and effectively. Spacy was released in 2016 by Matthew Honnibal, who wanted to provide small companies an accurate and fast pipeline for NLP tasks that is suitable for production use [14]. SpaCy comes

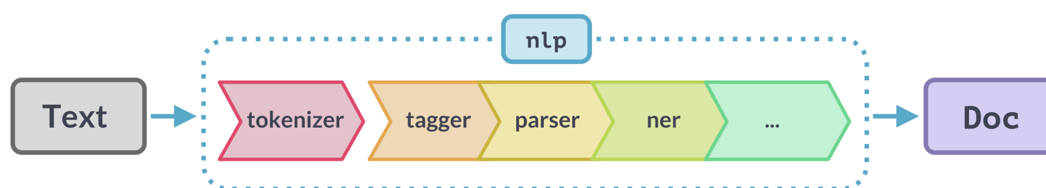


Figure 4.1: SpaCy processing pipeline. SpaCy first tokenizes the text and then processes it in several steps. Here we report the standard components included in the trained pipelines provided by the library itself. The last block, identified with three dots, indicates custom components that can be added by the users to the standard pipeline. This image is taken from SpaCy official website.

with some trained pipelines that provide support for several languages including English, Italian, Chinese and many others. Since version 3.0, SpaCy pipelines are transformer-based hence their accuracy has reached the current state-of-the-art in parsing, tagging and NER.

We provide an overview of SpaCy processing pipeline in Figure 4.1. We used the `en_sci_sm` trained pipeline provided by ScispaCy [1], that is a Python package containing SpaCy models for processing biomedical data. We can load the trained pipeline after downloading it using the function `nlp_biore = spacy.load(en_sci_sm)`. Now, to process some text and create the `Doc` object that we will later use to access all text annotation we simply write `doc = nlp_biore("Some random text!")`. By simply loading a trained pipeline we will only use the standard components provided by Spacy, i.e. SpaCy tokenizer, parser, NER and tagger, but we can also add some custom components to perform some tailored functions. We added a custom component to the SpaCy pipeline to expand entity mentions using hand-craft rules which will be useful when extracting information from the matched components. The details of such component will be described in Section 4.2. Once we parsed the input text with the SpaCy model, a `Doc` object is returned, which contains all annotated data we computed by the SpaCy pipeline, such as dependencies, Part-Of-Speech tags (POS) and all the tokens detected in each sentence. We will explain the `Doc` object in more detail in Section 4.2, where we describe the *Text Processing Module*.

[1]https://allenai.github.io/scispacy/

4.1 Input Handling

The original work does not share the source code, therefore to reproduce it we used the description of the paper and the supplementary files containing comparison patterns and trigger lists. Authors provide some data in BioXpress[2] which corresponds to the output of the original model. We downloaded from BioXpress website the three csv files corresponding to the three use-cases the authors of the original paper used to evaluate their system: differential expression glycosyltransferases in cancer, genes in lung cancer and all microRNAs in cancer. We used such files as input to our system therefore the implemented version of DEXTER takes as input a csv file whose rows must contain the sentence text and the PMID of the paper where the sentence was extracted. Our system processes each sentence separately and we do not split abstracts into sentences as they did in the original work.

As we already mentioned, our implementation is based on the data available in BioXpress. In total, we have 9636 entries and 14 columns. We use the *'PMID'* and *'Sentence'* columns as inputs to our system and we use columns *'geneMen'*, *'ExpressionLevel'* and *'SentenceType'* to check the accuracy of our implementation. After checking the input data we decided to add a pre-processing step where we clean each sentence and remove some noise that may affect the dependency parsing performance. In particular, the sentences may start with a number and a dash, which probably indicates the number of the sentence in the abstract. Moreover, some sentences are between quote marks and they may be in upper-case like *'RESULTS'*, *'CONCLUSION'*, etc. If we do not clean sentences before parsing them with SpaCy we may introduce noise resulting in poor parsing performance. For this reason, we first check if the sentence is enclosed in quote marks and in the case we remove them. Then we use some regular expressions (regex) to remove number and dash at the beginning of the sentence and the words in upper-cases. We design two regexs to remove numbers followed by a dash from the beginning of the sentence which we joined together with the 'OR' operator. In this way, we consider both sentences that begin with a number and sentences where the number is followed by a dash. The former case matches

[2] https://hive.biochemistry.gwu.edu/bioxpress

the regex `r'^\s*[0-9]+'`, where the symbol `'^'` indicates that we are matching the beginning of the string and we check for one or more occurrences (i.e, symbol `'+'`) of any digit (i.e., `[0-9]`). Similarly, to check if there is also a dash we simply add the symbol `'-'` at the end of the previous regex. In Example 4.2.(a) we highlighted the substring that matched the above regex and we show the result of the `sub` operation in Example 4.2.(b).

> Example 4.2:
> (a). 5-PAX5 expression level was significantly lower in NSCLC than that in adjacent non-cancerous tissues (P = 0.0201).
> (b). PAX5 expression level was significantly lower in NSCLC than that in adjacent non-cancerous tissues (P = 0.0201).

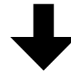
After removing the number and the dash, if present, we check if the sentence starts with some trigger words in upper-case that can be followed by a colon, as highlighted in Example 4.3.

> Example 4.3: CONCLUSIONS: We demonstrated that miR-30b/c was down-regulated in NSCLC specimens compared with adjacent non-tumor tissues.

Such words can be RESULTS, CONCLUSION, PURPOSE, etc. These words can also be combined together, e.g. *'RESULTS AND CONCLUSION:'* or *'RESULTS/CONCLUSION'*. We wrote a regex to match all cases and used the `sub` function as we did for the numbers before. Finally, we discovered that some sentences contain p-values or other statistical measures enclosed in square or round brackets. Such components worsen the performance of our system since the dependency parser wrongly attach dependencies to them. For this reason, we decided to remove also all substrings enclosed in square brackets. To remove all substrings enclosed in round brackets worsen the performance, because in this case we may lose relevant information. Therefore, we just removed numbers in between round brackets if they are at the beginning of the sentence, as shown in Example 4.4.

> Example 4.4: (2) Plasma concentrations of miR-18a were significantly higher in pancreatic cancer patients than in controls (P<0.0001).

Increased expression CDH5 is observed in lung cancer cells with EGFR mutations.



Text (t.text)	Lemma (t.lemma_)	Tag (t.tag_)	POS (t.pos_)	Head (t.head)	Dependency (t.dep_)
Increased	Increase	VBN	VERB	CDH5	amod
expression	expression	NN	NOUN	CDH5	compound
CDH5	cdh5	NN	NOUN	observed	nsubjpass
is	be	VBZ	AUX	observed	auxpass
observed	observe	VBN	VERB	observed	ROOT

Figure 4.2: Linguistic annotations returned by the SpaCy model for sentence *'Increased expression CDH5 is observed in lung cancer cells with EGFR mutations'*. We provide the sentence and the output of the first five tokens parsed by the system.

4.2 Text Processing Module

After the sentence preprocess, we parse the text using the SpaCy trained pipeline. The sentence text is transformed into a `Doc` object which contains all annotations for the sentence. More specifically, we can return a list of all tokens identified in the sentence through the instruction `toks = [t for t in doc]`. Then, for each token `t`, we can access information about its text (`t.text`), POS tags (`t.tag_`), its head (`t.head`) or its dependency label (`t.dep_`). We provide an example of computation in Figure 4.2. Here we can see a sentence and below the table with the output of the first five tokens parsed by the system. Both token attributes `t.tag_` and `t.pos_` return POS tags but the former returns fine-grained part-of-speech tags while the latter contains coarse-grained part-of-speech from the Universal POS tag set[3]. We can also notice that the ROOT node of the sentence, i.e. *observed*, does not have an artificial node as head but in SpaCy the root node's head is the root itself.

We also noticed that in some cases miRNA mentions were split into two different tokens. This may be due to the fact that some miRNA mentions contain dashes, e.g. *miR-150* or *miR-3940-5p*, hence SpaCy tokenizer could split the word into two tokens like *miR-* and *150*. This is problematic for the next modules, for example it becomes challenging to detect miRNA mentions because we cannot match the miRNA regex over each token but we need to check the whole sentence at once. In

[3]https://universaldependencies.org/u/pos/

order to mitigate this issue we retokenize miRNA mentions. In fact, SpaCy allows for the retokenization of the `Doc` object by means of the following code:

```
with doc.retokenize() as retokenizer:
retokenizer.merge(doc[start:end])
```

This instruction merges the document's tokens from id `start` to id `end-1` into a single token. Therefore, we check for miRNA mentions in the sentence text and if there is at least one we ensure mentions are stored in a single token. If not, we merge the tokens containing the miRNA mention and modify the sentence's tokens. To detect miRNA mentions we use the regex we developed based on miRBase convention [45], which we will describe in more detail when talking about the *Entity Detection and Phrase Typing module* in Section 6.1. After parsing, we check if the sentence contains certain words, that are called *trigger words* in the original paper, which convey the presence of TypeA or TypeB relations. Sentences that do not contain any trigger words are discarded and not processed any further. We developed a list of trigger words based on the two supplementary files provided as appendixes of the original paper. Instead of matching the tokens text we match lemmas so that we consider both singular and plural forms. Some example triggers we consider are: "high", "low", "increase", "decrease", "reduce", "elevate", "occur", "appear", "identify", "show", "prove".

An important difference between the original syntactic processing and the one we implemented is that we directly compute the dependency graph corresponding to each sentence by solving a dependency parsing task through SpaCy. Instead, in the original system sentences were parsed using the Charniak-Johnson parser [42, 43] with David McClosky's adaptation to the biomedical domain [44] which produces a constituency parse tree for each sentence. A constituency parse tree of a sentence is built by dividing the sentence into sub-phrases, e.g. NP, Verb Phrase (VP), etc, that have a specific syntactic meaning. Therefore, the syntax of the sentence is expressed according to a Context-Free Grammars (CFG) [7]. Each parse tree is then converted into a syntactic dependency graph by means of the Stanford conversion tool [10, 11] with the '*CCProcessed*' option which collapses and propagates dependencies to properly treat sentences involving conjunctions. Considering Example 4.5, token

patients has dependency label *'nmod'* and it is head of the dependency *'case'* of token *in*. This option propagates the case dependency to token patients, whose dependency becomes *'nmod:in'*.

> Example 4.5: Plasma miR-187 was significantly higher in OSCC patients than in normal individuals.

SpaCy does not provide such tool hence we modify the Semgrex pattern provided as supplementary files of the original paper accordingly. In addition, computing the dependency tree directly instead of inferring it from a constituency parse tree could affect the performance of the overall system. In fact, dependency trees do not make use of sub-phrases and the syntax of the sentence is expressed in terms of dependencies between words, not by means of a CFG. We decide to proceed in this way in order to provide an end-to-end application with seamless integration among modules.

Entity Expansion

We now describe in more detail the custom component we developed for entity expansion. We recall we placed such component in the SpaCy pipeline after all standard components, therefore we can make use of all annotations computed by the pipeline. Expanding entities in fact can be useful in the following modules, for example for extracting the matched components in the RE module. We provide an example of the results of our custom components in Figure 4.3. Here we can see that the standard components retrieved 6 entities while we collapsed them into 3 entities which corresponds to the Compared Aspect and the two Compared Entities. Entities are detected in SpaCy by means of a statistical NER system[4], which detects the named entities and assign a label to them. For the model we are considering, the only label available is *ENTITY*. Detected entities can be accessed by means of the `Doc` object's attribute `ents` which provides an iterator over the entities. We can access the entity's boundaries and labels by means of the attributes `start`, `end` and `label`. We expand entity mentions using hand-craft rules based on token dependencies and

[4]https://spacy.io/usage/linguistic-features#named-entities

Plasma miR-187 was significantly higher in OSCC patients than in normal individuals.

Plasma miR-187 was significantly higher in OSCC patients than in normal individuals.

Figure 4.3: Example of entity expansion, where entities detected are highlighted in grey. Standard SpaCy pipeline retrieved 6 entities for the considered sentence, while the custom component collapsed the 6 entities detected by the standard pipeline into 3 entities, which corresponds to the Compared Aspect and the two Compared Entities.

some trigger words. We consider four cases: first we check for compound names and adjectives between single tokens, then we apply the same checks to the entities we have already detected and finally we expand sentences like *expression level of..* or *patients with..* .

In the first case, we check if the sentence contains some words that can be expanded as entities. Such words can be: "patient", "sample", "tumor", "cancer", etc. Then, for each of these tokens we check if it is already within an entity or not in order to identify the starting entity boundaries. If the token is already within the entity the boundaries will correspond to the entity boundaries otherwise we set the token id as start of the entity and the subsequent id will be the end of the entity. Afterwards, we check among the children tokens (which can be returned by the attribute `token.children`) if any of them depends on the considered token with a *compound* or *amod* dependency and it is outside of the entity. If so, we expand the entity to contain the children token. If we do not match any children token with such dependencies, we perform the same checks among all sentence tokens. Once we expand single tokens, we check if we can expand any other entity. In particular, for each entity we check if there is any token outside of the span, whose head is inside the entity, that has a dependency of type *'compound'* or *'amod'* and if so we expand such entity to contain the matched token. We show an example of this second case in Figure 4.4. Here we can notice that the standard components already detected entities like *SFRP3*, *mRNA expression*, *tissues*, *lung samples*, etc. By adding our custom component all compound names and adjectives referring to the entity are collapsed together in the same span.

We also expand the expression level phrases and phrases indicating patients

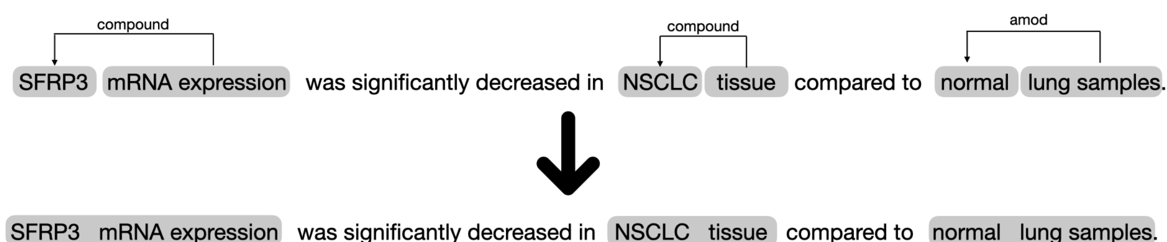


Figure 4.4: Example of entity expansion for the second case. Entities detected are highlighted in grey and we provide the dependencies of interest. The custom component collapsed all compound names and adjective into a single entities, that actually corresponds to the components that will be extracted by the RE module.

diagnosed with some conditions. In the first case, we check if the sentence contains words like *expression* or *level*. If so, we check if such words are an entity or not to understand the entity boundaries as we did for the previous cases. Then, we check if the subsequent word is *of* and it is followed by an entity. If so, we expand the entity to create one unique entity as shown in Example 4.6 (a). If not, we check if previous token is an entity and if so we expand it to contain also the detected token, as shown in Example 4.6 (b), where we only highlight the entity we are considering.

Example 4.6:
(a). In this paper, it was revealed that the expression of FOXJ2 was lower in NSCLC samples compared with matched peritumoral lung tissue.
(b). ELMO3 expression level was significantly up-regulated in NSCLC patients' tissues and serum compared with controls (P<0.001).

Finally, we expand entities of the type *patients with..*, which will be useful especially for the disease extraction task. In fact, in sentences like the one in Example 4.7., the RE module will extract token *patients* as compared entity hence we may lose information about the associated disease. In this case we search for the lemma *'patient'* in the sentence and if found, we check if the subsequent token is the word *'with'* followed by an entity. If so, we expand the entity as shown in Example 4.7, where, for the sake of simplicity, we only highlighted the entity we are considering.

Example 4.7: We found that serum DKK-1 level was significantly higher in patients with NSCLC than healthy controls.

Chapter 5

RE Module

In this chapter, we describe the implementation of the RE module, which is the core of the system. In fact, in this step we extract relevant components from each sentence that contain information about the expressed gene or miRNA, its level of expression and the sample where we detected such information. The RE module is based on rules defined as Semgrex patterns over the dependency tree representation of each sentence. For this reason, dependency parsing is crucial for the outcome of this module thus of the overall system. The rest of the chapter is structured as follows: firstly we briefly describe the dependency parsing task in general, providing some formal definitions. Then we describe the `DependencyMatcher`, which is a matcher provided by SpaCy library to match Semgrex patterns within dependency trees[1]. We will show how patterns are written and provide an example of how to apply them to sentences. In Section 5.1 we focus on the rules we develop to extract relevant components and in Section 5.2 we describe the module implementation. We use all rules described in the supplementary files provided as appendixes of the original paper and we also developed some rules ourselves to match more sentences.

The whole RE module is based on *Dependency Parsing*, which is an NLP task producing a dependency tree for a given sentence. A dependency tree is a directed graph $G = (V, E)$ where the set of vertices comprises the tokens of the sentence and edges are grammatical relationships among them. Dependency trees describe syntactic structures based on the words in the sentence and the grammatical relations

[1]https://spacy.io/usage/rule-based-matching#dependencymatcher

among them. This allows us to abstract away from word ordering information, thus allowing for a smaller number of rules for dependency grammars than phrase structure grammars in free word order languages (i.e, languages with no strict requirements on phrase order). Formally, a dependency tree satisfies three constraints:

1. There exists a single node, the root, with no incoming arcs.
2. Each vertex has one incoming arc, with the exception of the root node.
3. There exists a unique path from the root node to each vertex of the graph.

We can translate constituent-based trees into dependency trees through some rules and trees produced in this way are said to be *projective*. An arc is said to be projective if there is a path from the head of the arc to every word that lies between the head and the dependent in the sentence. Therefore, a dependency tree is said to be projective if all its arcs are projective [7]. We will describe the main approaches used for Dependency Parsing in Section 7.2, when we present the Spacy Dependency Parser in more detail.

Once we computed the dependency tree of a sentence, we can apply Semgrex patterns to match relevant tokens. In SpaCy, this operation is carried out by the *DependencyMatcher*[2]. This tool allows to match tokens in the dependency trees by specifying some attributes, e.g. dependencies among words, by means of Semgrex patterns. A pattern is basically a list of dictionaries, each containing rules to match a token based on its relation to an existing token in the pattern and some token attributes like lemmas for example. We recall that in Semgrex patterns matched nodes correspond to some tokens inside the dependency tree hence we will use the term node and token interchangeably. We provide a simple example of a pattern in Figure 5.1. Besides the first dictionary, which defines an anchor token using only `RIGHT_ID` and `RIGHT_ATTRS` , each pattern has the following keys:

- `LEFT_ID`: name of the left-hand node in the relation we are matching. This token must be already matched inside the pattern.
- `REL_OP`: operator that describes the relations among nodes in `LEFT_ID` and `RIGHT_ID`.

[2]https://spacy.io/api/dependencymatcher

```
pattern = [
    {
        # anchor token: downregulated
        "RIGHT_ID": "downregulated",
        "RIGHT_ATTRS": {"ORTH": "downregulated"}
    },
    {
        # downregulated -> subject
        "LEFT_ID": "downregulated",
        "REL_OP": ">",
        "RIGHT _ID": "subject",
        "RIGHT_ATTRS": {"DEP": "nsubjpass"},
    }]
```

Figure 5.1: Example of a pattern that can be used to match dependency trees using the `DependencyMatcher`. Each pattern has an anchor token and all other tokens are matched based on their attributes and relations with already-matched tokens.

- `RIGHT_ID`: unique name for the right-hand node in the relation we are matching. This token must be a new node that we have not matched before in the pattern.

- `RIGHT_ATTRS`: token attributes that the right-hand node must match. This can include dependency labels or other token attributes like lemmas (`LEMMA`), POS tags (`POS, TAG`) or the exact text of the token (`ORTH`).

Most of the operators supported by the `DependencyMatcher` come directly from Semgrex. The operators we will use the most are : *A > B*, that means A is the immediate head of B, *A >> B*, which imply that A is the head of B following a path from A to B, i.e. A is an ancestor of B, and finally *A ; B* that detect when A immediately follows B and the two nodes are within the same dependency tree. Considering the example in Figure 5.1, the anchor node corresponds to a token whose text is the word *"downregulated"* and the second node we match is a token that have relation *downregulated > subject* with the anchor token. This means the node identified with name *"subject"* depends on the anchor node with label *"nsubjpass"*, i.e. it is the passive subject of the verb we matched. If we apply the defined pattern to the sentence *"MiR-203 is downregulated in lung cancer cells."*, whose dependencies are displayed in Figure 5.2, we see that the anchor token, i.e. node named *downregulated*, is matched by the token *"downregulated"* while the node named *subject* is matched by token *"MiR-203"*. We show how to match a pattern against a sentence in Figure 5.3. We apply patterns by

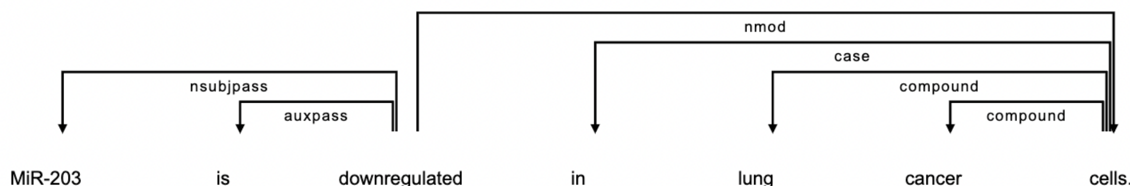


Figure 5.2: Dependency tree corresponding to the example sentence *"MiR-203 is downregulated in lung cancer cells."*. Such result was returned by means of the *DisplaCy* package.

```
matcher = DependencyMatcher(nlp_biore.vocab)
matcher.add("subject", [pattern])
matches = matcher(doc)
match_id, token_ids = matches[0]
for i in range(len(token_ids)):
    print(pattern[i]["RIGHT_ID"] + ':', doc[token_ids[i]].text)
```

Figure 5.3: Example of DependencyMatcher usage. We recall `doc` object is the parsed sentence we defined in Figure 5.2 and `pattern` is the pattern we showed in Figure 5.1.

creating a `DependencyMatcher` object that uses the same vocabulary of the pipeline we defined already, i.e. `nlp_biore.vocab`. Then we add the full pattern to the `matcher` object by means of the instruction `matcher.add(pattern_name, [pattern])`. The parameter `pattern_name` can be useful to identify the specific pattern since SpaCy `DependencyMatcher` allows to remove patterns from the `matcher` object. Instruction `matches = matcher(doc)` apply the pattern to the `doc` object passed as argument to the function and return the matches found in the document. In particular, the `matches` object is a list of all matches found by the `DependencyMatcher`. Each match contains a `match_id` which identify the specific match and a list of the identifiers of the matched tokens, namely `token_ids`. To print all matched tokens we iterate over the list of ids and return the name of the token assigned by the pattern, i.e. `pattern[i]["RIGHT_ID"]`, and the text of the token, i.e. `doc[token_ids[i].text]`. By matching the pattern in Figure 5.1 over the sentence in Figure 5.2, the code in Figure 5.3 will print "*downregulated*: downregulated" and "*subject*: MiR-203".

5.1 Rules Definition

DEXTER extract relevant information from sentences using Semgrex patterns. Such patterns are provided as supplementary files by the authors of the original paper [9]

and we have already described how these patterns were developed in Section 3.2. We also recall DEXTER classify sentences as TypeA and TypeB, as described in Section 3.1, hence we have different rules for TypeA and TypeB sentences. In this section we describe how we adapt Semgrex patterns to the `DependencyMatcher` pattern format we analyzed in the previous section. In particular, we provide an overview of TypeA and TypeB rules and we explain what changes we made and how we identify rules suitable for each sentence. For the sake of simplicity, we will use terms *patterns* and *rules* indistinctly. The first step in the RE module is to find suitable rules to apply to the sentence passed as input to the module. In particular, we pass the sentence as argument to a function that returns the applicable TypeA rules for the sentence. If we do not find any suitable rules we look for TypeB rules.

5.1.1 TypeA Patterns

As we explained in Chapter 3, TypeA sentences report gene or miRNA expressions contrasted between two different samples or conditions. Therefore, we extract the following components from such sentences: *Scale Indicator* (SI) indicates the level of expression, *Compared Aspect* (CA) is the gene or microRNA expressed in the sentence while *Compared Entity 1* (CE1) and *Compared Entity 2* (CE2) are the two entities being compared. By analyzing the TypeA patterns provided in the supplementary files of the original paper we assess each pattern could be split into two parts. The first part, which we will call *starter*, identifies SI, CA and Compared Entity 1 (CE1) while the second part of the rule, which we call *cmp_rule*, retrieve the Compared Entity 2 (CE2). This will allow us to write fewer rules and avoid repetitions. The computation of the rules selection phase for TypeA sentences is shown in Figure 5.4. As we can see, at first we check if the sentence contains any comparison word. In fact, if there is no comparison word we can directly check for TypeB rules since we assume comparative structures must contain at least one comparison word. Such words are *than*, *versus*, *vs.*, *compared* (i.e, compared with/to) and *comparison* (i.e, in comparison with/to). We first detect a starter rule by checking some trigger words and we attach to it the suitable cmp_rule, if we found any. Then we add the completed rules in a *rule_list*,

Figure 5.4: TypeA rules selection flow. We start by checking if the sentence contains any comparison word and then we check the rules applicability based on some trigger words and another function that returns the second part of the rule. We check applicability for each group of rules separately and in the end we check the *rule_list* content. If it is empty, we check for TypeB rules, otherwise we return the retrieved rules.

Table 5.1
Starter rules overview. The first column describe how we extract the Scale Indicator. In cmp1 and cmp2 rules the SI is the trigger itself, while for cmp3 rules the SI is a dependent of the trigger. The second column comprises the name of each rule and in the last column we provide a small description of the main differences among rules.

Scale Indicator	Rule Name	Rule Description
cmp1_trigger	cmp1_n0	CE1 depends on SI
	cmp1_n1	CE1 depends on CA
cmp2_trigger	cmp2_n0	CE1 depends on SI
	cmp2_n1	CE1 depends on CA
Dependency "amod"	cmp3_n0_amod	CE1 depends on n0
	cmp3_n1	CE1 depends on CA
Dependency "xcomp" or "acl"	cmp3_n0_xcomp_n0	All other components depends on cmp3_trigger
	cmp3_n0_xcomp_SI	CA depends on cmp3_trigger, others depends on SI

which is a list of dictionaries, each containing a rule we can apply. Each dictionary comprises the following keys: *name*, where we store the complete name of the rule, *starter* where we store the starter pattern to apply and *cmp*, which contains the pattern corresponding to the retrieved cmp_rule. At the end of the checks, if the rule list is empty we check if any TypeB rule is suitable for the sentence, otherwise we return the list of rules to be applied. As we have just seen, starter and cmp_rule applicability is based on different aspects thus we develop two different functions to detect the two parts.

TypeA Starter Rules

Starter rules can be divided into three main groups, based on the trigger words present in the sentence and on the role of the SI. The names of these groups recall the names of the patterns in the supplementary files of the original paper, which are *cmp1*, *cmp2* and *cmp3* rules. We provide a full overview of all the starter rules in Table 5.1. *cmp1* rules apply to sentences where the SI is an adjective such as *higher/lower* and most components depend on it. *cmp2* rules work similarly but in this case the SI is a past participle verb like *"increased"* or *"reduced"*. On the other hand, *cmp3*

rules have a different structure. In fact, we check for triggers like *"find"* , *"detect"*, *"observe"* or *"discover"* while the SI depends on such trigger words as an adjective or as a complement.

cmp1 rules comprises patterns where both CA and CE2 depends on the SI while the CE1 may depend on the SI (i.e, *cmp1_n0*) or on the CA (i.e, *cmp1_n1*). Semgrex patterns in Stanford CoreNLP has the OR operator among nodes, hence they can collapse rules *cmp1_n0* and *cmp1_n1* into one. At the time of writing SpaCy DependencyMatcher does not have the OR operations among tokens thus we need to write two separate rules. When developing the pattern, we use as anchor token the SI by checking for a list of triggers, namely *cmp1_triggers*, which actually comprises only words *"high"* and *"low"*. As a result, the attributes for the anchor node, which we name *n0*, for cmp1 rules are: `"RIGHT_ATTRS":{"LEMMA":{"IN":cmp1_triggers}}`. We use the word lemma so that we can have a shorter list of triggers which comprises all word forms. The CA is the subject whose head is identified in the SI. Hence the attributes used to match the token corresponding to the CA is: `"RIGHT_ATTRS":{"DEP":"nsubj"}`, while we use the operator `"n0 >> compared_aspect"` which indicates the scale indicator is the head in a chain to the compared aspect, following head → dep paths. About the CE1, it is attached as a noun modifier or as an adverbial clause modifier either to the SI (in rule *cmp1_n0*) or to the CA (in rule *cmp1_n1*). Therefore, to match CE1 we check among n0's dependants following head → dep paths (i.e, operator ">>") the tokens with the following attributes: `{"DEP":{"IN":["nmod","advcl",'conj']}}` and `{"ORTH":{"NOT_IN":["compared","comparison","Compared","Comparison"]}}`. The former represents that CE1 is a noun modifier (label *"nmod"*) or an adverbial clause modifier (label *"advcl"*), while the latter checks we are not matching the CE2. In *cmp1_n1* rule, the only difference is that we check among CA's dependants. Finally, we check there is a word *"in"* that depends on the CE1, hence we check for a token among CE1's dependants whose attributes are: `{"DEP":"case", "ORTH":{"IN":["in","In"]}}`. We provide an example of a sentence where we can apply cmp1 rules in Figure 5.5. In particular, we display dependencies for the sentence *"HLJ1 expression was lower in tumors than in adjacent normal tissue."*. Here we can clearly see how we apply our rules. The SI (i.e, node `"n0"`) is identified in

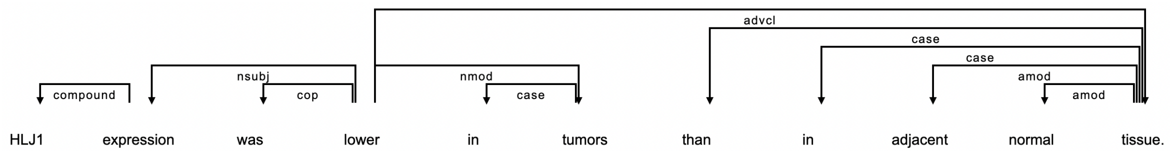


Figure 5.5: Example of a sentence that matches cmp1 rules. We visualize the dependency tree of the `Doc` object by means of displaCy.

token *"lower"* and both CA and CE1 depends on it, thus we will match starter rule *cmp1_n0*. In particular, CA is identified in the token *"expression"*, that is the subject of the sentence and the CE1 is a noun modifier detected in *tumors*. In the end, the token *"in"* which depends on the CE1 is retrieved. To conclude, we check the applicability of cmp1 rules by checking if the sentence contains any words inside the *cmp1_triggers* list. If so, we check if there is a *cmp_rule* applicable to the sentence and if we found one we add two rules to the *rule_list*. One with *cmp1_n0* as starter and one with *cmp1_n1*.

The only difference between cmp1 and cmp2 rules is on the the trigger words used to detect the SI, thus the same reasoning applies to cmp2 rules. We change the trigger list in the anchor token definition with the list *cmp2_triggers* and the CA can be a subject (dependency *"nsubj"*), a passive subject (dependency *"nsubjpass"*) or a direct object (dependency *"dobj*"). The rest of the rule structure is the same as for *cmp1* rules. We provide an example of a sentence that matches these rules in Figure 5.6, where we displayed the dependencies of sentence *"MiR-205 was upregulated in OC tissues and cells in comparison to the controls."*. In this case the SI is token *upregulated* and both CA and CE1 depends on it. In particular, the CA is the passive subject (with dependency *"nusbjpass"*), *"MiR-205"* while the CE1 is the noun modifier (with dependency *"nmod"*) *"tissues"*. In this sentence we can see an example of the additional `"ORTH"` rule we apply to CE1. In fact, without such rule the DependencyMatcher would have returned two matches for the sentence, one where the CE1 is the correct one and one where the CE1 is identified in token *"comparison"* which will be discarded in later modules. The applicability check for cmp2 rules work is identical to the cmp1 rules but in this case we check for words in *cmp2_triggers* list.

In *cmp3 rules*, components do not depends on the SI itself but on a verb belonging to the *cmp3_triggers* list. Such verbs are *"appear"*, *"demonstrate"*, *"show"*, etc. These

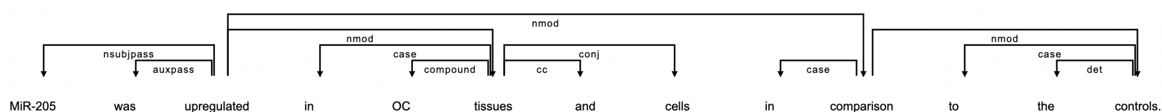


Figure 5.6: Example of a sentence that matches cmp2 rules, since the SI is a cmp2_trigger, namely *"upregulated"*. We visualize the dependency tree of the `Doc` object by means of displaCy package.

sentences have a complex structure, hence we have defined four different starters, as shown in Table 5.1. In this set of rules node *n0* and the SI do not always coincide as in the cmp1 and cmp2 rules. The only case in which SI and node "`n0`" identify the same token is in starter *cmp3_n0 _xcomp_SI*, while in all other rules the node "`n0`" is identified in the cmp3 trigger word. We decide to use this notation so that we can write fewer cmp_rules. As in previous starters, the anchor node is identified by checking which token has the word lemma that belongs to the *cmp3_triggers* list. The SI can be an adjective depending on the CA, i.e. dependency *"amod"*, or a complement depending on node "n0", i.e. dependency *"xcomp"*. Therefore, we can make a first distinction between rules *cmp3_n0 _amod, cmp3_n1* and *cmp3_n0_xcomp_n0, cmp3_n0_xcomp_SI*. As the names suggest, the first two rules apply to sentences where the SI is an adjective depending on the CA while in the other two it is a complement depending on the verb belonging to *cmp3_triggers* list.

In *cmp3_n0_amod* rules we identify as anchor node, i.e. node "`n0`", the token whose lemma belongs to the *cmp3_triggers* list. The CA is the subject (dependency *"nsubj"*), passive subject (dependency *"nsubjpass"*) or direct object (dependency *"dobj"*) depending on node "`n0`". The CE1 is returned as for the rules of the previous groups but in starter *cmp3_n0_amod* CE1 depends on node "`n0`" while on rule *cmp3_n1* it depends on the CA. Finally, the SI depends on the CA thus we use the operator >> from CA to SI, which is defined with token name *"scale_indicator"*. The attributes we check for the SI are: `{"DEP":"amod", "TAG":{"IN":["JJ","JJR","VBN","VBD"]}}`. In fact, we further verify the SI is an adjective by checking its fine-grained POS tag (i.e, `tag_` attribute) is either "JJ" (*adjective*), "JJR" (*comparative adjective*), "VBN" (*past participle verb*) or "VBD" (*past verb*). Sentence *"We found significantly higher TGF-β1 levels in lung cancer patients than in healthy individuals."* matches starter rule *cmp3_n1*. We provide its dependency tree in in Figure 5.7, where we only display the part we

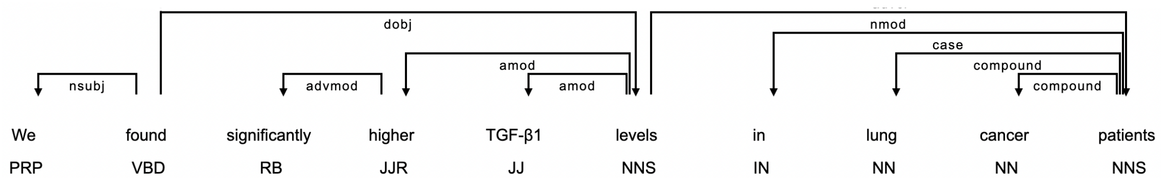


Figure 5.7: Sentence that matches *cmp3-amod* rules, we provide just the part of interest of the sentence due to space issues. The full sentence is *"We found significantly higher TGF-β1 levels in lung cancer patients than in healthy individuals."*. We visualize a part of the dependency tree together with the POS tags of each token by means of displaCy package.

are interested in so that we have a more readable image. The cmp3 trigger in the sentence, i.e. anchor node `n0`, is the token *found*. We use the dependency *"dobj"* to extract the CA, which is token *levels*. CE1 is a noun modifier, i.e. dependency *"nmod"*, whose head is the CA and it is identified in token *"patients"*. Finally, we identify the SI in token *"higher"*, which is an adjective (i.e, dependency *"amod"*) depending on the CA. We can also verify that the fine-grained POS tag of the SI is "JJR" (*comparative adjective*) , as requested by the rule.

The only difference between the previous rules and starter rule *cmp3_n0_xcomp_n0* is in the SI definition. In fact, in this rule the SI depends on the anchor node instead of CA, thus we use the operator >> from n0 to SI, which is still defined with token name *"scale_indicator"*. We check the same POS tags as before but now the SI depends on the verb as a open clausal complement (i.e, dependency *"xcomp"*), clausal complement (i.e, dependency *"ccomp"*) or as a clausal modifier of noun (i.e, dependency *"acl"*). Therefore, the attributes for token *scale_indicator* are: `"DEP":{"IN":["ccomp","xcomp","acl"]}` and `"TAG":{"IN":["JJ","JJR","VBN","VBD"]}`. Starter rule *cmp3_n0_xcomp_SI* differs from all other rules in group *cmp3*. In fact, only CA and SI depends on the cmp3 trigger word while all other components depends on the SI itself. For this reason, we change the names of the matched tokens. In this case, the anchor token is defined as node `"n1"` while the SI is defined as token `"n0"` as we did for groups *cmp1* and *cmp2*. The rules for extracting CA, CE1 and SI are similar to rule *cmp3_n0_xcomp_n0*. What changes is that the head of CA and SI is identified in node `"n1"` (i.e, the cmp3 trigger word), while CE1's head is node `"n0"` (i.e, the SI). We can apply starter rule *cmp3_n0_xcomp_SI* to the sentence *"SCUBE3 was found to be up-regulated in NSCLC tissue samples compared to adjacent normal tissue."* .

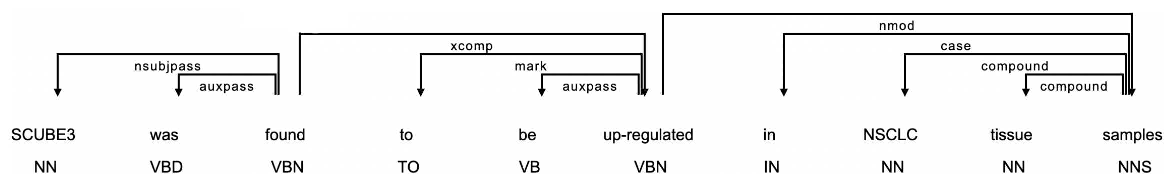


Figure 5.8: Sentence that matches *cmp3-xcomp* rules, we provide just the part of interest of the sentence due to space issues. The full sentence is *"SCUBE3 was found to be up-regulated in NSCLC tissue samples compared to adjacent normal tissue."*. We visualize a part of the dependency tree together with the POS tags of each token by means of displaCy package.

We provide a part of the dependency tree of such sentence together with the tags for each token in Figure 5.8. We can clearly see that the CA and SI depends on the anchor node `"n1"`, that is token *"found"*, while the CE1 depends on the SI, that is *"up-regulated"*. It is important to notice that *cmp_rules* depends on the node `"n0"` we extract from starter rules, thus when deciding which *cmp_rule* to apply we pass as argument to the function the trigger list whose node "n0" belongs to. While for *cmp1* and *cmp2* rules node "n0" could only belong to the corresponding trigger lists, in this case only the first three rules we describe identify node "n0" with a cmp3 trigger word. In rule *cmp3_n0_xcomp_SI*, node "n0" can be identified either with a cmp1 or cmp2 trigger word. Therefore, we perform two different checks for *cmp3* rules applicability. Firstly, we check if the sentence contains at least one cmp3 trigger. If so, we check if there is one *cmp_rule* applicable when node "n0" is a cmp3 trigger word. If so, we add the first three starters to the *rule_list*. If there is no suitable *cmp_rule*, we check if there is one *cmp_rule* applicable when node "n0" is a cmp1 or cmp2 trigger word. If so, we only add the rule with starter *cmp3_n0_xcomp_SI* to the *rule_list*.

TypeA Cmp Rules

Once the potential starter rules is identified, we check for the second part of the rule, namely *cmp_rule*, which will extract the last component, i.e. CE2, together with the token corresponding to the comparison word. We developed a second function to identify the suitable *cmp_rule* for the sentence. This function takes as arguments the sentence itself as a `Doc` object, a list of comparison words detected in the sentence and the list of trigger words which comprises node "n0" lemma. We recall that this function is called after we have already detected the potential starter of the rule

Table 5.2
Cmp rules overview. The first column indicates the considered comparison word, which can be *"than in"*, *"than"*, *"versus/vs. in"*, *"versus/vs."* or *"compared/comparison"*. The second column comprises the name of each cmp_rule and in the last column we provide a small description of the main differences among patterns.

Comparison Word	Rule Name	Rule Description
"than in"	than_1_SI	CE2 depends on SI
	than_1 _CE1	CE2 depends on CE1
"than/versus/vs."	than_2_SI	CE2 depends on SI
	than_2_CE1	CE2 depends on CE1
"versus/vs. in" (dep. "case")	vs_1_SI	CE2 depends on SI
	vs_1_CE1	CE2 depends on CE1
"versus/vs. in" (dep. "cc")	vs_2	CE2 depends on CE1
"compared/ comparison"	compare_1_SI	CE2 depends on SI
	compare_1_CE1	CE2 depends on CE1
	compare_2	CE2 depends on comparison word, which depends on SI
	compare_3	CE2 depends on comparison word, which depends on CE1

hence we know which list contains the anchor node. In this function, which is called *find_cmp_rule*, we loop over the comparison words in the list passed as argument and as soon as we find a suitable *cmp_rule* we return it. The first thing we check is the comparison word, hence we can divide the rules based on its text form. A full list of *cmp_rule* is provided in Table 5.2.

If the comparison word is *"than"* we have four different rules which depends on whether token *"than"* is followed by the preposition *"in"* and based on CE2's head. Therefore, we can make a first distinction between *than_1* and *than_2* rules. *than_1* rules match sentences where the comparison word *"than"* is followed by the preposition *"in"*. With these rules we will extract three components: CE2, the comparison word (identified as token *"case_than_2"*) and the preposition "in" that follows the comparison word, which we call *"case_in_2"*. We recall we detected a preposition also before the CE1, thus we firstly need to check that we have not detected

the CE2 in the starter rule. To do so, we check if the token "*in*" we matched before does not immediately follows the comparison word. This translates in the operator `";"` between node *case_in_1* and node *case_than_1* and the attributes for node *case_than_1* are: `"{"DEP":"case", "ORTH":"than", "OP":"!"}"`. The attribute `"OP"` defines how many times we want to match the token in the rule and in our case `"!"` negates the pattern, hence we do not want any token to match such rule. CE2 is a dependant whose label can be "*nmod*" (nominal modifier), "*advcl*" (adverbial clause modifier) or "*dep*" (generic dependency). What differs between rule *than_1_SI* and *than_1_CE1* is the head of the CE2. In fact, in the first rule the CE2 depends on node `"n0"`, which can be the SI or a cmp3 trigger, while in the other one the CE2 depends on CE1, as the name suggests. Therefore, we can extract the CE2, which will be identified as *compared_entity_2*, by checking which token depends on either node n0 or CE1 with dependency labels among the one mentioned before and with operator >>. Once we have detected the CE2 we can extract tokens "*case_than_2*" and "*case_in_2*" by checking CE2's children (i.e, operator >) with dependency label "*case*" for both tokens. Then for node "*case_than_2*" we check the token text (i.e, attribute `"ORTH"`) is "*than*", while for "*case_in_2*" the word must be "*in*". We have an example of rule *than_1_SI* in Figure 5.5. Here we notice the comparison word is "*than*" followed by preposition "*in*" and the first check we made allow us to correctly discard token "*tissue*" as CE1. We can also see that CE2, which is identified in token "*tissue*", depends on the SI as an adverbial clause modifier (dep "*advcl*") and both "*than*" and "*in*" depends on the CE2 with dependency label "*case*". From this example we notice we can check if the CE2 depends on the SI by checking if the lemma of `cmp_word.head.head` is a cmp1 or a cmp2 trigger word. In fact, attribute `head` returns the head of the token it is applied to and we can follow the dependency tree path by concatenating such attribute. Therefore, `cmp_word.head` returns the CE2, since comparison word "*than*" always depends on the CE2, thus token `cmp_word.head.head` returns the head of CE2. If CE2 depends on CE1 instruction `cmp_word.head.head` will return the CE1, which may directly depend on the SI or follow a path CE1->CA->SI. For this reason, to check if CE2 depends on CE1 we check if the lemma of either `cmp_word.head.head.head` (if CE1 directly depends on SI this will return the SI) or `cmp_word.head.head.head.head`

(if CE1 follows the path `CE1->CA->SI` the previous token would be the CA hence we need to do one more hop) is a cmp1 or cmp2 trigger word. To sum up, to check which *than_1* rule to apply we firstly check if the comparison word is *"than"*. Then we check if "than" is followed by preposition *"in"* and if so, we check if CE2 directly depends on the SI (in this case we return rule *than_1_SI*) or on the CE1 (resulting in rule *than_1_CE1*) by applying the reasoning we explained above. If *"than"* is not followed by preposition *"in"* we check for *than_2* rules.

than_2 rules match sentences where the comparison word is *"than"* but it is not followed by preposition *"in"*. This is the case of sentence *"The serum level of Mac-2BP was significantly higher in lung cancer patients than healthy controls."*. This set of rules comprises two patterns which differs based on CE2's head. Rule *than_2_SI* is applied when CE2 directly depends on the SI while we use *than_2_CE1* when the head of CE2 is the CE1. Therefore we check which rule we can apply as we did above for *than_1* rules. In this case we only extract two tokens, namely *compared_entity_2*, which is the CE2, and *case_cmp_2*, which identifies the comparison word. CE2 is extracted with the same rule we used for *than_1* rules and we use a different head based on which rule we are applying. About *case_cmp_2*, we extract such token as we did for token *case_than_2* in the previous group of rules but we modify attribute `"ORTH"` as follows: `"{"ORTH":{"IN":["than", "versus", "vs."]}}"`. In fact, we will use the same set of rules for comparison word *"versus"/"vs."* when it is not followed by preposition *"in"*.

If the comparison word is *"versus"* or *"vs."*, we can apply five different rules that are based on the dependency label of the comparison word. Therefore, the first thing we check is the dependency label of tokens *"versus"/"vs."*. If it is *"case"*, we can apply *"vs_1"* rules, that works as *"than _1"* rules but we substitute `"{"ORTH":"than"}"` with `"{"ORTH":{"IN":["versus","vs."]}}"`, or we can apply *than_2* rules, if the comparison word is not followed by preposition *"in"*. Comparison words *"versus"/"vs."* can also have dependency *"cc"* when *"versus"* is the coordinating conjunction word between CE1 and CE2. In this case we apply rule *vs_2*, which extracts the CE2 and the comparison word identified in token *case_cmp_2*. CE2 is a conjunctive clause whose head is the CE1 hence we match the token with operator ">>" between CE1 and the

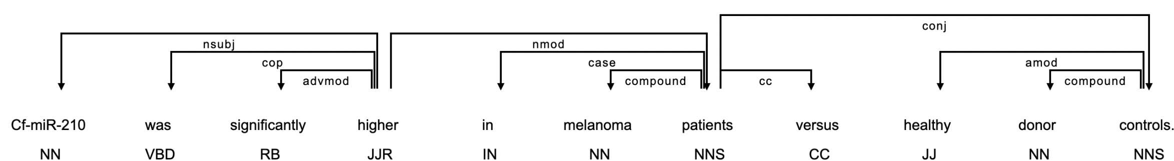


Figure 5.9: Sentence that matches *vs_2* cmp rule, in particular this sentence matches rule *cmp1_n0_vs_2*. We visualize the dependency tree together with the POS tags of each token by means of displaCy package.

new token *compared_entity_2*, using attributes: `{"RIGHT_ATTRS":{"DEP":"conj"}}`. Finally, the comparison word is extracted by checking which CE1's children (i.e, operator ">") has dependency label *"cc"*. Figure 5.9 provides an example of such structure. We can see that CE2 is a conjunctive clause depending on CE1 hence we follow dependency *"conj"* to extract the CE2. The comparison word also depends on CE1 with dependency label *"cc"*.

If the comparison word is *"compared"* or *"comparison"* we can apply four different rules that can be divided into two categories based on the head of CE2. In fact, if CE2 depends directly on the comparison word we will use either rule *compare_2* or *compare_3*, otherwise we choose between *compare_1_SI* or *compare_1_CE1*. To distinguish between the two groups we check the comparison word's children, if none of them has a dependency label *"nmod"* we check which *compare_1* rule to apply, otherwise we check for the other rules. *compare_1* rules work like *than_1* and *vs_1* rules. The only difference is that in token *case_compared* we match the token with attribute `"{"ORTH":{"IN":["compared", "comparison", "Compared","Comparison"]}}"`. The distinction between *compare_1_SI* and *compare_1_CE1* is the same of *than_1* case and the selection function works identically.

compare_2 and *compare_3* match sentences where the CE2 directly depends on the comparison word. The only difference between the two rules is that in rule *compare_2* the comparison word directly depends on node `"n0"`, while in *compare_3* it directly depends on the CE1. Here, we extract token *case_compared* by checking tokens that depends on n0 or CE1 (i.e, we use operator ">>") and matching attributes: `"DEP":{"IN": ["xcomp","advcl","nmod","dep","prep"]}, "ORTH":{"IN": ["compared", "comparison","Compared","Comparison"]}`. Then we extract the CE2 by checking the comparison word's dependants (i.e, we use

operator ">>") and matching the token with dependency label *"nmod"*. Sentence in Figure 5.6 matches rule *compare_2*. In fact, the comparison word *"comparison"* is a nominal modifier (dep *"nmod"*) whose head is node "n0", which in this case corresponds to the SI. Instead, the CE2 is a nominal modifier depending on the comparison word. Overall, once we detected both the starter rule and the suitable cmp rule we can add the complete rule to the *rule_list*. By following the flow in Figure 5.4 for example in Figure 5.6, we saw that such sentence comply with cmp2 rules requirements and we found a suitable cmp rule in *compare_2*. Therefore, we will append two dictionaries to the *rule_list*, which are {*"name"*: "cmp2_n0_compare_2", *"starter"*: cmp2_n0, *"cmp"*: compare_2} and {*"name"*: "cmp2_n1_compare_2", *"starter"*: cmp2_n1, *"cmp"*: compare_2}.

5.1.2 TypeB Patterns

If there is no comparison word inside the sentence or if we find no suitable TypeA rule, we check if any TypeB rule applies to the sentence. We recall TypeB sentences contain only one entity where the expression of the gene or miRNA has been detected. In the original paper the components extracted for TypeB sentences have a different name (*level indicator (LI)*, *expressed aspect (EA)* and *expressed location (EL)*), however we decided to keep the same names used for TypeA sentences so that we can access both TypeA and TypeB components with the same lines of code. In particular, EL will be called SI, EA is the CA and the EL is the CE1. We divide rules in *starter* and *cmp* as before but in this case we also divide rules into two classes, namely *exp* and *fnd* rules since the two set of rules uses different starter and cmp patterns. In *exp* rules the starter part of the pattern extract the SI and CA and this set of rules comprises patterns where the CE1 can depend either on the SI and the CA. On the other hand, in *fnd* rules we only extract the CA and an anchor node "n0" in the starter rule while the CE1 and SI are extracted in the cmp part. In this case, both CE1 and SI can depend either on the CA or on node n0, which is a cmp3 word trigger. This is why we have developed different starter and cmp rules for the two classes.

Table 5.3
Starters for *exp rules*. In the first column we provide the name of the starter rule. In the second and third column we describe how we extract the SI and the CA in the different rules. In all cases we check if the lemma of the potential SI belongs to *cmp12B_triggers* list, however additional checks are required to correctly extract the SI in the second and third rules.

Starter Name	SI	CA
subj_exp	cmp12B_triggers	subject depending on SI
conj_exp	dep *"conj"*, head: *conj_clause*	subject depending on *conj_clause*
appos_exp	dep *"conj"*, head: *CA*	head of *"appos"* dependency

TypeB "exp" Rules

exp rules comprises patterns where the SI is usually a cmp1 or cmp2 trigger word that is central in the extraction of the other components. In fact, all components directly depend on it or follow a path dep->head to the SI. We split rules into starter and cmp as before, however for this type of rules we just check which starter to apply and then add to the *rule_list* all applicable starters with the two cmp rules available for this class of patterns without any further check. One important note is that we are still using the same lists of triggers we defined for TypeA rules, however we do not need to distinguish between cmp1 and cmp2 triggers thus we merged the two into one list called *cmp12B_triggers*.

We provide a full overview of starter rules for *exp* patterns in Table 5.3. In all rules we check if the lemma of the potential SI belongs to *cmp12B_triggers* list, however both SI and CA differs in all three rules. In *subj_exp* rule, the SI is the anchor token and we just check if its lemma belongs to the above-mentioned list. Then the CA is extracted by checking the dependants of node SI, using operator ">>", that have a dependency *"nsubj"*, *"nsubjpass"*, *"dep"*, *"dobj"* or *"acl"*. In *conj_exp* rule, the anchor token is the head of the SI and we extract it following dependency *"conj"*. The CA is still the subject of the sentence thus we check the same dependencies as before but now the head of the relation is the new anchor node. In *appos_exp* rule, the anchor token is a dependant of the CA, which is the head of the dependency *"appos"*. That means the anchor node is an appositional modifier of the CA. The SI is extracted by checking the dependant of the CA with dependency label *"conj"*. We check the

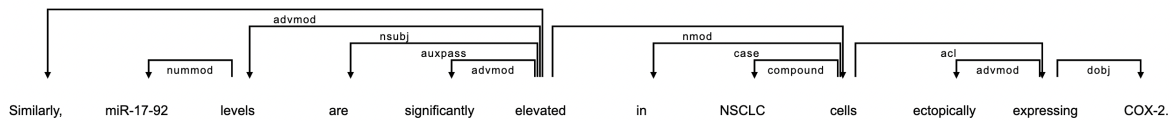


Figure 5.10: Sentence that matches *subj_exp* starter rule, in particular this sentence matches rule *subj_expressionIn_1*. We visualize the dependency tree by means of displaCy package.

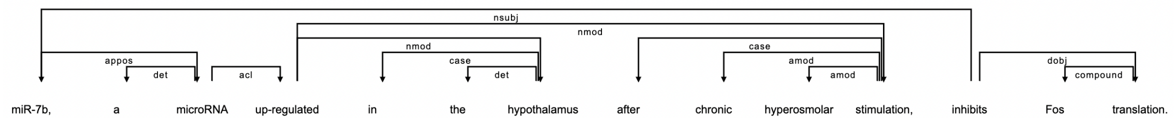


Figure 5.11: Sentence that matches *appos_exp* starter rule, in particular this sentence matches rule *appos_expressionIn_1*. We visualize the dependency tree by means of displaCy package.

applicability of each starter rules by detecting any cmp12B trigger word in sentence as potential SI. If any potential SI is head of a subject we can add *subj_exp* rule to the starter rules. If any of them has a dependency label *"conj"*, *"acl"* or *"dep"* we also add *conj_exp* rule. Finally, if any potential SI has a dependency label *"conj"*, *"acl"* or *"dep"* and its head is also the head of an *"appos"* dependency we add rule *appos_exp* to the starters.

We have two cmp patterns that we can apply to this group of rules: *expressionIn_1* and *expressionIn_2*. The only difference between these two rules is that in the former the CE1 depends on the SI while on the latter it depends on the CA. Therefore, we extract the CE1 as a dependant of either SI or CA (we use operator >>) with label *"nmod"* or *"dobj"*. We assume the CE1 is preceeded by a preposition *"in"* hence we extract it by checking the CE1's dependant and matching the one with dependency label *"case"* and word form *"in"*. Once we have determined the suitable starters, for each of them we add two rules, one for each cmp, to the *rule_list* without any further checks. We provide an example of *subj_exp* rule in Figure 5.10. This sentence matches rule *subj_expressionIn_1* since both the CA, that is *"levels"*, and the CE1, identified in *"cells"*, depend on the SI, namely *"elevated"*. In particular, the CA is the subject of the sentence and the CE1 is a noun modifier whose head is the SI. Sentence in Figure 5.11 matches *appos_expressionIn_1*. As we can see, the structure of this sentences allows us to extract information from sub-sentences enclosed between comas that refers to the subject of another sentence, in this case *"miR-7b"*, which is also the CA. In this case

Table 5.4
Starters for *fnd rules*. In the first column we provide the name for each starter while the following columns provide a description of each pattern. The first column describes how we extract node "n0", which is a cmp3 word trigger. The last column focus on CA extraction, which differs only on the head of the dependency.

Starter Name	n0	CA
subj_fnd	cmp3_triggers	subject depending on n0
conj_fnd	dep *"conj"*, head: *conj_clause*	subject depending on *conj_clause*

the anchor token is *"microRNA"*, which is a dependant of the CA with label *"appos"* and it is also the head of the SI, namely *"up-regulated"*, which is also the head of the CE1, identified in token *"hypothalamus"*.

TypeB "fnd" Rules

fnd rules comprise patterns where most components do not depend on the SI but on a cmp3 word trigger, which we will define as *"n0"*. This group of rules resembles cmp3 rules that we saw for typeA sentences and matches sentences like *"We found that X is higher in Y"*. Therefore, we firstly check if the sentence contains any cmp3 trigger word to see if we can apply any of these rules. If so, we proceed with further checks otherwise we skip all *fnd* rules. In this case, the starter rule extracts the CA and node "n0" while the cmp rule extracts both the SI and the CE1.

fnd rules comprise two different starter rules that are shown in Table 5.4, that differ both in the CA and n0 extraction. In *subj_fnd* rule the anchor token is the cmp3 trigger itself hence we detect node "`n0`" as the token whose lemma belong to the *cmp3_triggers* list. Then, the CA is the subject who depends on node `n0`. This means we check for n0's dependants with the operator ">>" whose dependency label is either *"nsubj"*, *""nsubjpass"*, *"dep"*, *"acl"* or *"dobj""*. For this reason, in order to see if *subj_fnd* can be applied we simply check if any cmp3 trigger word has a children with any of the above-mentioned dependencies. We provide an example of a sentence that matches *subj_fnd* starter rule in Figure 5.12. Here the anchor node is identified in token *"found"* and the CA is the passive subject (dep. *"nsubjpass"*) of node n0, namely token *"levels"*. In *conj_fnd* starter the anchor node is no longer n0 but a conjuction

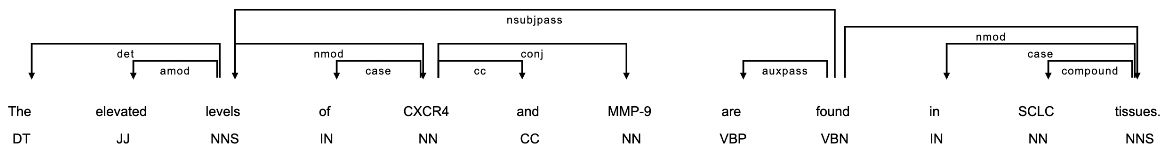


Figure 5.12: Sentence that matches *subj_fnd* starter rule. We visualize the dependency tree together with the POS tags of each token by means of displaCy package.

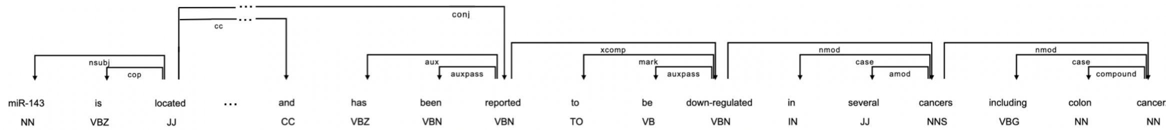


Figure 5.13: Sentence that matches *conj_fnd* starter rule. We visualize only part of the dependency tree together with the POS tags of each token by means of displaCy package.

clause, which we actually define as *conj_clause*, that is the head of both CA and n0. In this case we extract node "n0" by following the dependency *"conj"* from node *conj_clause* while the CA is extracted by searching among the same node's dependant the dependency labels we used in the previous rule. We provide an example of a sentence that matches such rule in Figure 5.13. The full sentence is *"miR-143 is located at a fragile site on chromosome 5 frequently deleted in cancer, and has been reported to be down-regulated in several cancers including colon cancer."*, however we report only the dependency tree of the part of interest so that we provide a more readable image. Here we can see that the anchor node is token *"located"* which is the head of both the CA, that is token *"miR-143"*, and node n0, which is identified in token *"reported"*. In this case we extract the CA by following dependency *"nsubj"* and node n0 by following dependency *"conj"*. With this rule we are able to extract information about the subject even if we have a longer sentence composed of different sub-phrases that refers to the same subject. We assume starter *conj_fnd* is applicable to a sentence if any cmp3 trigger words has dependency label *"conj"* and the head of such token is the head of a dependency *"nsubj"*, *""nsubjpass"*, *"dep"*, *"acl"* or *"dobj""*.

If we find any applicable *fnd* rule, we check for the cmp rules. Differently from *exp rules*, in this case we do not add all cmp rules to the starters we found but we also check which are the suitable ones to extract both the CE1 and the SI. In total, we can apply five different cmp rules to *fnd* rules, which differs both on the SI and CE1 extraction. We provide an overview of all cmp rules in Table 5.5. We can divide the

Table 5.5
Cmps for *fnd rules*. In the first column we provide the name for each starter while the following columns provide a description of each pattern. The first column describes how we extract the SI, which may be an adjective or an adverb and it may depend on CA or node n0 with different dependency labels. The last column focus on CE1 extraction, which may depend on the CA, SI, node n0 or on another node that we defined as *"adverb"*.

Rule Name	**SI**	**CE1**
foundIn_1	adjective, dep *"amod"*, head: CA	depends on n0
foundIn_2	adjective, dep *"amod"*, head: CA	depends on CA
EXPfoundIn _xcomp_1	adjective, dep *"xcomp"*, head: n0	depends on SI
EXPfoundIn _xcomp_2	adjective, dep *"xcomp"*, head: n0	depends on n0
RBfoundIn _xcomp	adverb, dep *"xcomp"*, head: n0	depends on *adverb*

cmp rules into three groups, based on the SI form and dependency label. In fact, in the first four rules the SI is an adjective, therefore we will check if its tag is "JJ", "JJR", "VBN" or "VBD", while in the last rule the SI is an adverb, i.e, it has tag "RB". We can further divide the first four rules based on SI dependency, which can be either an adjectival modifier (i.e, dep. *"amod"*) or a complement (i.e, dep. *"xcomp"*, *"ccomp"* or *"advcl"*). Finally, we further distinguish rules based on the head of the CE1 and in all rules we extract the preposition *"in"* which preceeds the CE1 as we saw in the cmp rules for the *exp* group.

foundIn rules comprise patterns where the SI is an adjectival modifier that depends on CA. Therefore, after checking the SI is an adjective, we can extract the SI by looking among CA's dependants and look for one with dependency label *"amod"*. The CE1 is extracted as in the cmp rules for *exp rules*, the only difference is in the head of the relation. In fact, in rule *foundIn_1* the head is node n0, while in rule *foundIn_2*, it is the CA. We add these two cmp rules to the determined starters if there is any adjective in the sentence with dependency label *"amod"*.

RBfoundIn_xcomp rule matches sentences where the SI is an adverb, i.e. with tag *"RB"*. In this case we extract the SI by searching among the dependants of node n0 one token that is an adverb with dependency label *"xcomp"* or *"ccomp"*. The CE1 does not directly depend on the SI but we extract an intermediate node defined as *"adverb"* which depend on the SI as a clausal modifier of noun (dep. *"acl"*). This

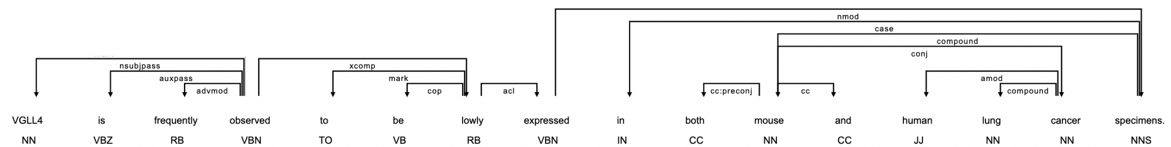


Figure 5.14: Sentence that matches *RBfoundIn_xcomp* cmp rule. In particular, we can apply the rule *subj_fnd_RBfoundIn_xcomp*. We visualize only part of the dependency tree together with the POS tags of each token by means of displaCy package.

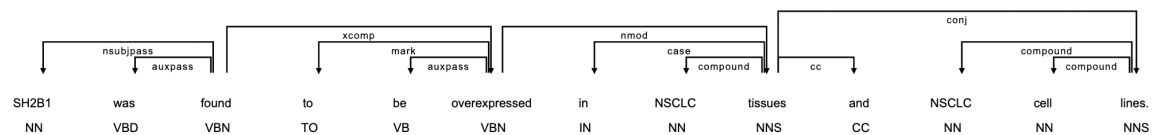


Figure 5.15: Sentence that matches *EXPfoundIn_xcomp* cmp rule. In particular, we can apply the rule *subj_fnd_EXPfoundIn_xcomp_1*. We visualize the dependency tree together with the POS tags of each token by means of displaCy package.

node is the head of the CE1, which depends on it with dependency label *"nmod"* or *"dobj"*. We use *RBfoundIn_xcomp* as cmp rule if there is an adverb in the sentence with dependency label *"xcomp"* or *"ccomp"* whose head is a cmp3 trigger. We provide an example of such rule in sentence *"Our data show that VGLL4 is frequently observed to be lowly expressed in both mouse and human lung cancer specimens."*. We show only the part of interest of the dependency tree in Figure 5.14 so that the image is more readable. In particular, we can notice the sentence matches rule *subj_fnd_RBfoundIn_xcomp*, as the CA, identified in *"VGLL4"* directly depend on node n0, that is *"observed"*. The SI, identified in adverb *"lowly"*, depends on node n0 with label *"xcomp"* and we extract the CE1, identified in token *"specimens"*, using the intermediate node *"expressed"* as *"adverb"* token.

EXPfoundIn_xcomp rules comprise patterns where the SI is an adjective which depends on node n0 with dependency label *"xcomp"*, *"ccomp"* or *"advcl"*. CE1 is extracted as a noun modifier (dep. *"nmod"*) or direct object (dep. *"dobj"*) depending either on the SI (in *EXPfoundIn_xcomp_1* rule) or on node n0 (in *EXPfoundIn_xcomp_2* rule). We use these two rules as cmp to a *fnd* starter if there is an adjective in the sentence with dependency label *"xcomp"*, *"ccomp"* or *"advcl"* whose head is a cmp3 trigger word. We provide an example of sentence where we can apply *EXPfoundIn_xcomp* rules in Figure 5.15. In particular, we can notice the sentence matches

rule *subj_fnd_EXPfoundIn_xcomp_1* since the CA, identified in *"SH2B1"*, is the passive subject of node n0, that is *"found"*. The cmp rule we apply is *EXPfoundIn_xcomp_1*, since the SI (*"overexpressed"*) is an adjective, actually it is a past participle verb (i.e, tag "VBN"), which depends on node n0 with label *"xcomp"*. Finally, the CE1, identified in token *"tissues"*, depends on the SI with relation *"nmod"*.

5.2 Module Overview

In this section we describe the RE module, in particular defined above and we provide an overview of the components extraction. In fact, the DependencyMatcher matches single tokens but components like CA, CE1 and CE2 are Noun Phrases (NPs), therefore we developed a function to expand the matched components. We defined the RE module by means of a function that we called `relation_extraction`, which takes as argument the input sentence parsed as a `Doc` object by the Text Processing module and the SpaCy model we defined in the previous module (see Chapter 4 for more details). The trained pipeline will be useful when defining the `DependencyMatcher` object since it uses the same `vocab` as the SpaCy model as we saw earlier. We start the computation by checking if any TypeA rule is applicable to the sentence and we return a list of such rules following the reasoning shown in Section 5.1. If the *rule_list* is empty we return another function called `re_module_b`, which performs the RE module for TypeB sentences and takes the same arguments as input. Otherwise, we define a `DependencyMatcher` object, using the instructions shown at the beginning of the chapter, and we loop over rules in the *rule_list* and apply each of them to the sentence. After applying each rule, if we found any match we extract the relevant components and add them to the final result set if it is not a duplicate. The result set is a list of dictionaries. Each dictionary is a match, hence the keys are the names of the components and the values are the components extracted for the specific match. We will describe the components extraction in detail later in this section. Then we remove the rule we have just used and add the next rule. Once we applied all rules, we check the result set. If it is empty, we return the function `re_module_b` and check if we found any TypeB matches. Otherwise we return the

result set, the sentence type (that is *"TypeA"* in this case) and the list of rules we applied, which is useful for debugging purposes and to check the computation in detail.

Function `re_module_b` works identically to the previous function, but in this case we check for TypeB rules. The only difference with the previous module is in the outputs. In fact, if we found no suitable TypeB rules or at the end of the computation there are no matches, in this case we return a custom Exception called `MatchNotFound Exception`. On the other hand, if we found any TypeB matches we return the result set, the sentence type (*"TypeB"* in this case) and the list of rules we applied.

Extracting the Components

As we briefly mentioned, the DependencyMatcher matches tokens but we are interested in extracted relevant NPs from the input sentence. For this reason we developed a series of rules that expands the matched tokens. The `matches` object returned by the matcher is usually a list and each match is composed of the *match_id* and the *token_ids*, which is a list of the identifiers of matched tokens. We provided an example of how to extract matched components in Figure 5.3. As we can see, we access the name of the matched token using instruction `pattern[i]["RIGHT_ID"]`, where `"i"` is the index of the list token_ids corresponding to a specific component. Then we can access the specific token object by means of the instruction `doc[token_ids[i]]`. When matching the input sentence with rules defined in Section 5.1, we extract some intermediate tokens that are useful to match the relevant components but have no usage in the broader computation. Therefore, we return a dictionary for each match containing only the SI, CA, CE1 and CE2 and when extracting relevant components we only focus on these tokens. We developed a function called `extract_components` which expand the matched tokens and return a list of dictionaries containing the relevant components for each match. During computation we loop over the matches of the `matches` object returned by the DependencyMatches and we focus on each match separately. As we said earlier, we are only interested in returning the SI (key: *"scale_indicator"*), CA (key: *"compared_aspect"*), CE1 (key: *"compared_entity_1"*) and, if we are considering TypeA matches, the CE2 (key: *"compared_entity_2"*) thus we

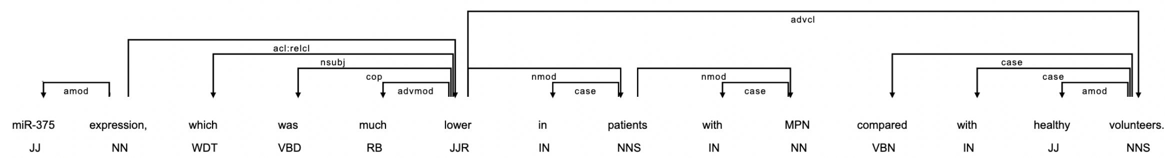


Figure 5.16: Sentence where the CA corresponds to token *"which"*. We visualize part of the dependency tree together with the POS tags of each token by means of displaCy package.

develop different expansion rules for each component.

The SI is a single token which indicates the level of expression of the CA detected in the sentence. For this reason, we do not need to expand the token but we recall in some TypeA rules, namely *cmp1*, *cmp2* and *cmp3_n0_xcomp_SI*, we named the SI as `"n0"`. Therefore, we simply change the name of the components matched for these rules to *"scale_indicator"*.

We now focus on two special cases that we may encounter especially in longer sentences. Firstly, when we write long sentences we may include sub-phrases that refers to the same subjects by means of word *"which"* or *"that"*, as in sentence *"'Finally, histone deacetylation (HDAC) inhibitors restored miR-375 expression, which was much lower in patients with MPN compared with healthy volunteers."*. We provide a part of the dependency tree for such phrase in Figure 5.16 and we can clearly notice that the subject of the sub-phrase containing relevant information is *"which"* thus the DependencyMatcher will identify this token as the CA following our rules. Such token contains no useful information for us since we would like the CA to be a gene or miRNA, hence we need to solve the reference of the determiner "which". We notice that *"which"* refers to *"expression"*, which is the head of the determiner's head, i.e. *"lower"*. For this reason, we can substitute token *"which"* by checking if its head has dependency label *"acl:relcl"* (that indeed identify a relative clause), then the head of such dependency becomes the new CA. Back to the example in Figure 5.16, this translates into identifying the CA in the token *"expression"*. The same reasoning apply if we have determiner *"that"* instead of *"which"*. Another special case is the one shown in Figure 5.17. Here, the CE2 is identified in determiner (i.e, pos "DET") *"those"* thus we need to solve again the reference problem. In this case we check the determiner dependant and we identify as the new CE2 the token with dependency

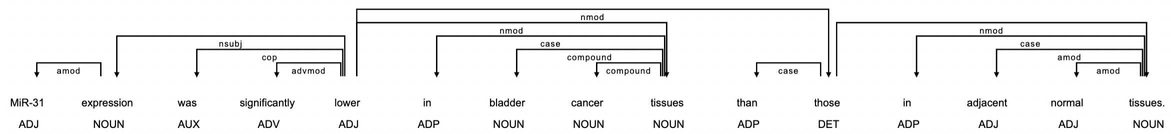


Figure 5.17: Sentence where CE2 is identified in token *"those"*. We visualize the dependency tree together with the POS tags (in this case we used coarse-grained tags, i.e, `Token.pos_`) of each token by means of displaCy package.

label *"nmod"* or *"acl"*. In Figure 5.17, this translated into identifying as CE2 token *"tissues"*. The same reasoning apply if we have *"that"* instead of *"those"*.

Once we solved all reference problems we can expand the CA, CE1 and CE2 so that they become NPs containing all relevant information. We will use sentence in Figure 5.17 as reference example throughout this part. The expansion works in two phases: firstly we expand all three components so that they contain all their children tokens and the children of their children and then we bound the start and end of the retrieved span for CE1 and CE2. About the CA, we perform some further expansions. We assume the CE1 starts with proposition *"in"* and, if we have a TypeA sentence, it ends with the comparison word. On the other hand, if we are considering TypeB sentence, we bound the end of CE1 when we find a conjunction. This will potentially include unwanted tokens inside the span however we can still extract relevant and correct information even with noise. In example in Figure 5.17, this translates into identifying as CE1 the span *"in bladder cancer tissues"*. If we have a TypeA sentence, we expand CE2 and we bound the start of the component by assuming the CE2 will start with a comparison word. If we consider Figure 5.17, this translates into identifying as CE2 the span *"than those in adjacent normal tissues"*. Additionally, we check if the next token outside the spans is a closing bracket, i.e. `")"`, and if so we add it to the span. Then, we also check if there is at least one entity inside the span and if not, we check if there is an entity right after the span and add it. To sum up, considering example in Figure 5.17, the output of the extracted component is a list of dictionaries, each containing a match with the relevant components. An example of a match for the sentence in Figure 5.17 is:

- *"scale_indicator"*: lower
- *"compared_aspect"*: MiR-31 expression

- *"compared_entity_1"*: in bladder cancer tissues
- *"compared_entity_2"*: than those in adjacent normal tissues

Chapter 6

Information Extraction

In this chapter we complete the description of our implementation of DEXTER. In particular, we present the last two modules and we explain how we handle the output of the system. Up to now, we have extracted the components from the sentence following the patterns we developed. Now we need to extract from them the relevant information for the task, which are the gene or miRNA mentioned in the sentence, the level of expression and the associated disease. The first step involves the detection of gene, miRNA and disease mentions in the input sentence, which is achieved in the *Entity Detection and Phrase Typing* module described in Section 6.1. Then, in the *Argument Filtering and Extraction* module we filter out irrelevant components we may have detected in the RE module and extract the GDAs.

6.1 Entity Detection and Phrase Typing

In the *Entity Detection and Phrase Typing* module we detect gene and disease names, microRNA mentions in the text and phrases that mention expression information or details about diseases. To achieve this goal, we developed several functions that perform the different tasks. This module takes as input the parsed sentence and its output will be useful in the last module to filter and extract relevant information from matches returned by the RE module.

Annotations

As we briefly discussed in Chapter 3, we download gene and diseases annotations from PubTator [12]. Therefore, during pre-processing, before looping over sentences, we collect all PMIDs that are present in the input file and retrieve annotations from the PubTator API[1]. We issue a POST request to the server for 1000 PMIDs at a time and the server returns a JSON response file with all annotations. Then, we build a dictionary with all annotations using PMIDs as keys. In particular, for each PMID we create a dictionary with keys: *'abstract'*, where we store the text of the abstract, *'title'*, where we store the text of the title of the article, *'genes'*, where we store a dictionary of all gene mentions for the abstract (for each key representing a gene mention we provide the corresponding gene id returned by PubTator), *'diseases'*, where we store a dictionary of all disease mentions in the abstract (for each key representing a disease mention we provide the corresponding DOID) and *'diseases_title'*, where we store all diseases mentioned in the title only. In addition, we also create a dictionary called *'general_annotations'* where we store two dictionaries, one for *'genes'* and one for *'diseases'*, containing genes and diseases mentioned in the abstracts. This allow us to retain the information about genes or diseases that PubTator may have failed to detect in one abstract, but detected in another one. Consider Example 4.1, if PubTator fails to return the disease mention *'SCLC'*, but we detected it in another abstract we will still be able to retrieve such information from this sentence.

> Example 4.1: Immunohistochemistry results showed that EHD1 protein was significantly increased in SCLC tissues compared with normal tissues.

PubTator disease annotations contain the mention in the text and the MeSH id corresponding to the detected disease. However, the authors of the original paper return the DOID of the disease in order to provide a better integration with BioXpress thus we need to perfom a mapping between MeSH identifiers and DOID. The Disease Ontology (DO) [49] provides a standardized ontology for human diseases, integrating disease and medical vocabularies through cross mapping of DO terms to MeSH, ICD, NCI's thesaurus, SNOMED and OMIM. We download the DO ontology file

[1]https://www.ncbi.nlm.nih.gov/research/pubtator/api.html

in json format from the DO GitHub repository[2] and for each DOID we extract the corresponding MesH ID (using property `"xref"`) and the name associated to the DOID (using property `"lbl"`). We then create two different dictionaries, one containing the mapping from MeSH id to DOID and one containing the names associated to each DOID. For example, the disease *"lung non-small cell carcinoma"* has DO identifier `"DOID:3908"` and MeSH id `"MESH:D002289"` thus we produce the following entries for the two dictionaries: `{"MESH:D002289":"DOID:3908"}` and `{"DOID:3908":"lung non-small cell carcinoma"}`. We made the two dictionaries available in our GitHub repository as json file, namely *"mesh_to_doid.json"* and *"doid_to_names.json"*. We use the mapping between MeSH ids and DOID when retrieving PubTator annotations while the second file is used when extracting the associated disease in the last module.

miRNA Detection

To detect miRNA mentions, the authors of the original paper developed a regex to retrieve such instances based on the miRBase convention [45]. However, they did not release the regex, therefore we implemented our own version. Following the miRBase convention [45], miRNAs are assigned sequential numerical identifier and they use 3 or 4 letter prefixes to designate the species, such as *"mir"* or *"miR"*. After checking our input data, we developed a regex that matches two cases for the miRNA mentions. In fact, some mentions contain the infix *"mir"* or *"miRNA"*, e.g. *"miR-325"* or *"miRNA-101"*, while others contain the infix *"microRNA"* or *"micro-RNA"*, e.g. *"microRNA 155"*. Therefore, the only difference between the two cases is that in one we matches infix `"(mi|Mi|MI)(RNA|R|r)"` (so that we match all lower and upper-case combinations) and in the other we match infix `"[M,m]icro[-]?RNA"`. The rest of the regex works in the same way for both cases. At the start of the mention we could find a sequence of 3 or 4 letters that indicate we are considering a human miRNA, e.g. *"hsa-miR-107"*. For this reason, the first part of the regex optionally matches the expression `"([a-z]{3,4}-)?"`. The second part of the expression matches the infix indicating it is a miRNA and, as we explained above, we have two different

[2] https://github.com/DiseaseOntology/HumanDiseaseOntology

forms we can match. Finally, we have the numerical identifier that can follow the previous infix with a dash or a space and the identifier could be followed by another letter or number, e.g. *"miR-101-3p"*, therefore the last part of the regex is matched by the expression `"s?(([0-9]|[a-z])+)|(-[^ ,.]*))"`. In particular, expression `"-[^ ,.]*"` matches a sequence of characters that follows the infix matched in the second part of the expression with a dash and sequence of characters that do not comprise spaces, commas or periods.

Lastly, we developed two boolean functions that identify if a phrase is of type *"Expression"* or *"disease-sample"*, that will be useful when filtering out irrelevant matches in the next module. To determine if a phrase, which is a span of text, is of one of the above-mentioned type we check if the tokens inside the span comprise some trigger words. About Type *"Expression"*, we directly check the token word and some triggers can be "expression", "level", etc. On the other hand, for Type *"disease-sample"* we check the lemma of each token so that we can consider different word forms. Some triggers for this type of phrases can be "tissue", "sample", "cell", "patient", etc.

6.2 Argument Filtering and Extraction

In this section, we describe the last module of the system, namely the *Argument Filtering and Extraction* phase. We can divide this step into two subtasks: first we filter matches returned by the RE module so that we keep only the one with relevant information and then we extract the GDAs from them. In particular, we extract the gene or miRNA mentioned, we normalize its expression level to "UP" or "DOWN" and finally we return the associated disease. This module takes as input the matches returned by the RE module and the output of the Entity Detection and Phrase Typing module. We recall that for each sentence we can have multiple matches and each match is returned by the RE module as a dictionary whose keys are the name of the components. In this module we process each match separately, however we check we do not insert duplicate information in the output results for the sentence. In particular, for each GDA detected we only store one copy referring to the same gene-expression level-disease triple.

Argument Filtering

For each match we check if the components we have found are of the correct type. In particular, we check the CA contains a gene or miRNA or is a phrase of type *"Expression"* while the CE1 must contain a disease or be a phrase of type *"disease-sample"*. If the checks succeed, we keep the components and start extracting relevant information otherwise we discard the match and check the others. If we are considering a TypeA sentence, we also need to check the CE2 contains a disease mention or is a phrase of type *"disease-sample"*. For each component, we perform a series of checks. If one succeeds we simply return True otherwise we continue with the next checks. If none of the checks succeed for one component, if we are considering the CA or the CE1, we raise a custom Exception called `"InvalidArgument"` and discard the whole match we were considering. In the case of the CE2 instead, if the check fails and we do not have any other matches we simply discard the CE2 and change the sentence type to TypeB.

About the CA, we firstly check if it contains a gene mention by checking the gene annotations for the abstract the sentence belongs to returned by the previous module or if it contains a miRNA mention by searching a match for the regex we developed also in the previous module. If we found none, we check if the CA is a phrase of type *"Expression"* by using the boolean function described in Section 6.1. We can further check for gene mentions using the *general_annotation* dictionary we described in Section 6.1. Finally, we perform a lower-case normalization and perform the same checks again, to avoid missing mentions due to different cases of the letters in the words. As we said before, if none of the checks succeed, we stop.

> Example 6.1: Here, we show that miR-136 is significantly upregulated in human NSCLC primary tumors and cell lines compared to their nontumor counterparts.

Consider the sentence in Example 6.1, the RE module returns two matches for this sentence which differ only in the CA. In fact, one match identify token *"we"* as CA while the other correctly identify the word *"miR-136"* as CA. Following the reasoning we described above, the check fails for the match with *"we"* thus we discard such

match and we only proceed with the one containing the correct CA.

We perform the same checks both for CE1 and CE2. In particular, we firstly check if the component contains a disease mention by searching among the one returned for the specific abstract the sentence belongs to. Then, we check if the phrase corresponding to either CE1 or CE2 is of type *"disease-sample"* and finally we can check for disease mentions inside the components by searching among the *general_annotations* dictionary.

Extraction

Once we checked the components are of the correct type, we extract relevant information from each match. In particular, we extract the gene/miRNA mentioned in the sentence, its expression level normalized to "UP" or "DOWN" and the associated disease. We also set a flag comparison which is useful when we search for results that can extend BioXpress database. We described the meaning of this flag in Section 3.1.

We extract the gene or miRNA mentioned in the sentence from the CA by means of a function that returns a list of the genes or miRNAs mentioned in the CA. Each entry of the list is a list as well containing the gene or miRNA mention and its identifier. In particular, for genes we return as identifier the one we retrieved from PubTator while for miRNAs we simply return a string *"micro-RNA"* to distinguish them from genes. We allow for multiple genes or miRNAs because we may encounter sentences where we have data about multiple genes or miRNAs expressions, as we can see in Example 6.2. In this case, the function will return "VEGF", "BMP-2" and "BMP-4" thus when we output the result of the full pipeline we write one row per gene mention.

> Example 6.2: Our study has shown that the expressions of the VEGF, BMP-2 and BMP-4 mRNAs were significantly higher (7.1-fold, 25.6-fold and 2.3-fold, respectively) in lung cancer samples than in adjacent normal lung tissues (real-time RT-PCR).

As we did for filtering, we perform a series of operation to extract the gene or miRNA mentions. When one of this operation returns some results, we stop and return the mentions otherwise we proceed with the next search. If we found no gene or miRNA

mention, we raise a custom Exception called `"GeneNotFound"`. Firstly, we check if the CA comprises any of the gene mentions returned by PubTator for the abstract the sentence belongs to or any miRNA mentions. If we found none, we check for the token that precedes the CA since in some cases the gene expressed is placed just before the CA we detected but has no dependency connection to it. Then, we can also check for gene mentions in the *general_annotations* dictionary or perform some lower-case normalization or term expansion to mitigate some issues we encounter with mentions. For example, we may refer to the same gene with different names that could be abbreviations or different terms for special characters, such as β (which could be referred to as *"beta"*) or α (which could be referred to as *"alpha"*).

> Example 6.3: In the present study, we confirmed that Tβ4 expression was increased in NSCLC tissues and cell lines.

Consider Example 6.3, here the expressed gene is *"Tβ4"* but in the mentions returned by PubTator such gene is stored as *"Tbeta4"* thus we find no information about the expressed gene at first. To solve this issue, we expand term *"β"* to *"beta"* and check again the mentions so that we can return the expressed gene correctly.

Information about the expression level of genes and miRNAs is stored in the SI component thus we normalize such token to either "UP" or "DOWN". This operation is performed by checking a mapping between possible SIs and their normalized level. For example, terms "upregulated", "increased" or "positive" refer to level "UP" while terms like "reduced", "silenced" or "low" refer to "DOWN". There are some cases in which the SI is not in dictionary because it may have a neutral connotation and not refer to a specific level.

> Example 6.4: HOTAIR was highly expressed both in NSCLC samples and cell lines compared with corresponding normal counterparts.

Consider Example 6.4, the SI is identified in *"expressed"*, which is neutral. In this case we try to expand the term by checking any adverb depending on it. In fact, in the example we find adverb *"highly"* thus we can return expression level "UP". If we find no adverb depending on the SI and the SI is *"expressed"*, we assume it refers to level

"UP". Additionally, we can also check if any token corresponding to an expression level is connected to the other components and return its normalized level. If even this expansion fails, we raise a custom Exception called `"MistypedExpressionLevel"`.

DEXTER was primarly developed to extend BioXpress database, which contains GDAs referring to cancer where the comparison is between cancer tissues and some control samples. For this reason, in the original system authors set a comparison flag which identifies if we have an implicit comparison for TypeB sentences and we check if there is a comparison with a control sample in TypeA sentences. Since this is a reproducibility study, we developed a function to assess this flag as well, however in the files containing the results of the original system there is no information about this flag therefore we do not output it. We perform two different operations based on the sentence type. If we are dealing with a TypeA sentence, we check if either the CE1 or CE2 contains any "Control Triggers", which could be "healthy", "control", etc. If we find any control trigger words in the components, as in Example 6.5.(a), we set the flag to *"Control"*, otherwise we set it to *"Not-Control"* as in Example 6.5.(b).

> Example 6.5:
> (a). JAK1 expression was higher in NSCLC tissues than in normal lung tissues. (flag: *"Control"*)
> (b). The expression of STIM1 was significantly higher in carcinoma tissue than in the adjacent non-neoplastic lung tissue. (flag: *"Not-Control"*)

About TypeB sentences, we check if the SI refers to any "Control Implicit Triggers", which are terms that imply a comparison with another sample. Such words could be "increased", "reduced", etc. This is the case of sentence in Example 6.6. and here we set the flag to *"Control_Implicit"*. On the contrary, if the SI is not a control implicit trigger we set the flag to *"none"*.

> Example 6.6: MiR-145 is downregulated in several human malignancies, including lung cancer, but the responsible molecular mechanisms remain unclear. (flag: *"Control_Implicit"*)

Inferring the disease

The only thing left to extract is the associated disease, which requires some extra care. We developed a function to extract the associated disease that returns the DOID of the disease, the corresponding DO name (which is stored in the dictionary *doid_to_names* available in the homonymous json file), the disease mention in the text and the location where we found it. In fact, we can either extract the disease directly from the CEs (location: *"Sentence_ARG"*) or we may need to infer it from the title (location: *"Title"*) or from context (location: *"First Sentence/Sentence"*).

Firstly, we check if the CE1 or CE2 contains any disease mention by searching among the ones returned by PubTator for the abstract the sentence belongs to. If we found any, we check if it is not a generic disease such as "tumor", "cancer", etc. In fact, consider sentence in Example 6.7, here we found the associated disease to be *"tumor"* (from CE1) yet it is a generic disease and we do not return it. If we find a non-generic disease, we return it and the location is set to *"Sentence_ARG"* otherwise we further check if the CEs contain any disease mentions from the ones in the *general_annotations* dictionary.

> Example 6.7: MiR-204 expression was decreased in tumor samples compared with non-cancerous tissue-derived controls. (PMID: 25157435)

If we found no disease mentions in the CEs or we only found generic disease, we can infer it from the abstract. We recall that in the Entity Detection and Phrase Typing module (Section 6.1), for each PMID we also stored the disease mentioned in the title, the title text and the whole abstract text. Firstly, we check the diseases mentioned in the title and if any of them does not refer to a generic disease we return it and set the location to *"Title"*. In fact, any disease mentioned in the title is likely to be the disease associated to the sentence. Back to Example 6.7, the title of the article is *"miR-204 functions as a tumor suppressor by regulating SIX1 in NSCLC."* thus we find the disease mention *"NSCLC"* in the title and we return it as the associated disease for the sentence.

If we find no disease in the title, we infer it from the abstract by parsing the full text with the SpaCy pipeline and by checking each sentence separately using

the iterator `"doc.sents"`. We firstly check if there is any disease mentioned in the first sentence with the same techniques employed for the CEs and if so we return it and set the location to *"First Sentence"*. If we find no disease in the first sentence, we look for sentences that may discuss the investigational aims of the paper or describe the experimental set-up. In the first case, we check the sentence contains any "investigation triggers" that can be "investigate", "demonstrate", etc and then we check the subject of the sentence is either "we", "authors", "results" or similar terms. Then we extract the disease mention from such sentences.

> Example 6.8: MiR-21 expression was higher in cancer tissues than in normal tissues. (PMID: 22323912)

Consider the sentence in Example 6.8, we found no disease either in the CEs or in the title but in the abstract we found the following sentence *"The aim of this study was to investigate the significance of miR-21 expression level in relation with clinicopathological factors and prognosis in breast cancer."*. As we can see, we have trigger "investigate" and the subject is "aim" thus we extract the associated disease to be *"breast cancer"* (DOID: 1612) and set the location to *"Sentence"*. Similarly, in the second case we check if there is a sentence that contains any "analyzed triggers" like "tested", "analyzed", etc. If so, we check for disease mentions in such sentence and return it as the associated disease together with the location set to *"Sentence"*. If we cannot find any disease after inference, we return the generic disease we detected earlier if we found any. Otherwise, we raise a custom Exception called `"DiseaseNotFound"`.

At the end of the pipeline we output results as a csv file where each row is a GDA. We recall we can have multiple matches for each sentence and we process each match separately hence before writing each row we check we do not already have information about the same gene, expression level and associated disease for the considered sentence. We may still have multiple rows for the same sentence in the case we extract more than one expressed gene. Each output row comprises the following information:

- `"PMID"`: PMID of the abstract the input sentence belongs to.
- `"geneMen"`: expressed gene

- `"geneID"`: identifier of the gene, which is the gene ID for genes and 'NA' for microRNAs.
- `"DOID"`: DO identifier of the associated disease
- `"DOID_Name"`: DO name of the associated disease
- `"DiseaseMention"`: mention in the text of the associated disease
- `"DiseaseDetectedFrom"`: location of the disease mention, which can be either *"Sentence_ARG"*, *"Title"* or *"First Sentence/Sentence"*.
- `"ExpressionLevel"`: gene/miRNA expression level normalized to *"UP"* or *"DOWN"*.
- `"SentenceType"`: sentence type, which can be either *"TypeA"* or *"TypeB"*.
- `"Sample1"`: text of the CE1.
- `"Sample2"`: text of the CE2.
- `"Sentence"`: text of the input sentence.

Chapter 7

Results and Discussion

In this chapter we evaluate our implementation of DEXTER by using data provided by the authors in BioXpress[1] as ground truth to assess the accuracy of our system. In particular, we are interested in how many sentences we can parse with our system and the correctness of the results. The chapter is structured as follows: in Section 7.1 we provide a brief overview of the input data and give some statistics of the number of entries and unique sentences. Then we also describe the evaluation notebook we used to assess our system's performance. Finally, in Section 7.2 we provide some statistics of the output of our system and discuss the results in terms of percentage of parsed sentences and correctness of the results. We also perform an error analysis where we compare the dependency parser we are using and the one used in the original system and we also investigate the impact of PubTator in our system's performance.

7.1 Materials and Methods

As we explained in Section 3.3, the original system was evaluated in two different ways. One evaluation focused on results relevant to BioXpress while the other tested the general ability of DEXTER of extracting GDAs. The motivation behind our work is to reproduce DEXTER thus our evaluation focuses on obtaining the same results as the original system described in [9]. For this reason, we focus on testing our implementation on data provided by the authors of the original paper available in

[1]https://hive.biochemistry.gwu.edu/bioxpress

Table 7.1
Input data statistics. In the second and third columns we provide the number of entries in the file based on the sentence type. In the forth and fifth columns we provide the number of unique sentences in the file based on the sentence type. In fact, we may have multiple entries for the same sentence.

Use Case	# of input entries		# of unique sentences	
	TypeA	TypeB	TypeA	TypeB
lung cancer	985	2019	788	1574
GT genes	106	236	97	201
microRNAs	1842	4448	1468	3251

BioXpress. Since the input data comprises results obtained from runs of DEXTER original system, ideally we would like to parse all sentences and extract the same GDAs.

Test Data

The sets of sentences we use as test data comprises the results of the original system for three use cases, filtered according to BioXpress guidelines. In particular, we recall that DEXTER output is appropriate for BioXpress if the associated disease is cancer and there is a comparison between some cancer tissues and an healthy sample [9]. These three use cases comprise abstracts related to lung cancer, a group of genes called glycosyltransferases (GTs) and microRNA's role in cancer. We provide some data statistics for all three use cases in Table 7.1. For the first file, data focuses on the relation between some genes and different types of lung cancer. As we can see from Table 7.1, we were provided with 3088 entries, in particular 1059 entries comprise TypeA information while the other 2029 extract GDAs from TypeB sentences. We also provide some statistics on the number of unique sentences in the file, since we may extract multiple entries from the same sentence. For lung cancer we have 808 sentences identified as TypeA and 1480 detected as TypeB. The second use case focus on GTs, which is a family of genes which influence protein functions thus their dysfunction or deregulation may lead to disease, including some types of cancer [9]. This set is smaller than the previous one and comprises 277 unique sentences, respectively 99 TypeA and 178 TypeB, for a total of 408 entries, divided as 130 TypeA and 278 TypeB. For the last dataset, the authors of the original paper focused on the

role of microRNAs in cancer. In fact, in cancer cells miRNAs have been found to be heavily dysregulated thus it is crucial to investigate how miRNAs regulate the development of human tumors by acting as tumor suppressors or oncogenes [50]. We have 4719 sentences regarding microRNAs, respectively 1468 TypeA and 3251 TypeB, for a total of 6290 entries divided as 1842 TypeA entries and 4448 of typeB. Data for the use cases is provided as three csv files in BioXpress website[2] thus we downloaded the files and run our system on them both separately, producing three different output files, and then we merged all three input files and provide a single output run to assess the quality of our implementation in terms of accuracy, precision and recall.

Evaluation Notebook

We assess our implementation performance by means of a Jupyter Notebook [51], where we compare our output with the test data we previously described. We briefly explain how the comparison was performed for all three use cases in a single run. We also computed the same metrics for each use case separately to see if the system performed much better in some cases. Firstly, we store in memory both the test data (i.e, our ground truth) and the output run by loading the csv files as a pandas DataFrame object[3]. We check the number of entries for both the ground truth and our output based on the sentence type and then we compare the number of unique sentences between the two files. In particular, we count how many unique sentences were correctly parsed by our system, i.e. those that did not raise any exception, and how many unique sentences there were in the test data. The ratio between these two values gives us the percentage of parsed sentences, which ideally would be 100%. Then, we assess the correctness of our output data by comparing our entries with the one in the test data. To do so, we create a dictionary based on the ground truth file where for each PMID we store a dictionary for each sentence belonging to the same abstract. In each sentence dictionary, we store information about the genes mentioned in the sentence (i.e, column *"geneMen"*), their expression level (i.e,

[2]https://hive.biochemistry.gwu.edu/bioxpress
[3]https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

Table 7.2
Data statistics referring to the output of our implementation of DEXTER pipeline, when we run each use case separately. In the second and third columns we provide the number of entries in our system output based on the sentence type. In the forth and fifth columns we provide the number of processed sentences in the file based on the sentence type. In fact, we may have multiple entries for the same sentence.

Use Case	# of resulting entries		# of processed sentences	
	TypeA	TypeB	TypeA	TypeB
lung cancer	1059	2029	808	1480
GT genes	130	278	99	178
microRNAs	1974	4151	1479	3111

column *"ExpressionLevel"*) and the sentence type (i.e, column *"SentenceType"*). We assess the correctness of our output both with respect to the expression level and sentence type. In both cases, all sentences that are present in the ground truth but not in our output are considered wrong matches. About the expression level, we consider correct all output entries were we have the same expression level and gene mention of the ground truth. Therefore, we not only check if we extract the same expression level but we also verify we are considering the same gene. On the other hand, when assessing the correctness with respect to the sentence type we only check we identify the sentences with the same type as in the ground truth. We assess the system performance by computing the accuracy, precision, recall and F1 score with respect to the expression level and sentence type. In order to compute such metrics, we use the `sklearn.metrics` module[4] and use the weighted average when computing precision, recall and F1 score.

7.2 Results

In this section we describe our implementation performance. As we briefly mentioned before, we run our system on the three use cases separately and provide one output file per use case but we also perform one single run for all three use cases merged together in a single file. In Table 7.2 we provide some statistics on the output data for each use case. Overall, if we compare the numbers with the one in Table 7.1 we notice

[4]https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics

that in general we are identifying more TypeA sentences than the one in the input files and fewer TypeB. This result can have two explanations: there may be some mistyped sentences in the input files that we are correctly classifying or we wrongly detect some CE2s that make our system classify such entries as TypeA while they are indeed TypeB. We manually checked such sentences and we found out there are some entries that are wrongly identified as TypeB in the input data. In fact, TypeA sentences can be viewed as TypeB if we do not consider the sample of comparison. Consider sentence in Example 7.1, we can clearly identify it as TypeA however the original system classifies it as TypeB and extract the GDA without considering the comparison sample.

> Example 7.1: The expression of SALL4 was significantly increased in SCLC than in para-carcinoma tissues (P < 0.01).

In this case, the original system wrongly considers both *"SCLC"* and *"para-carcinoma tissues"* as CE1 and outputs two rows with the same GDA where the only difference lies in the CE1. On the contrary, our system correctly identifies *"para-carcinoma tissues"* as CE2 and correctly classify the sentence as TypeA thus it outputs one single row. Overall, we have 88 TypeA entries that are wrongly classified as TypeB by our system and 335 TypeB sentences wrongly detected as TypeA. We manually checked entries wrongly classified as TypeA and discovered that most of them are actually typeA sentences that were misclassified in the input, thus we are extracting more information than the original system. However, since this is a reproducibility study we will consider such sentences as wrong matches because we are assuming our ground truth is correct. Another thing we can notice from Table 7.2 is that we have a greater number of entries than the input data. This may be due to some differences in the RE module between our system and the original one. In fact, we have no clue on how the authors of the original system apply the rules to the sentences hence we may be applying more rules than them and extract more matches that may contain the same information but are marked as different. Overall, what concerns us is to find correct matches therefore we are not bothered by the higher number of entries as long as we achieve high accuracy.

We recall we evaluate our system performance in terms of the percentage of parsed sentences and the accuracy of the results. Overall, our implementation correctly parse 97% of the sentences, these are sentences for which we are able to extract GDAs and that do not raise any exceptions during computation. Ideally, our system should parse 100% of the sentences since our input data is actually the output of the original system. For this reason, we perform an error analysis to understand why we are missing some sentences.

Dependency Parsing

After a deeper analysis, we discovered that most missed sentences were due to parsing error that translated into unmatched sentences in the RE module. We recall the original system employs the Stanford CoreNLP toolkit [10] while we are using the SpaCy package thus we perform a comparison between the dependency parsers available in the two libraries. In fact, we noticed several parse trees computed by SpaCy were different from the one returned by the CoreNLP library and in some cases the resulting dependency tree from SpaCy library failed to detect some dependencies that are correctly displayed by the CoreNLP parsing. We provide a brief introduction to the Dependency Parsing task in general and perform a comparison between the dependency parsing in SpaCy and CoreNLP.

We recall we defined the Dependency Parsing task at the beginning of Chapter 5 since DEXTER is based on matching patterns in the dependency tree of each sentence. We saw that Dependency Parsing produces a dependency tree, that is a directed graph where the vertices are the tokens of the sentence and arcs display grammatical relationships among them. There are many approaches used to solve the Dependency Parsing task which comprise transition-based methods, graph-based models and neural architectures. Transition-based dependency parsing is based on two main data structures: a stack, where the parse is built, and a buffer, which comprises tokens to be parsed. The parser is non deterministic because there is no unique action to take thus an oracle is used to make decisions on which transition to apply. Oracles are usually produced using supervised learning methods. The basic model for transition-based dependency parsing is arc-standard, which allows for three

transition operations. `SHIFT` removes the first buffer element and pushes it into the stack. This transition is applied when no other action is applicable. `LEFTARC` pops the second top-most stack element and attaches it as a dependent to the top-most element. This operation is applicable it there are at least two elements in the stack and the second element is not the root node. `RIGHTARC` pops the top-most element in the stack and attaches it as a dependant to the second top-most element. This transition is applicable when there are at least two elements in the stack and all of the dependants of the top-most element have already been assigned. There are other transition-based models like attardi parser, beam search and arc eager. We focus on arc eager since SpaCy Dependency Parser is based on it. Arc Eager allows for words to be attached to their heads as soon as possible by making some small changing in the previous operations and adding a new transition called `REDUCE`. `SHIFT` operation is the same as in arc standard. `LEFTARC` pops the top-most stack element and attaches it as a dependent to the first buffer element. `RIGHTARC` attaches the first buffer element as a dependant of the top-most stack element and shifts the first buffer element into the stack. Finally, `REDUCE` pops the stack. Given the dependency tree produced as output by the system and a corresponding reference parse, the performance of a dependency parser is evaluated following two scores, namely Unlabelled Attachment Score (UAS) and Labelled Attachment Score (LAS). UAS is the percentage of words in the input that are attached to the correct head, ignoring the dependency label. On the other hand, LAS is the percentage of words in the input that are attached to the correct head with the correct dependency label [7].

While SpaCy Dependency Parsing is based on arc eager and uses a dynamic oracle trained using an online learning strategy [13], Stanford Dependency Parsing is based on arc standard and uses a different neural architecture to train the oracle [52]. More precisely, the SpaCy dependency parser described in [13] rely on a previous work, where the authors developed a dependency parser modifying the Arc Eager transition system to allow for non-monotonic transitions [53]. We report the UAS and LAS scores of the two systems in Table 7.3. Results are taken from [13] and were computed using the OntoNotes corpus converted into dependencies using the ClearNLP 3.1 converter with the train/dev/test split of CoNLL 2012 shared task [13].

Table 7.3
SpaCy Dependency parser and Stanford CoreNLP Dependency Parser scores. We report the scores from [13], which were computed using the OntoNotes corpus converted into dependencies using the ClearNLP 3.1 converter with the train/dev/test split of CoNLL 2012 shared task.

Tansition System	UAS	LAS
SpaCy [13]	91.85	89.91
Stanford CoreNLP [52]	89.59	87.63

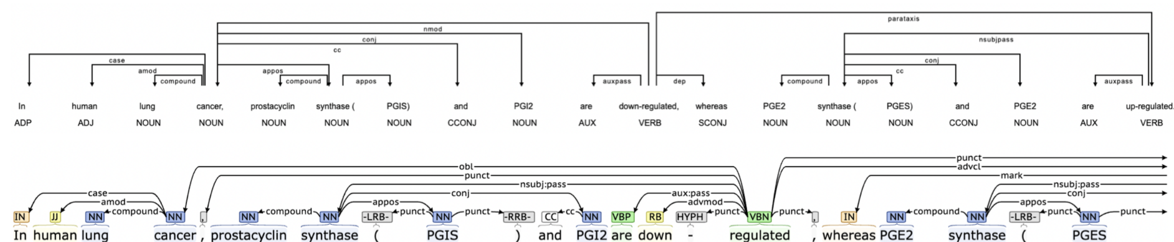


Figure 7.1: We display the dependency tree of the same sentence both using the Spacy dependency parser (figure above) and the Stanford CoreNLP one (figure below). The SpaCy tree is displayed using the `DisplaCy` package while we computed the dependency tree using the tool https://corenlp.run/. We only display the part of interest of the dependency tree computed by the Stanford library due to space issues.

From this table, it looks like SpaCy dependency parser outperforms the Stanford one by 0.97% both on UAS and LAS, however we discovered that in our datasets there are some cases in which the Stanford dependency parser produces the correct tree while the SpaCy parser fails to detect some dependencies. For example, we discovered that if we place a part of the sentence that is not the subject at the beginning of the sentence, SpaCy dependency parser struggles to detect the correct subject. Consider sentence *"In human lung cancer, prostacyclin synthase (PGIS) and PGI2 are down-regulated, whereas PGE2 synthase (PGES) and PGE2 are up-regulated."*, the dependencies for both libraires are displayed in Figure 7.1. Here we can see that the SpaCy dependency parser fails to detect the subject for verb *"down-regulated"* thus our system will raise an Exception and fails to extract the GDA for such sentence. On the contrary, Stanford CoreNLP is correctly identifying the subject of the sentence in token *"synthase"*. For this reason, we can assess that the performance drop due to unparsed sentences is negligible (since we are still able to parse 97% of them) and mostly due to the use of SpaCy library.

Table 7.4
Performance metrics. Results refer to all three use-cases and we consider the data provided in BioXpress as ground truth. The first two columns show the performance metrics when we run our implementation using the full DEXTER pipeline, i.e. using the PubTator annotations. The other two columns report the system performance when using the annotations provided by the input data instead of the PubTator annotations. Metrics are computed in terms of correct gene expression level and correct sentence type, using metrics from `sklearn` with weighted average.

	Full Pipeline		Without PubTator Annotations	
	Expression Level	Sentence Type	Expression Level	Sentence Type
Accuracy	0.8444	0.9356	0.9293	0.9326
Precision	0.8445	0.9379	0.9296	0.9350
Recall	0.8444	0.9356	0.9293	0.9326
F1 score	0.8445	0.9361	0.9294	0.9332

Accuracy Evaluation

Our system performance is shown in Table 7.4. Results refer to all three use-cases and we consider the data provided in BioXpress as ground truth. The first two columns show the performance metrics when we run our implementation using the full DEXTER pipeline, i.e. using the PubTator annotations. The other two columns report the system performance when using the annotations provided by the input data instead of the PubTator annotations. As already mentioned, metrics are computed in terms of correct gene expression level and correct sentence type. *"Full Pipeline"* refers to the run of all three use cases where we apply the DEXTER pipeline as described in [9] thus for detecting gene and disease mentions we are using PubTator API[5]. As shown in Table 7.4, our system achieves an accuracy of 84% on correct expression level and of 93% on correct sentence type. Results for Precision, Recall and F1 Score are almost identical because we are using the weighted average option for computing the metrics. We notice that the accuracy with respect to the expression level is 10% lower than the one considering the sentence type. We recall we consider an entry correct with respect to the expression level if both the expression level and the extracted gene are the same as in the ground truth. For this reason, to check if the performance drop depends on our system implementation or it has to do with the mentions detection we run our system without using PubTator for extracting

[5]https://www.ncbi.nlm.nih.gov/research/pubtator/api.html

Table 7.5
Performance metrics. Results refer to each use case run separately. The first column reports the percentage of parsed sentences with respect to the ground truth, which ideally would be 1. Accuracy is computed in terms of correct gene expression level and correct sentence type, using metrics from `sklearn`.

Use Case	Parsed Sentences	Accuracy	
		Expression Level	Sentence Type
lung cancer	0.9687	0.8955	0.9373
GT genes	0.9295	0.8713	0.8758
microRNAs	0.9727	0.8176	0.9390

the mentions but we directly use the mentions detected in the ground truth. The results of such run are reported in the *"Without PubTator Annotations"* columns in Table 7.4. As expected the system performance in terms of correct sentence type is unchanged, however we can notice the performance in terms of expression level improves by almost 10% and achieves an accuracy of almost 93%. This means that most of the wrongly detected levels are due to the different version of PubTator and do not depend on the system implementation itself. In fact, PubTator is an automated tool based on a neural model that is periodically retrained therefore the version of PubTator we use to retrieve mentions is different from the one used by the authors of the original paper [9].

Finally, we computed the same performance metrics for each use case separately, so that we can assess if our system works well for all use cases or performance differs based on the dataset. We report the percentage of parsed sentences and the accuracy based on correct sentence type and expression level for each use case in Table 7.5. We notice that the percentage of parsed sentences is above 92% for all three use cases. In particular, for the first and last use case, such value is almost 97% while it drops to 93% for the second dataset. This could be due to the fact that the second use case has fewer entries that the other two hence a missed sentence plays a more important role in the computation of the metric. The same holds for results about the accuracy with respect to the sentence type. About the correctness based on the expression level, all use cases reaches almost the same results. In particular, the first two use cases have an accuracy of 87/89% while the last one reaches only 81% accuracy. These

results refers to full runs where we used the PubTator API to detect mentions and after a deeper analysis we discovered that most mention errors were related to the microRNA use case. Overall, the difference in performance between the use cases is negligible and Table 7.5 confirms our system works well in all three datasets.

Chapter 8

Conclusion

This work reproduces DEXTER, a system to automatically extract GDAs from biomedical abstracts [9]. The goal is to provide a reliable baseline for future works regarding RE. We implemented DEXTER as an end-to-end application that takes as input biomedical abstracts and returns relevant information about the expressed gene/microRNA and the associated disease. We preserved the original block architecture and we made some changes in each block to enable a seamless integration of the different modules. The system is entirely developed in Python thus we used the SpaCy library [13], which provides annotated text using some pre-trained models available on their website[1]. We also added a custom component to the SpaCy standard pipeline to expand entity mentions using hand-craft rules which will be useful when extracting information from the matched components. We evaluated the effectiveness of the reproduced version of DEXTER by running our implementation on the sentences of the three use-cases available in BioXpress and compared the results in terms of correct sentence type and gene expression level. Overall, our system was able to parse 97% of the input sentences while discarded sentences are mostly due to missing PubTator annotations or problems related to the Dependency Parsing. In fact, after a deeper analysis we noticed several parse trees computed by SpaCy were different from the one returned by the CoreNLP library therefore most unparsed sentences can be attributed to it. Nevertheless, the benefits of having an end-to-end system completely written in Python outweigh the liabilities therefore we decide to maintain the SpaCy

[1]https://spacy.io/usage/models

library. The system achieved an accuracy of 84% on the gene expression level. Such value raises up to 93% if input mentions are used instead of PubTator annotations. In fact, PubTator is an automated tool based on a neural model that is periodically retrained therefore the version of PubTator we use to retrieve mentions is different from the one used in the original paper. As a consequence, missing mentions affect negatively the accuracy performance of the system. Nevertheless, results in Table 7.4 demonstrate that the system is reproducible and it can be used as a benchmark for future works. However, implementing this system highlighted its limits. DEXTER is a rule-based model hence information are extracted based on patterns written to match sentences with a very specific syntactic structure, namely TypeA and TypeB sentences. The main drawback of using patterns to extract relations lies in the fact that language is incredibly varied hence there are a lot of ways of conveying the same message. This translates into requiring a huge number of patterns for each relation, that is laborious, since it requires the cooperation between domain experts and linguists, and time-consuming. The implemented version of DEXTER is publicly available in GitHub. In this way researchers can compare newly developed systems with a state-of-the-art instead of merely rely on annotated data. An extended abstract of this work has been presented in the 10th edition of the PhD Symposium on Future Directions in Information Access (FDIA) held in Lisbon, Portugal in July 2022. The proceedings of such event will be published at CEUR-WS.org.

Bibliography

[1] M. Jackson, L. Marks, G. H. W. May, J. B. Wilson, The genetic basis of disease., Essays in biochemistry 62 (2018) 643–723. doi:10.1042/EBC20170053.

[2] J. Piñero, J. Ramírez-Anguita, J. Saüch-Pitarch, F. Ronzano, E. Centeno, F. Sanz, L. I. Furlong, The disgenet knowledge platform for disease genomics: 2019 update, Nucleic acids research 48 (2019). doi:10.1093/nar/gkz1021.

[3] Q. Jiang, Y. Wang, Y. Hao, L. Juan, M. Teng, X. Zhang, M. Li, G. Wang, Y. Liu, mir2disease: a manually curated database for microrna deregulation in human disease, Nucleic acids research 37 (2008) D98–104. doi:10.1093/nar/gkn714.

[4] H. Dingerdissen, J. Torcivia-Rodriguez, T.-C. Chang, R. Mazumder, R. Kahsay, Biomuta and bioxpress: Mutation and expression knowledgebases for cancer biomarker discovery, Nucleic Acids Research 46 (2018) D1128–D1136. doi:10.1093/nar/gkx907.

[5] Q. Wan, H. Dingerdissen, Y. Fan, N. Gulzar, Y. Pan, T.-J. Wu, C. Yan, H. Zhang, R. Mazumder, Bioxpress: An integrated rna-seq-derived gene expression database for pan-cancer analysis, Database : the journal of biological databases and curation 2015 (2015). doi:10.1093/database/bav019.

[6] S. Marchesin, G. Silvello, TBGA: a large-scale gene-disease association dataset for biomedical relation extraction, BMC Bioinformatics 23 (2022). doi:10.1186/s12859-022-04646-6.

[7] D. Jurafsky, J. H. Martin, Speech and language processing, 2022. 3rd ed. draft available at: https://web.stanford.edu/~jurafsky/slp3/.

[8] H. Wang, K. Qin, R. Y. Zakari, G. Lu, J. Yin, Deep neural network-based relation extraction: an overview, Neural Computing and Applications 34 (2022). doi:10.1007/s00521-021-06667-3.

[9] S. Gupta, H. Dingerdissen, K. E. Ross, Y. Hu, C. H. Wu, R. Mazumder, K. Vijay-Shanker, DEXTER: Disease-Expression Relation Extraction from Text, Database 2018 (2018). doi:10.1093/database/bay045.

[10] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, D. McClosky, The Stanford CoreNLP natural language processing toolkit, in: Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, Association for Computational Linguistics, Baltimore, Maryland, 2014, pp. 55–60. URL: https://aclanthology.org/P14-5010. doi:10.3115/v1/P14-5010.

[11] M.-C. de Marneffe, T. Dozat, N. Silveira, K. Haverinen, F. Ginter, J. Nivre, C. D. Manning, Universal Stanford dependencies: A cross-linguistic typology, in: Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14), European Language Resources Association (ELRA), Reykjavik, Iceland, 2014, pp. 4585–4592. URL: http://www.lrec-conf.org/proceedings/lrec2014/pdf/1062_Paper.pdf.

[12] C.-H. Wei, A. Allot, R. Leaman, Z. Lu, PubTator central: automated concept annotation for biomedical full text articles, Nucleic Acids Research 47 (2019) W587–W593. URL: https://doi.org/10.1093/nar/gkz389. doi:10.1093/nar/gkz389.

[13] M. Honnibal, M. Johnson, An improved non-monotonic transition system for dependency parsing, in: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics, Lisbon, Portugal, 2015, pp. 1373–1378. URL: https://aclweb.org/anthology/D/D15/D15-1162.

[14] M. Honnibal, Introducing spacy, https://explosion.ai/blog/introducing-spacy, 2015.

[15] J. Li, A. Sun, J. Han, C. Li, A survey on deep learning for named entity recognition, IEEE Transactions on Knowledge and Data Engineering 34 (2022) 50–70. doi:10.1109/TKDE.2020.2981314.

[16] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (2015) 436–444. URL: https://doi.org/10.1038/nature14539. doi:10.1038/nature14539.

[17] S. Zhang, N. Elhadad, Unsupervised biomedical named entity recognition:

Experiments with clinical and biological texts, Journal of Biomedical Informatics 46 (2013) 1088–1098. URL: https://www.sciencedirect.com/science/article/pii/S1532046413001196. doi:https://doi.org/10.1016/j.jbi.2013.08.004, special Section: Social Media Environments.

[18] W. Shen, J. Wang, J. Han, Entity linking with a knowledge base: Issues, techniques, and solutions, IEEE Transactions on Knowledge and Data Engineering 27 (2015) 443–460. doi:10.1109/TKDE.2014.2327028.

[19] S. Brin, Extracting patterns and relations from the world wide web, in: The World Wide Web and Databases, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 172–183.

[20] E. Agichtein, L. Gravano, Snowball: Extracting relations from large plain-text collections, in: Proceedings of the Fifth ACM Conference on Digital Libraries, Association for Computing Machinery, New York, NY, USA, 2000, pp. 85–94. doi:10.1145/336597.336644.

[21] O. Etzioni, M. Cafarella, D. Downey, S. Kok, A.-M. Popescu, T. Shaked, S. Soderland, D. S. Weld, A. Yates, Web-scale information extraction in knowitall: (preliminary results), in: Proceedings of the 13th International Conference on World Wide Web, Association for Computing Machinery, New York, NY, USA, 2004, pp. 100–110. doi:10.1145/988672.988687.

[22] A. Yates, M. Banko, M. Broadhead, M. Cafarella, O. Etzioni, S. Soderland, TextRunner: Open information extraction on the web, in: Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT), Association for Computational Linguistics, Rochester, New York, USA, 2007, pp. 25–26.

[23] T. Hasegawa, S. Sekine, R. Grishman, Discovering relations among named entities from large corpora, in: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04), Barcelona, Spain, 2004, pp. 415–422. doi:10.3115/1218955.1219008.

[24] A. Fader, S. Soderland, O. Etzioni, Identifying relations for open information extraction, in: Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics,

Edinburgh, Scotland, UK., 2011, pp. 1535–1545.

[25] N. Bach, S. Badaskar, A review of relation extraction (2007). URL: https://www.cs.cmu.edu/~nbach/papers/A-survey-on-Relation-Extraction.pdf.

[26] C. Liu, W. Sun, W. Chao, W. Che, Convolution neural network for relation extraction, in: Advanced Data Mining and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 231–242.

[27] P. Bose, S. Srinivasan, W. C. Sleeman, J. Palta, R. Kapoor, P. Ghosh, A survey on recent named entity recognition and relationship extraction techniques on clinical texts, Applied Sciences 11 (2021). doi:10.3390/app11188319.

[28] I. Segura-Bedmar, P. Martínez, C. de Pablo-Sánchez, A linguistic rule-based approach to extract drug-drug interactions from pharmacological documents, BMC Bioinformatics (2011). doi:10.1186/1471-2105-12-S2-S1.

[29] Q. Li, S. A. Spooner, M. Kaiser, N. Lingren, J. Robbins, T. Lingren, H. Tang, I. Solti, Y. Ni, An end-to-end hybrid algorithm for automated medication discrepancy detection, BMC Medical Informatics and Decision Making 15 (2015). doi:10.1186/s12911-015-0160-8.

[30] D. Mahendran, B. T. McInnes, Extracting adverse drug events from clinical notes, 2021. URL: arXiv:2104.10791.

[31] K. Swampillai, M. Stevenson, Extracting relations within and across sentences, in: International Conference Recent Advances in Natural Language Processing, RANLP, 2011, pp. 25–32.

[32] S. K. Sahu, A. Anand, K. Oruganty, M. Gattu, Relation extraction from clinical texts using domain invariant convolutional neural network, in: Proceedings of the 15th Workshop on Biomedical Natural Language Processing, 2016, pp. 206–215.

[33] M. Kim, S. H. Baek, M. Song, Relation extraction for biological pathway construction using node2vec, BMC Bioinformatics 19 (2018). doi:10.1186/s12859-018-2200-8.

[34] J. Devlin, M.-W. Chang, K. Lee, 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), Association for Computational Linguistics, Minneapolis, Minnesota, 2019, pp. 4171–4186. doi:10.18653/v1/N19-1423.

[35] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, J. Kang, Biobert: a pre-trained biomedical language representation model for biomedical text mining, Bioinformatics 36 (2019) 1234–1240. doi:10.1093/bioinformatics/btz682.

[36] E. Alsentzer, J. Murphy, W. Boag, W.-H. Weng, D. Jindi, T. Naumann, M. McDermott, Publicly available clinical BERT embeddings, in: Proceedings of the 2nd Clinical Natural Language Processing Workshop, Association for Computational Linguistics, Minneapolis, Minnesota, USA, 2019. doi:10.18653/v1/W19-1909.

[37] Q. Wei, Z. Ji, Y. Si, J. Du, J. Wang, F. Tiryaki, S. Wu, C. Tao, K. Roberts, W. Qi, Relation extraction from clinical narratives using pre-trained language models, AMIA ... Annual Symposium proceedings. AMIA Symposium 2019 (2020) 1236–1245. doi:10.1186/1471-2105-12-S2-S1.

[38] C. Quan, M. Wang, F. Ren, An unsupervised text mining method for relation extraction from biomedical literature, PLOS ONE 9 (2014) 1–8. doi:10.1371/journal.pone.0102039.

[39] A. Alicante, A. Corazza, F. Isgrò, S. Silvestri, Unsupervised entity and relation extraction from clinical records in italian., Computers in biology and medicine (2016) 263–275. doi:10.1016/j.compbiomed.2016.01.014.

[40] H.-J. Lee, S.-H. Shim, M.-R. Song, H. Lee, J. C. Park, Comagc: a corpus with multi-faceted annotations of gene-cancer relations, BMC Bioinformatics 14 (2013). doi:10.1186/1471-2105-14-323.

[41] R. Xing, J. Luo, T. Song, Biorel: towards large-scale biomedical relation extraction, BMC Bioinformatics 21 (2020). doi:10.1186/s12859-020-03889-5.

[42] E. Charniak, A maximum-entropy-inspired parser, in: 1st Meeting of the North American Chapter of the Association for Computational Linguistics, 2000. URL: https://aclanthology.org/A00-2018.

[43] E. Charniak, M. Johnson, Coarse-to-fine n-best parsing and MaxEnt discriminative reranking, in: Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05), Association for

Computational Linguistics, Ann Arbor, Michigan, 2005, pp. 173–180. URL: https://aclanthology.org/P05-1022. doi:10.3115/1219840.1219862.

[44] D. McClosky, E. Charniak, M. Johnson, Automatic domain adaptation for parsing, in: Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics, HLT '10, Association for Computational Linguistics, USA, 2010, p. 28–36.

[45] S. Griffiths-Jones, R. Grocock, S. Dongen, A. Bateman, A. Enright, Griffiths-jones, s, grocock, rj, van dongen, s, bateman, a and enright, aj. mirbase: microrna sequences, targets and gene nomenclature. nucleic acids res 34(database issue): D140-d144, Nucleic acids research 34 (2006) D140–4. doi:10.1093/nar/gkj112.

[46] S. Gupta, A. A. Mahmood, K. Ross, C. Wu, K. Vijay-Shanker, Identifying comparative structures in biomedical text, in: BioNLP 2017, Association for Computational Linguistics, Vancouver, Canada,, 2017, pp. 206–215. URL: https://aclanthology.org/W17-2326. doi:10.18653/v1/W17-2326.

[47] N. Jindal, B. Liu, Mining comparative sentences and relations, in: Proceedings of the National Conference on Artificial Intelligence, volume 2, 2006.

[48] P. Qi, Y. Zhang, Y. Zhang, J. Bolton, C. D. Manning, Stanza: A Python natural language processing toolkit for many human languages, in: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, 2020. URL: https://nlp.stanford.edu/pubs/qi2020stanza.pdf.

[49] L. M. Schriml, E. Mitraka, J. Munro, B. Tauber, M. Schor, L. Nickle, V. Felix, L. Jeng, C. Bearer, R. Lichenstein, K. Bisordi, N. Campion, B. Hyman, D. Kurland, C. P. Oates, S. Kibbey, P. Sreekumar, M. Giglio, C. Greene, Human disease ontology 2018 update: classification, content and workflow expansion., Nucleic Acids Research (2019). doi:10.1093/nar/gky1032.

[50] Y. Peng, C. M. Croce, The role of micrornas in human cancer, Signal Transduction and Targeted Therapy (2016). doi:10.1038/sigtrans.2015.4.

[51] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, Jupyter notebooks – a publishing format for reproducible computational workflows, in: F. Loizides, B. Schmidt (Eds.), Positioning and Power in Academic

Publishing: Players, Agents and Agendas, IOS Press, 2016, pp. 87 – 90.

[52] D. Chen, C. Manning, A fast and accurate dependency parser using neural networks, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Association for Computational Linguistics, Doha, Qatar, 2014, pp. 740–750. URL: https://aclanthology.org/D14-1082. doi:10.3115/v1/D14-1082.

[53] M. Honnibal, Y. Goldberg, M. Johnson, A non-monotonic arc-eager transition system for dependency parsing, in: Proceedings of the Seventeenth Conference on Computational Natural Language Learning, Association for Computational Linguistics, Sofia, Bulgaria, 2013, pp. 163–172. URL: https://aclanthology.org/W13-3518.

Acknowledgments

This work was supported by the ExaMode Project, as a part of the European Union Horizon 2020 Program under grant 825292. We also sincerely thank Gianmaria Silvello and Stefano Marchesin whose suggestions helped in improving this study.