A Natural Language Interface for Querying Linked Data

Simon Tham
Christoffer Akrin

Faculty of Health, Science and Technology

Computer Science

C-Dissertation 15 HP

Supervisor: Nurul Momen

Examiner: Lothar Fritsch

Date: 2020-06-01

Abstract

The thesis introduces a proof of concept idea that could spark great interest from many industries. The idea consists of a remote Natural Language Interface (NLI), for querying Knowledge Bases (KBs). The system applies natural language technology tools provided by the Stanford CoreNLP, and queries KBs with the use of the query language SPARQL. Natural Language Processing (NLP) is used to analyze the semantics of a question written in natural language, and generates relational information about the question. With correctly defined relations, the question can be queried on KBs containing relevant Linked Data. The Linked Data follows the Resource Description Framework (RDF) model by expressing relations in the form of semantic triples: *subject-predicate-object*.
With our NLI, any KB can be understood semantically. By providing correct training data, the AI can learn to understand the semantics of the RDF data stored in the KB. The ability to understand the RDF data allows for the process of extracting relational information from questions about the KB. With the relational information, questions can be translated to SPARQL and be queried on the KB.

Keywords: SPARQL, NLP, RDF, Semantic Web, Knowledge Base, Knowledge Graph

Acknowledgments

We would like to thank our supervisor at Karlstad University, ***Nurul Momen*** for guidance and support.
We would also like to thank our supervisor from Redpill Linpro, ***Rafael Espino*** for advice and inspiring us in the fields of Natural Language Processing and the Semantic Web.
Lastly, we would like to thank Redpill Linpro for allowing us to do this project.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Roadmap 2

2 Background **4**

2.1 Semantic Web 4

2.1.1 RDF 5

2.1.2 Knowledge Graph 5

2.1.3 SPARQL 6

2.2 Natural Language Processing 6

2.2.1 The Stanford CoreNLP 7

2.2.2 Natural Language to SPARQL 8

2.3 Literature Survey 9

2.3.1 Three-pass 10

2.3.2 Survey Results 10

3 Design **12**

3.1 Build Automation & Framework 12

3.1.1 Spring 13

3.1.2 Java Build Tools 14

3.2 NLP Frameworks 15

3.2.1 Rasa 15

3.2.2 Apache OpenNLP 15
3.3 Initial Design 16
3.4 Natural Language to SPARQL 16
3.4.1 Relation Extraction 17
3.4.2 Relation Recognition 17
3.4.3 Parsing Triples to SPARQL 18
3.5 Remote Natural Language Interface 18

4 Implementation **21**
4.1 Interface 21
4.1.1 Initial Version 21
4.1.2 Final Version 22
4.2 Query Parser 23
4.2.1 Generating Triples 24
4.2.2 Formatting Triples 24
4.2.3 Relation Matching 26
4.2.4 Relation Gathering 26
4.3 Generating a SPARQL Query 28
4.4 Connect to a Repository 29
4.5 Training a NER Model 30
4.5.1 Conditional Random Field 30
4.5.2 Training Data 31
4.6 Final Pipeline Implementation 32

5 Evaluation **34**
5.1 Memory Usage 34
5.2 Ramification of Training Data 34
5.3 Fundamental Distinctions 35

6 Limitations & Future Work **36**

6.1 Word2vec 36

6.2 Generating Training Data 37

6.3 Distributed Network of Repositories 37

6.4 Interactive NLI 37

7 Conclusion **38**

Bibliography **39**

A Setting up the Query Service 45

A.1 Setup GraphDB Repository 45

A.2 Setup Application Properties 46

A.3 Setup VM Options 46

B Using the NLI 46

B.1 Example Queries 46

List of Figures

1.1 Roadmap Diagram describing the work flow of the project. 3

2.1 An overview of the CoreNLP pipeline 8

3.1 An overview of the initial design. 16

4.1 Early version of the web interface shows an example of how to find all the Star Wars movies. 22

4.2 Final version of the web interface shows an example of how to find the orbital period of all planets. 23

4.3 Overview of the final pipeline implementation. 33

List of Tables

2.1 An overview of our systematic literature survey. 9

3.1 Performance test for NER and POS tagging 16

Listings

2.1 An example of a simple SPARQL query. 6
3.1 An example of Maven build script. 13
3.2 An example of Gradle build script. 14
3.3 An example handshake sent by the client using WebSocket. 20
3.4 WebSocket response handshake sent by the server. 20
4.1 Formatted triples for the question *What is the gravity of Tatooine?*. . . . 25
4.2 SPARQL query for fetching the vocabulary from `uri`. 27
4.3 SPARQL query for the question *What is the height of Vader?*. 29
4.4 Example of creating a repository. 29
4.5 An example snippet of training data used by the NER model. 31

Abbreviations

NLP	Natural Language Processing
NLI	Natural Language Interface
KB	Knowledge Base
KG	Knowledge Graph
RDF	Resource Description Framework
NER	Named Entity Recognition
POS	Part of Speech
WS	Web Service

Chapter 1

Introduction

The Semantic Web provides many great technologies for understanding more complex data structures. Furthermore, the recent use of Natural Language Processing (NLP) and Knowledge Graphs (KGs) have had an important role in analyzing data, and adopting the business.

Knowledge Graphs and the SPARQL query language are areas of growing interest in the IT industry of today. This technology is already being used by many large corporations to analyze user behavior and relations between various topics and persons. Used in the right way, this technology can create insight into large portions of data and help organizations in making the right decisions. Facebook, Google and their likes are already using this technology to help advertisers target the right audience with the right messaging. This has sparked an interest also from the traditional industries on how they can make use of this powerful technology. For example, in the car industry, companies may gather information about the vehicle and easily extract necessary information. When the interest for the technology broadens and moves into new organizations, one challenge is the user ability to adopt and make use of the technology.

1.1 Motivation

NLP is an interesting branch of Artificial Intelligence. The idea is to ease the interaction between the human languages and computer language. With the help of NLP, large amounts of Linked Data [1] can become available for the everyday user. This technology could potentially also be used to give users access to KGs and data analysis. Currently, in order to query Linked Data, the user needs knowledge of a certain query language.
This thesis focuses on the Semantic Web technology, RDF [2], and the RDF-specific query language, SPARQL [3]. We evaluate the possibility to translate natural language to SPARQL, and retrieve information from a KB.

1.2 Roadmap

Chapter 2 describes relevant information that plays a part in the implementation. First, we introduce the Semantic Web and some of the technologies it contains. Second, we briefly explain the syntax and semantics of the query language SPARQL. Afterwards, we discuss NLP and how we can utilize it in our project. Lastly, we discuss how we approached the literature study in order to find relevant work, and answer how we can contribute in the research area.
Chapter 3 introduces a design approach for a remote NLI. We discuss how the initial vision is outlined. Also, we motivate the technical design decisions made, such as build automation, network architecture, and NLP framework. Lastly, we describe the design of the pipeline for translating natural language to SPARQL.
Chapter 4 describes the proof of concept implementation. We discuss how the web interface was modified during development. We also describe how the server connects to a repository in order to query SPARQL. Later, we explain the approach taken for parsing a query, how the model trains, how triples are generated and formatted, and how we match relations with the KB. Lastly, we describe the final pipeline implementation, showcasing how a query gets processed by the system.

In Chapter 6 we describe possible future work of the prototype. The project has a lot of potential for future development, and in this chapter we describe ideas that were planned or took shape during development.

Figure 1.1 presents a diagram describing the work flow of the project, with the given time frame. The project can be categorized into four different blocks, as seen in Figure 1.1. Under the timeline, the project is split into phases describing more specifically the work being done at the given time.

In the Research Phase we gather relevant knowledge, and literature for a survey. Towards the Initial Design phase we discuss how the initial design could come to intuition. With a basic intuition, the approach taken for the design needs to be defined, and executed.

In the Initial Prototype phase we implemented a bare bone prototype capable of querying KBs with SPARQL, using a simple web interface. At this point, NLP is not yet implemented and the user can only query using SPARQL directly. The Final Prototype phase is the largest phase, and contains the work for using natural language to query a KB. Also, the work accommodates a cleaner and more intuitive web interface.

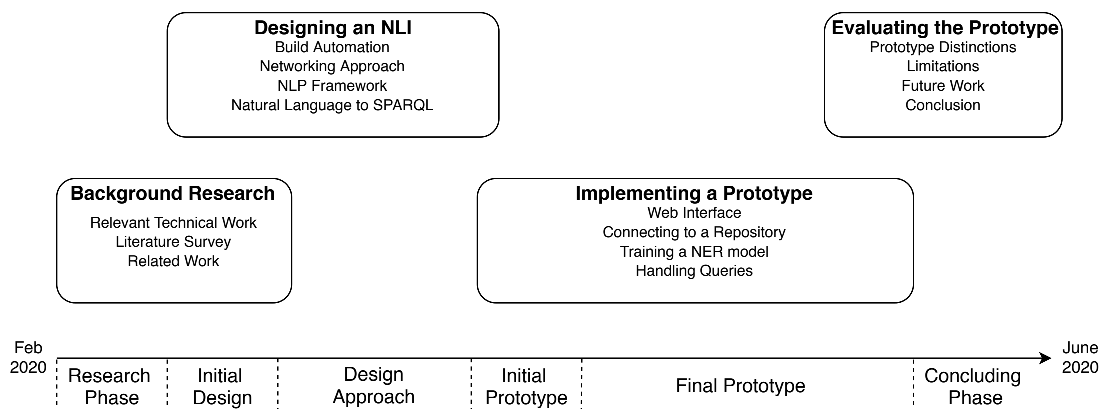

Figure 1.1: Roadmap Diagram describing the work flow of the project.

Chapter 2

Background

This chapter will first discuss relevant technical work, and later explain the approach taken for literature study. We begin by introducing the Semantic Web and some of its building blocks. Next, we discuss how NLP works, and how it can be implemented to translate natural language to SPARQL. Finally, we discuss techniques used for the literature study in order to gain information about relevant topics, and elaborate on relevant papers that contains similar work.

2.1 Semantic Web

The Semantic Web is a standard, set by the World Wide Web Consortium (W3C) [4], and is a vital component for the next generation internet known as Web 3.0. The overall premise of The Semantic Web is to translate large amount of data into machine-readable information [5]. The Semantic Web contains numerous technologies that for example makes it possible to query the data, or to generate new relations between data. To make The Semantic Web a reality, we need large amounts of data to work on, which the web provides to us. Also, the data does not only need to be accessible, but also need relations between data, known as Linked Data [1]. In the coming subsections, some of the vital technologies used in The Semantic Web are briefly explained.

2.1.1 RDF

The Resource Description Framework [2] is the basic framework for The Semantic Web, and is the format that allows anyone to describe any kind of statement, and model them together. RDF statements can be boiled down to what is known as triples. Triples are the fundamental data structure of RDF and are divided into three parts; subject, predicate, and object. With triples we can describe the relation between data. For example: *Shakespeare Wrote Hamlet*, where *Shakespeare* is the subject, *Wrote* is the predicate, and *Hamlet* is the object. With RDF we can create a Directed Graph of Linked Data, structured by a group of triples [6].

2.1.2 Knowledge Graph

The term Knowledge Graph was first coined by Google back in 2012 [7]. From that point KGs have had a major widespread adoption, both academically and commercially. There is a handful of different definitions regarding KGs. A common interpretation is that a KG represents a collection of linked descriptions of entities, and is a graphical representation of a KB. The KB contains all the data stored in the KG. KB is also sometimes referred to as an ontology. In order to avoid confusion, we will only use the terms KB and KG to describe storage of data and its graphical representation respectively.

The intention of KGs is to describe data in a formal structure that is readable by both human and machine. Together, the entities create a network of relations between each other, and can for example be interpreted as triples to represent an RDF Graph. However, not all RDF Graphs should be considered KGs. It is important to consider that it is not always necessary to represent the semantic knowledge of data. Relations between entities define the KG, not the language used for representation of data [7, 8].

A handful of large, openly available KBs exist today [9]. For example, DBpedia is a Wikipedia data extraction tool, and allows us to access information from Wikipedia in the form of a KB [10]. With DBpedia we can query on more than 38.8 million entities that Wikipedia offers [11].

2.1.3 SPARQL

The query language SPARQL Protocol and RDF Query Language (SPARQL) is necessary when accessing information described in the RDF format. SPARQL allows us to model questions that are written in natural language, and query them on data that follows the RDF format. For example, the SPARQL query for the question: *What are all the poems written by Shakespeare?* would become the query as seen in Listing 2.1.

```
1 PREFIX ex: <http://ShakespeareExample>
2
3 SELECT ?poem WHERE {
4     ?poem ex:medium ex:Poem .
5     ?poem ex:author "Shakespeare" .
6 }
```

Listing 2.1: An example of a simple SPARQL query.

In this case we are using a **SELECT** query to gather all poems written by Shakespeare. The first triple selects all the mediums that are considered poems, and the second triple checks if the poem has the author Shakespeare. The *?* token indicates a variable and can match any resource given by the RDF data set. The prefix *ex* is our prefixed name that represent resources and provides the RDF data set. Resources are accessed with a Uniform Resource Identifier (URI), which in this example has the string *http://ShakespeareExample* [12]. Prefixes also operate as namespaces by providing organization between objects originating from different data sets [6].

2.2 Natural Language Processing

NLP is the transformation of human language to machine readable information. Semantic understanding of the human language is a very hard problem in the field of AI. Researchers have yet to master the arts of fully translating the human language with NLP. NLP can be split up into multiple tasks, and we will briefly explain some of the

most common tasks [13].

The first task for NLP is to generate tokens from the words forming the provided sentence. To create tokens we use what is known as tokenization, which can for example present the words of a sentence in XML-format. With the tokens, we can use many different analysis tools to gain semantic understanding of the sentence. One of the key analysis tools is Part-Of-Speech (POS) Tagging. POS allows for tagging of tokens. The tag indicates the syntactic role of the token, for example plural, noun, adverb [13, 14, 15]. Morphological analysis is another way to analyze tokens, and allow us to find the base form of the token, also known as a lemma [15].

Lastly, a very important task done by NLP is—Named Entity Recognition (NER). NER recognizes potential entities in a sentence and labels them such as PERSON, TIME, LOCATION [13].

2.2.1 The Stanford CoreNLP

CoreNLP is an NLP toolkit built by Stanford University and is one of the most widely used natural language analysis tools [15]. CoreNLP provides a lightweight JVM-based (Java Virtual Machine) framework, that is split up into multiple APIs, known as annotators. The annotators can be used separately to annotate the raw text provided. Also, the annotators allow the user to tailor their own pipeline by creating a flow of execution on the raw text. Annotators generate *analyses of the text*, and outputs an annotated text as a result. CoreNLP also allow us to create our own annotators that we can add to the pipeline [16, 15].

Originally, CoreNLP was designed for internal use only, and was a set of independent natural language analysis components, which were glued together to create the open source framework CoreNLP, showcased in Figure 2.1 [15].

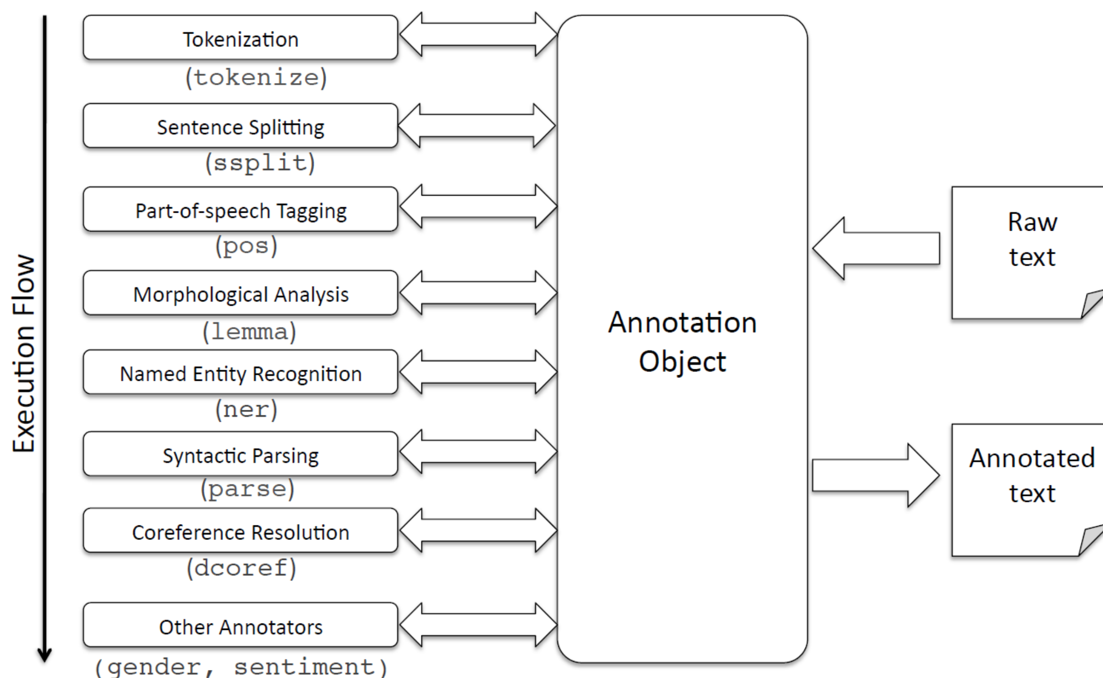

Figure 2.1: An overview of the CoreNLP pipeline [16].

2.2.2 Natural Language to SPARQL

With large scale KBs, there is a need to manage searches on the KBs in an easy and user-friendly way. However, forming SPARQL queries to search an RDF KB is not the simplest task, even for advanced users. One solution to the problem is to accept questions in natural language, and translate them to SPARQL queries. Due to high level of expressiveness in natural language, complex queries can be generated. If queries can be translated from natural language to SPARQL, human users can search KBs without having to learn SPARQL [17, 18]. However, translating natural language to SPARQL is not a trivial task. One of the challenges is to correctly identify the desire of a user query. The semantics of words can vary in different contexts. For example, *how big* might refer to *length*, *height*, or *population*. Also, the same fact can be expressed in different triple forms and even in multiple triples [18, 19].

2.3 Literature Survey

The literature study is mainly made using two similar methods; snowballing and reverse snowballing [20]. Before starting the snowballing, we identify relevant work as a starting set of papers. For example, the starting set of papers could be identified using Google Scholar [20]. Some of the keywords for identifying relevant papers are: *NLP*, *CoreNLP*, *Semantic Web*, *SPARQL*, *RDF*, *Knowledge Graph*. The keywords are used in different permutations and combinations.

First a tentative start set is created from the search for papers to include in the start set. The actual start set are the papers from the tentative start set, included at the end of the literature study.

Snowballing means identifying new papers which are citing the paper being examined. To decide if a paper is relevant for our work, a technique called three-pass [21] is used, which will be discussed in the coming subsection. Reverse snowballing refers to the idea of going through the reference list of the current paper being examined, and finding new papers to include.

Start Set—Iteration one	**Iteration two**	**Iteration three**
PANTO (Wong et al. [22])	-	-
AutoSPARQL (Lehmann, Bühmann [23])	AutoSPARQL	AutoSPARQL
SPARKLIS (Ferré [24])	SPARKLIS	-
Controlled Natural Language (Fuchs, Schwitter [25])	-	-
SNOMEDCT (Jin-Dong, Cohen [17])	SNOMEDCT	SNOMEDCT
SQUALL (Ferré [26])	SQUALL	-
Triplet Extraction (Rusu et al. [27])	Triplet Extraction	-
NLP-Reduce (Kaufmann et al. [28])	NLP-Reduce	NLP-Reduce
The Penn Tree Bank (Taylor et al. [29])	The Penn Tree Bank	-
Gradle in Action (Muschko [30])	Gradle in Action	Gradle in Action
NLP for the Semantic Web (Maynard et al. [31])	NLP for the Semantic Web	-

Table 2.1: An overview of our systematic literature survey.

2.3.1 Three-pass

The three-pass approach is a way of reading through papers and giving us the chance to make up our mind early, whether the paper is interesting for us. Instead of reading through the whole paper from start to end, we read the paper in three passes.
In the first pass we get an overview of the paper, where we read the title, abstract, introduction, and conclusion. We also glance over the section and subsection headers. The first pass is good for deciding if we want to keep reading, or if the paper is not interesting to us, or if we have to learn more about the area to understand the paper.
In the second pass, we look carefully at figures and diagrams. This pass is also used to look for relevant references for further reading.
In the third pass, we read through the whole paper. Table 2.1 shows a small sample of articles, used in our research, and what articles were excluded in each pass. The inclusion criterion consists of helping to understand the process of translating natural language to SPARQL. Also, papers are included to help understand and choose appropriate tools for the design process.

2.3.2 Survey Results

Querying on the Semantic Web in the form of natural language has had multiple different implementations throughout the years. This section provides information about some NLIs [32] used on the Semantic Web. Lastly, we discuss our possibilities for contribution in the field of NLIs.
Ginseng is a guided input natural language search engine. In other words, Ginseng does not use a predefined vocabulary and does not try to understand the syntax, nor the logic of the query. Instead, Ginseng uses its preloaded KBs to gain a vocabulary. The user has to tailor their questions according to the vocabulary, which limits the freedom of asking any type of question. However, by querying relevant questions, Ginseng will provide accurate results; hence the name guided input. Ginseng uses the RDF Data Query Language (RDQL), predecessor to SPARQL [33, 34].

AutoSPARQL is an NLI that utilize supervised machine learning. The system learns the concept with help of good and bad examples as input. The user also does not need any knowledge of the underlying KB, nor any previous SPARQL prowess [23].

NLP-Reduce uses a lexicon, a query input processor, a SPARQL query generator, and a KB access layer to transform natural language to SPARQL queries. The lexicon is automatically built when a KB is loaded into NLP-Reduce. It is built by extracting all the triples that exist in the KB. The input query processor removes any punctuation mark and stop words, and passes the words to the SPARQL query generator. The query generator tries to generate SPARQL queries by matching the queries to the triples stored in the lexicon [28].

FREyA includes features to let the user improve recall and precision. Therefore it is called an interactive NLI. FREyA uses *disambiguation dialog* and involves the user to resolve any ambiguity. For example, if the user asks a question about *New York*, the user is asked to disambiguate, because it can refer to the state or the city. With the help of the user, the system learns and improves over time. When there is no ambiguity left to resolve, the system identifies the *answer type*, and form triples, which are used to form SPARQL queries [19].

We believe that our work can contribute towards a proof of concept idea, exhibiting the usefulness and practicality an NLI provides. Many industries store large amount of data, and the power of NLIs could prove great usefulness for analyzing any kind of data. Our work is intended to provide a stepping stone for further development of NLIs. Also, we want to show how it could be useful for many industries that manage some sort of information.

Chapter 3

Design

When designing an NLI, many different approaches are possible. In our approach, we want to create an NLI that can be accessed remotely, and query on KBs. Also, the NLI should rely on machine learning when parsing questions, instead of using a predefined set of rules. During the development process of the NLI, many decision were made in order to achieve our vision. In this chapter we describe the reasoning behind the choice of adopted tools and frameworks. Furthermore, we discuss the initial design and describe the process of translating natural language to SPARQL.

3.1 Build Automation & Framework

To ease the development of our project, we use a framework called Spring [35]. When we create a Spring project, we get to choose a build automation tool. We can choose between Gradle or Maven [36, 37]. Maven uses XML in its build script which can get complex as the project grows, but this will be discussed further in the coming subsection. To avoid XML, we use Gradle as our build automation tool.

3.1.1 Spring

Spring Boot is a Java-based framework used to make the development process easier. Spring Boot helps with a number of different tasks. To name a few: it eases the dependency management, makes it possible to run the application independently, and offers an easy way to get started with the project [38]. From the start, Spring makes it easy to set up a project. All we have to do is choose a build tool, what language we want to use, and we can also choose what dependencies to add. Spring sets up the project for us with our chosen build tool, the correct dependencies, etc.

For implementation feasibility, the application has to run independently. Spring contains infrastructure support for developing stand-alone applications.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.example</groupId>
    <artifactId>Example</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

Listing 3.1: An example of Maven build script.

```
plugins { id 'java' }
group 'org.example'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
    mavenCentral()
}
dependencies {
    testCompile group: 'junit', name: 'junit', version: '
       4.12'
}
```

Listing 3.2: An example of Gradle build script.

3.1.2 Java Build Tools

Maven and Ant are two tools that are highly used to build Java projects. Both Ant and Maven use XML[1] for their build script, which is good for describing hierarchical data, but is not as effective for describing program flow and conditional logic. XML has a tendency to get hard to read and understand as the project grows. Listing 3.1 and 3.2 displays the difference between a Gradle and Maven fresh build script. When the project grows and more dependencies are added, the XML code tends to get cluttered and become difficult to interpret. This is an advantage of Gradle, which does not use any XML at all, but instead uses a language called Groovy[2]. Compared to XML, Groovy is easier to read and understand [30].

Gradle is the most recent addition to the build automation tools for Java. Gradle has learned from Ant and Maven, and combines their best features to create a better build tool. For example, Gradle combines the flexibility from Ant, the dependency

[1] XML: Version 1.0

[2] Groovy: Version 4.0

management from Apache Ivy, and convention over configuration from Maven [30].

3.2 NLP Frameworks

There are a handful of NLP APIs that we can choose between, which all are using machine learning to annotate text. CoreNLP became a natural choice because it is very simple to use, yet very fast and accurate [39]. Also, CoreNLP has a documentation that is easy to understand, and contains practical examples for how to properly use the framework. We will now briefly introduce the frameworks we researched, but were discarded.

3.2.1 Rasa

Rasa is an open source framework for developing contextual assistants, and is written in Python [40]. An external SDK (Software Development Kit) exist for using Rasa with Java [41]. The SDK allow us to communicate with Rasa using a REST endpoint. Overall, we felt that Rasa might not be the perfect fit for our project. Rasa learns from receiving natural language, and decides on the best response with the conversation history in mind. Our design does not handle conversations, but handles each question separately, without past questions in mind. We will discuss the possibilities more for dynamically creating a model over time in Chapter 6.

3.2.2 Apache OpenNLP

With OpenNLP we can build our own NLP pipeline; much like CoreNLP. OpenNLP works very similarly to CoreNLP and both are written in Java. Both frameworks require tokenization in order to do any semantic extraction. In our program, POS tagging is a critical component in the pipeline, and CoreNLP outperforms OpenNLP when in comes to POS tagging [42]. However, the OpenNLP NER tagging is significantly faster, as seen in Table 3.1, it is limited regarding accuracy. For example, OpenNLP NER tagger can

	OpenNLP	CoreNLP
POS	11.65s	2.69s
NER	11.26s	18.04s

Table 3.1: Performance test for NER and POS tagging [42].

not understand abbreviations with punctuation, which CoreNLP can. Also, OpenNLP needs separate models to understand words that are not English-alphabetical, which CoreNLP can handle with the same model [42].

3.3 Initial Design

Initially, the project was presented by Redpill-Linpro in Karlstad, Sweden [43]. The purpose of the project is to explore the possibilities of using NLP to query the Semantic Web. The initial design, as seen in Figure 3.1, depicts an overview of the project's vision. The user asks questions in natural language using a web interface. The question will be sent to a server, and an NLP framework will extract relevant information about the questions using syntactic analysis. The analysis will then be translated into a SPARQL query. With the SPARQL query we can access data from a KB in the form of RDF triples.

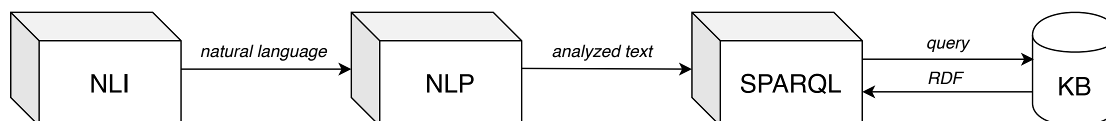

Figure 3.1: An overview of the initial design.

3.4 Natural Language to SPARQL

As mentioned in Section 2.2.2, the process of translating natural language to SPARQL is not a trivial task, and requires multiple subtasks in order to be attained properly. In

this section we will discuss the design approach of all the necessary tasks contained in the pipeline. First, we will discuss how we can define relations from natural language. Second, we want to match the defined relations with the vocabulary of the KB. Lastly, we need to parse the created triples to a SPARQL query, which should be a trivial task if the triples are correctly defined.

3.4.1 Relation Extraction

When the user defines a question, the AI needs to extract relations between entities in order to find what the user wants in return. However, a question does not accommodate any relational information. For example, the question *Where was Shakespeare born?* does not tell the AI any information about the relation between *Shakespeare* and his *birthplace*. Nevertheless, by using CoreNLP we can annotate the question and reconstruct it as a statement. In our case, the question would translate to *Shakespeare be bear at location*. The statement also makes sure to write the tokens in their lemma form; *was* becomes *be*, and *born* becomes *bear*. The CoreNLP framework can now extract relational information about the entities. The relations can be defined in the form of RDF triples: *subject-predicate-object*. Once again in our case, the triple would roughly be defined as following: *Shakespeare - be bear at - location*. Furthermore, it becomes quite obvious that *location* is the variable the user is asking for.

3.4.2 Relation Recognition

When the triples have been roughly defined, we need to match the predicates with the existing relations defined in the remote KB, also know as a vocabulary. In order to recognize the relations, we need to link the relations with the existing vocabulary. For example, the NLP recognizes the relation *be bear at*; and in our vocabulary the corresponding relation is *birthplace*. If we were to query with the relation *be bear at*, we would not get any matches, because the KB uses a different predicate, in this case *birthplace*. We design the system to query on KBs from GraphDB. With GraphDB we

can visually display a graph of the predefined triples. Also, GraphDB allows us to test the system on smaller KBs, which simplifies the implementation part of the project. However, the design of the system should not limit the scalability. In other words, the size of the KB should not increase the difficulty of the task, but can make it harder to debug during implementation.

3.4.3 Parsing Triples to SPARQL

With correctly defined triples, the translation from triples to a SPARQL query becomes definite. Apache Jena is a framework for Java, and contains multiple tools to create Semantic Web and Linked Data applications. ARQ is one of the tools provided, and is a SPARQL processor for Jena. With ARQ we can manipulate SPARQL, or even create SPARQL queries from scratch [44].

All the queries in our system will have certain similarities. We know for a fact that all the queries will be of the **SELECT** type, because we only want to select and provide data to the user. When creating a select statement as seen in Listing 2.1 we are selecting poem entities that match the relations. In other words, the AI needs to know what we are supposed to select. For example, the question *What are all the poems written by Shakespeare?*. As a human, it is trivial that we should select all the poems that was written by Shakespeare. For an AI it is not obvious what exactly we should select when parsing the query, and needs to figure this out in order to create the correct query. In the upcoming chapter, we will dig deeper into how we can solve the problem of selecting correct entities.

3.5 Remote Natural Language Interface

In order to develop an NLI remotely, as discussed in Section 3.3, we decided to use a client-server model. The client can query from anywhere with a given connection to the server. Also, it allows for having multiple seamless connections at the same time, from different locations. A client-server model offers superior flexibility over a local-only

connection. Although, with a client-server model, the users have to rely on a server to be stable and handle their queries. With a local-only connection, every user relies on their own system. To achieve a remote NLI, we design a WS (Web Service). The WS provides the possibility to query over the World Wide Web. With the help of the Spring Framework, we can develop the WS in Java.

When developing a WS we need to create a web server that handles the incoming requests. The design of a web server can be constructed in multiple ways. The first option is to create a REST (Representational state transfer) API. REST puts certain constrains on the design of a WS. Nevertheless, REST is very simple to use and implement, and allows for high scalability [45]. The reason we decided to not design a REST API was due to statelessness. When designing an NLI, states can be useful in order to create greater server flexibility, and lower latency. NLP takes great processing power and memory usage [46]. With states we can optimize the server greatly. Also, we realized that the response data sent back to the user can be very large, and therefore requires chunking of data. With a HTTP based protocol, we need to establish a new connection for every message sent between the server and client, which adds a lot of unnecessary data for each packet.

Another approach is to use WebSocket, which allow us to achieve a stateful connection [47]. WebSocket is a communication protocol that allows two-way communication between clients and server. We can create a single TCP (Transmission Control Protocol) connection to each client, and the client connects to the server by sending a HTTP handshake, as seen in Listing 3.3. The server will respond with a handshake, as seen in Listing 3.4, and a connection will be established [47].

```
1 GET /chat HTTP/1.1
2     Host: server.example.com
3     Upgrade: websocket
4     Connection: Upgrade
5     Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6     Origin: http://example.com
7     Sec-WebSocket-Protocol: chat, superchat
8     Sec-WebSocket-Version: 13
```

Listing 3.3: An example handshake sent by the client using WebSocket.

```
1 HTTP/1.1 101 Switching Protocols
2     Upgrade: websocket
3     Connection: Upgrade
4     Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
5     Sec-WebSocket-Protocol: chat
```

Listing 3.4: WebSocket response handshake sent by the server.

When a connection is established, the server does not need to differentiate between messages. We can simply just handle each client session separately, and listen for incoming packets.

Because we are using a WebSocket based WS, we decided to design a web page to represent the NLI, which allows clients to connect and send questions to the server. The web page logic is designed using the JavaScript library jQuery[3]. With jQuery it becomes much easier to design the WebSocket part of the web page. Consequently, the web page also needs to handle sending and receival of packets.

[3]jQuery: Version 3.5

Chapter 4

Implementation

In this chapter we describe the approach taken to implement a prototype for translating natural language to SPARQL, as described in Section 3.4. For implementation feasibility, we used a rather small KB, containing information from the Star Wars universe. We will describe how the NLI changed over time through development, and how we solved problems encountered along the way. Lastly, we introduce the final version of the prototype and how it handles a query.

4.1 Interface

We require a simple web interface where the user can enter a question in natural language and receive an answer. To create the interface, we use CSS, HTML and Javascript. CSS and HTML describes the design of the interface, whereas Javascript adds functionality to it.

4.1.1 Initial Version

In the early version of the interface, which is shown in Figure 4.1, we need to enter a SPARQL query which is sent to GraphDB. To send a query to GraphDB, a session must first be established with the server, by pressing the *connect* button. A SPARQL query

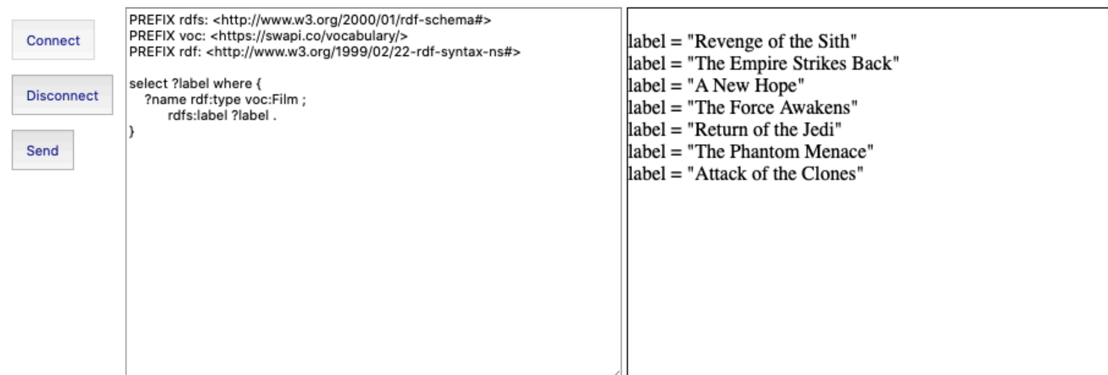

Figure 4.1: Early version of the web interface shows an example of how to find all the Star Wars movies.

can then be entered into the text box and sent when pressing the *send* button. The server will connect to the repository and send the query to the endpoint. The endpoint replies to the server and the server forwards the answer to the client, which shows up in the right box.

4.1.2 Final Version

The goal is to not require a SPARQL query, but instead write a question in natural language. The interface should also be user friendly, e.g. the user should not have to press the *connect* or *disconnect* buttons. The server should handle the connection and let the user know when they are connected or disconnected. For our final version of the interface, we chose to remove the *connect* and *disconnect* buttons. The server now establishes connection, and a text appears to let the user know when they are connected. If the user disconnects, the text changes and lets the user knows they have been disconnected. Also, the text area where the user asks the question has been changed to a single input line. The retrieved data will show up underneath the input field and is presented in tabular form. Figure 4.2 shows our final version with a slightly more difficult question than in Figure 4.1.

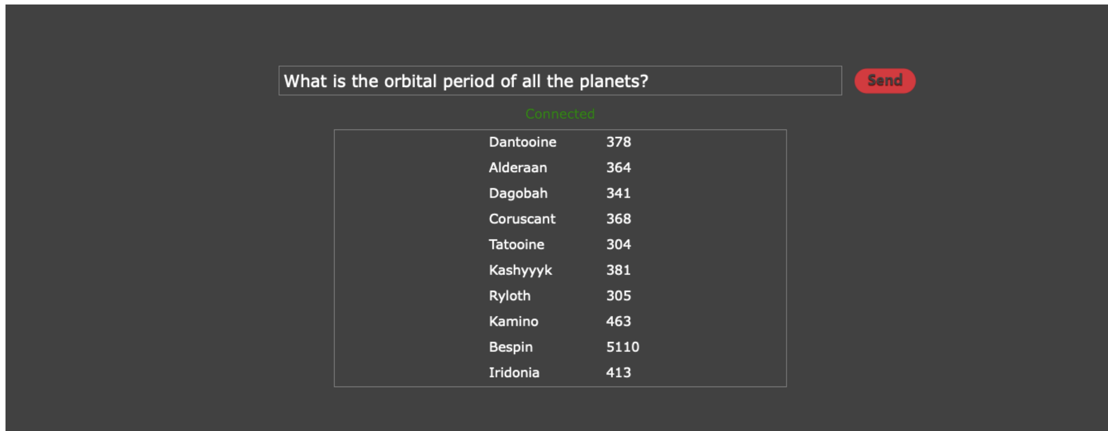

Figure 4.2: Final version of the web interface shows an example of how to find the orbital period of all planets.

4.2 Query Parser

When the server receives a query from one of the sessions, the server needs to parse the query into SPARQL before sending it to the repository. The translation from a query to SPARQL is handled by the `QueryParser` class. Before parsing queries, we need to initialize the `QueryParser` properties used by the pipeline. When creating a pipeline, we use the `StanfordCoreNLP` [48] class, and pass the `java.util.Properties` instance into the constructor. The properties for the annotators key, are defined in order of execution as following: *tokenize, ssplit, pos, depparse, lemma, ner, natlog, openie.* We make sure to include all the annotators we are using in the program, otherwise we might need to load in the models while parsing a query.

The `QueryParser` class can handle multiple questions with a single instance. The query string passed into the constructor can contain multiple questions, separated with a *?*. The separation is made possible by the `Document` [49] class in CoreNLP, which allows us to split the sentences in a string, and annotate the sentences seamlessly during the same annotation.

As mentioned in Section 3.4.1, the query needs to be translated to a statement in order for the NLP framework to find relations. In order to translate the question into a

statement, we use the `QuestionToStatementTranslator` [50] class. The `QuestionToStatementTranslator` class provides a function `toStatement` that takes a list of the annotated words, written as a question. The function transforms the annotated words into a statement, and returns a new list of annotated words. With the newly created statement, the sentence contains relational information that can be extracted to generate triples.

4.2.1 Generating Triples

When generating triples we use OpenIE (Open Information Extraction) [51], which is useful when limited on training data. Generally, the questions translated into statement will follow a similar pattern, and will make it possible for the OpenIE to extract relations. Therefore, we do not need to train our own relation extractor. Also, OpenIE is very fast compared to the Stanford relation extractor, and can process around 100 sentences per second per CPU core [51].
Before extracting the triples we need to make sure that the predicate is formatted correctly. In order to format correctly we analyze the POS tags of the tokens. For example, when the OpenIE extracts triples from the statement: *thing be the orbital period of all the planets*. The final triple should be: *thing* - *orbitalPeriod* - *planet*. In this case the word *orbital* is an adjective that modifies the noun *period*. So we concatenate the words and write it back in camel case, which is the naming standard used in the repositories. If we avoid the concatenation, the OpenIE can find undesirable relations. Another example would be: *thing be the eye color of Shakespeare*, which has two nouns as the predicate; *eye* and *color*. To match the predicate in the vocabulary, we simply concatenate the nouns to create the predicate *eyeColor*.

4.2.2 Formatting Triples

After creating the OpenIE triples, we need to format the triples before creating a SPARQL query. The triples received from the `QueryParser` follow common patterns,

which simplifies the algorithmic approach when formatting to SPARQL readable triples. We can split up the queries into two main different categories; queries with an entity, and those without an entity. When the query contains an entity, we can predict that the answer should contain some sort of information related to the entity. For example, if we ask the question: *What is the gravity of Tatooine?*, the `TripleFormatter` would find the NER tag PLANET for *Tatooine*. With a known NER tag, the SPARQL query should select all the planets containing the label *Tatooine*. The triples for the example question would become as following:

thing - be - gravity
thing - be gravity of - Tatooine

With the triples we know from previously that *thing* becomes the answer we want to select. In this case, *thing* will be the subject that relates to the gravity of the planet *Tatooine*. In other words, the triples would be formatted as seen in Listing 4.1.

```
1 ?root a voc:Planet .
2 ?root rdfs:label "Tatooine" .
3 ?root voc:gravity ?answer
```

Listing 4.1: Formatted triples for the question *What is the gravity of Tatooine?*.

We define the SPARQL variable *?root* for selecting the entity object. First, we select all the planets. Second, we filter the planets by their label and only selects the planet with the label *Tatooine*. Lastly, we select the literal that is the object with the relation gravity of the planet.

If the query does not contain an entity, the formatting becomes slightly more trivial. For example, if we ask: *What is the gravity of all the planets?*. The only main difference is that we do not need to select a label for the entity, instead select the gravity for all the planets.

4.2.3 Relation Matching

The relation in the triples has to exactly match the relation defined in the KB. When the user asks a question, the relation can be defined in many forms. For example, in the Star Wars repository all the entities have a description. The description is mapped with the relation *desc* in the repository, and provides a literal object containing the description text. We want the user to be able to express the questions without having to provide the exact name of the relation. If we ask the question: *What is the description of Han Solo?*, the answer should contain the *desc* of Han Solo.

Initially, we wanted to use and train word vectors to find similar words. For example, the user asks for the *birthplace* of a character, but the repository has the corresponding relation *homeworld*. With word vectors we could possibly map the words to each other. However, we had no time to fully implement a Word2vec model, but we will discuss the possibilities of Word2vec more in Chapter 6.

Another important scenario to consider is the use of abbreviation. We implemented a proof of concept method for matching abbreviations. With the CoreNLP `SentimentModel` [52] we can use the model for word recognition. The `SentimentModel` class provides us with a `java.util.HashMap` containing a vector space of real words. We can check if the relation described by the user correlates to a key in the `java.util.HashMap`, which allows us to identify real words. For example, if the relation is *description*, but the repository contains the relation *desc*. The word vector contains the key for *description*, which implies that it is a real word. Furthermore, the word *desc* is a substring of *description*, so we can make the assumption that *desc* is the matching word in the vocabulary.

4.2.4 Relation Gathering

Before matching relations, we need to know what relations exist in the repository. We define the existing relations as the vocabulary of the KB. Fetching the vocabulary from the KB is necessary in order to match similar relations correctly. By implementing a

class for handling the vocabulary we can simply refer to a `Vocabulary` instance when parsing queries.

When creating a `Vocabulary` instance we pass a URI string to the constructor. The class generates a SPARQL query for selecting all the predicates in the KB:

```
1 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
2 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3 PREFIX voc: < uri >
4
5 SELECT DISTINCT ?p WHERE {
6     ?s ?p ?o FILTER(CONTAINS(str(?p), str(voc:))
7 }
```

Listing 4.2: SPARQL query for fetching the vocabulary from `uri`.

The query selects all the predicates in the KB. By selecting all the predicates we will end up with duplicates if the KB contains triples with identical predicates. To avoid duplication, we use the `DISTINCT` modifier, which eliminates predicates that have already been selected. The `FILTER` keyword is used for adding restrictions on the triples, and in our case we use the `CONTAINS` keyword to make sure that the predicates we select are from the `uri` specified in the `Vocabulary` constructor. We do not need the predicates that exist in the other URIs. The `str(voc:)` variable obtains the URI of the prefix `voc` as a string.

After creating the SPARQL query, we want to send it to the connected repository. The repository will return the vocabulary. We have to keep in mind that a vocabulary can be quite large and should not be dynamically allocated. It might be necessary to chunk the data and serialize it into a file. In our case, the KB is very small and does not need chunking or serialization.

4.3 Generating a SPARQL Query

From the triples generated, we need to create a SPARQL query. Our initial idea included Apache Jena and the library ARQ to create SPARQL queries. The ARQ library has a lot of features to create and manipulate queries. However, because we know that we are only using **SELECT** queries, and we had a relatively small KB to practice on, we could figure out a pattern. Once we noticed that almost all the questions could be boiled down to very similar SPARQL queries, we decided to generate our own SPARQL queries.
Our idea was to utilize SPARQL functions such as `BIND`, `OPTIONAL`, and `FILTER`. Listing 4.3 shows what the query would look like for the question *What is the height of Vader?*.
In the Star Wars repository, Vader is defined as a character. Therefore Vader is assigned the NER tag CHARACTER. We begin by selecting all the characters. Afterwards, we select the name of all the characters and store them in the variable *?rootLabel*. Next we use the relation *voc:height* to get the height of all the characters, which is saved in the variable *?answer*.
We need to filter them to only get the values the user is asking for. For this query, `FILTER` and `CONTAINS` are used to find literals that match a certain string. `CONTAINS` performs the check and if *?rootLabel* contains Vader, it will return true, else false. Since *?rootLabel* contains all the names, it will return true only for Darth Vader. `FILTER` takes a condition that returns a boolean value and uses only the results that return true. Next we use the `OPTIONAL` function, which creates a new binding if possible. A new binding can be created if the relation asked for exists, else the object variable will be empty. In this example, *?answer* does not have the relation *rdfs:label*, it actually contains a literal, more specifically the height of Darth Vader. If we would not use `OPTIONAL`, we would not get any information with this query since *?answer* does not have the relation we ask for.
Lastly in this query we use `BIND`, `IF`, and `isURI`. `isURI` takes a variable and returns true if it is a URI, else false. The `IF` function is a bit more interesting and takes

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX voc: <https://swapi.co/vocabulary/>

SELECT DISTINCT ?answerLabel WHERE {
    ?root a voc:Character .
    ?root rdfs:label ?rootLabel .
    ?root voc:height ?answer .

    FILTER(CONTAINS(?rootLabel, "Vader")) .
    OPTIONAL {?answer rdfs:label ?literal .}
    BIND(IF(isURI(?answer), COALESCE(?literal),
        COALESCE(?answer)) AS ?answerLabel) .
}
```

Listing 4.3: SPARQL query for the question *What is the height of Vader?*.

three arguments. The first argument is evaluated and if it is true, the second argument is returned, else the third argument is returned. In this example, the first argument is `isURI(?answer)`, second argument is `COALESCE(?literal)`, and the third is `COALESCE(?answer)`. `COALESCE` simply returns the value of the given argument. The returned argument is then bound to a new variable called *?answerLabel* with the `BIND` function.

4.4 Connect to a Repository

When a SPARQL query has been created, a connection to a repository must be established. To be able to connect to a repository, the URL to an endpoint is required. We can connect to multiple endpoints simultaneously if we want to query multiple RDF bases. However, to start we want to connect to a local endpoint through GraphDB.

```
Repository repo = new HTTPRepository(
"http://localhost:7200/repositories/graphDB");
RepositoryConnection connection = repo.getConnection();
```

Listing 4.4: Example of creating a repository.

Listing 4.4 shows how to create and connect to a repository with the entered URL as our endpoint. Once a connection is established with the endpoint, queries can be sent to that endpoint.
The `Repository` [53] interface represent a repository that contains RDF data that can be queried and updated. Access to the repository is acquired by opening a connection to it. `HTTPRepository` [54] takes a URL as input and tries to parse the server URL from the repository URL, and must be done before a connection can be established. Lastly, a connection to the repository is established, using the `RepositoryConnection` [55] interface, which is used for performing queries on a repository containing RDF data.

4.5 Training a NER Model

When querying a repository, the NLP framework needs to know what kind of entities exist. The standard NER model provided by Stanford CoreNLP is limited, and only looks for entities such as PERSON, LOCATION, NUMBER, etc. We need to create our own model were we define our own NER tags that suits the repository. For example, when querying the Star Wars repository, we want the entity tag CHARACTER that identifies characters in the question. In CoreNLP, we use the Sequence Classifier, more specifically the `CRFClassifier` [56] class, which allow us to train a model. We specify a property file to the Classifier, were we state options for training, and how the training data is structured. When training, the `CRFClassifier` uses a CRF (Conditional Random Field) model [56].

4.5.1 Conditional Random Field

Training a NER model using CRF has shown to be an appropriate model [57]. The CRF model will take context into account. For example, in the sentence: *Darth Vader emerges from the shadows.* The model tags both *Darth* and *Vader* as a CHARACTER. In this sentence we know that the neighboring tags belong to the same entity. The model takes neighboring tokens into account, and in our case would learn to treat *Darth Vader*

as one person [58].

4.5.2 Training Data

In the properties file[1] we define the structure of our training data. Data is described per row, and in our case we tell the Classifier that the rows contain the columns *word*, and *answer*. The *word* column contains the next word in the text used to train on; and the *answer* column contains the NER tag of the word. An example snippet of training data is shown in Listing 4.5. The NER tag *O* stands for *Other*, and is used as the default NER tag for tokens we do not want to define.

```
1 Chewbacca CHARACTER
2 growls  O
3 and O
4 Artoo CHARACTER
5 beeps O
6 with  O
7 happiness O
8 . O
```

Listing 4.5: An example snippet of training data used by the NER model.

For testing purposes we train a model for querying the repository with Star Wars related RDF data [59]. We used the movie script from the first movie as training data[2], where we split each word in the script, and tagged them with corresponding NER tags. For example, the repository contains entities for planets; so we will tag planet names in the text as PLANET. Also, the repository uses the entity name CHARACTER to define characters in the movie, as seen in Listing 4.5.

[1] `https://github.com/acke80/StarQ/blob/master/src/main/resources/properties/roth.properties`

[2] `https://github.com/acke80/StarQ/blob/master/src/main/resources/trainingData/trainingDataStarWars.txt`

The process of tagging each word individually is very time consuming, and should be automated. We created a script that finds common entities in the movie script, and tags everything else with *O*. We will discuss how the tagging process could be automated in a more generalized approach in Chapter 6.

4.6 Final Pipeline Implementation

The final implementation of the pipeline is presented in Figure 4.3. The pipeline describes how a query is received in natural language and what steps are taken to give back an answer to the user. When the question is received, it will be parsed and triples are generated. The triples will be formatted to fit the SPARQL syntax and then a SPARQL query is created with the formatted triples. The query is sent to a KB, and the KB replies with an answer, consisting of bindings. Bindings are variables with assigned values and are displayed to the user.

In Figure 3.1 we visualize the initial design of the prototype. Figure 4.3 inherits the abstraction of the initial design and provides more detail. The core premise of the initial design is still intact. We can view the initial design as a group of black boxes, being defined during the implementation of the prototype. Furthermore, Figure 4.3 describes the flow of execution; from the server receiving the query, to the user receiving an answer. The flow of execution is undefined in the initial design.

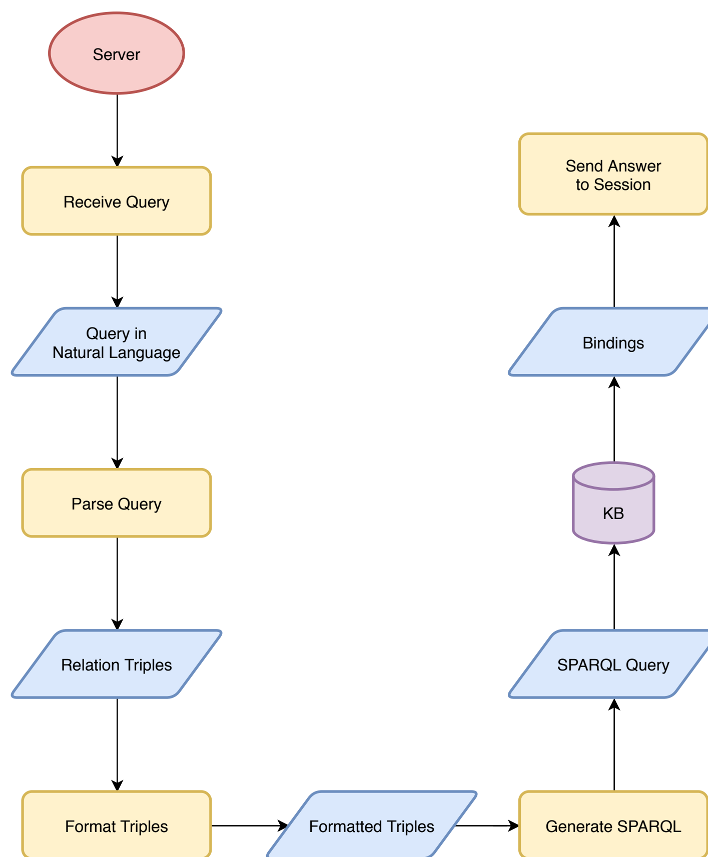

Figure 4.3: Overview of the final pipeline implementation.

Chapter 5

Evaluation

In this chapter, we evaluate the implemented prototype described in the previous chapter. We will discuss how the proof of concept implementation compares to the initial design and vision. Lastly, we discuss fundamental distinctions with related work, introduced in Section 2.3.2.

5.1 Memory Usage

The configuration of annotators used in the implementation does acquire certain amount of memory. The JVM needs a minimum of 4GB of memory in order to load in all the necessary annotators used by the pipeline. We have to keep in mind that the memory usage is only based on the the server side of the program. In other words, the clients accessing the web page do not need to care about memory usage. This is one of the great features of having a remote NLI.

5.2 Ramification of Training Data

The model used for NER tagging learns from contextual based data. In other words, the model needs data presented in context in order to extract entities from sentences. We

used the movie script from the first Star Wars movie *A New Hope* as training data. The structure of the text in the movie script is not perfectly suited for a CRF model, but demonstrates a proof of concept of our vision. Our implementation could be implemented to work with any KB, as long as sufficient training data is provided.

We noticed that the model needs plenty of data in order to extract entities in context. For example, when asking a question ending with *of Tatooine*, the model had no problem recognizing *Tatooine* as a planet. Nevertheless, when asking a question ending with *on Tatooine*, the model struggle to recognize the planet. We tried the same principle with a different more common planet, and the model successfully extracted the entity in both scenarios. The model needs to be highly confident when extracting entities, otherwise the annotation can produce false negative NER tags. We did however notice that the model is exceptional at avoiding false positives. Although, with such a small training set, the possibility for false positives becomes dubious, but could be indeterminate for larger training sets. For example, in the scarce scenario where two different types of entities, with identical or similar names, are frequent in the training set, the model might generate false positives during NER tagging.

5.3 Fundamental Distinctions

There are many different NLIs available and we mentioned a few in Chapter 2.3.2. Our work has some similarities and distinctions from these NLIs. For example, to use our NLI, the user needs some knowledge about the vocabulary, and tailor the questions accordingly. This is the case for Ginseng as well but instead of RDQL, we use SPARQL to query RDF KBs. Our program is an early prototype and does not include the more advanced features. For example our program does not have an interactive interface and a dialog with the user. We do not learn from user input either, to increase precision and recall.

Chapter 6

Limitations & Future Work

Many different features have been left for future work due to lack of time. Also, the prototype exhibits multiple limitations. In this chapter we will mention some of the work that we would have liked to implement, but did not have time for.

6.1 Word2vec

When the user asks a question, they need to have knowledge about the KB. In order to avoid the requirement of knowledge, we believe the use of a Word2vec model could assist in matching similar relations. For example, if the user asks about the *residents* of a location, but the KB contains the corresponding relation *inhabitants*, a word vector space could possibly map the queried relation with the existing relation. We have implemented a simplified module for recognizing abbreviations, and is our proof of concept that word vectors could play an important role in further development of the prototype. Possibly, we could train our own word vector model that is designed for a certain KB, the same way our NER model works. Training our own word vector model breaks the limitations of only matching real words, included in the English vocabulary. We could create our own vocabulary for the word vector space.

6.2 Generating Training Data

The training data used with the prototype was manually generated. We simply copied the movie script from the Star Wars movie *A New Hope*, and tagged the text with a simple script. The script looks for known entities and tags them. We have to manually define the entities to look for.

We wanted to create a program for gathering large amounts of sensible text, and the system would tag the words automatically. Possibly, the system asks for feedback when encountering new unknown entities, and the user would respond with an answer. Over time the system can tag the training data without needing a lot of feedback.

6.3 Distributed Network of Repositories

In Chapter 4.4 we talked about how to connect to a repository, and briefly mentioned the possibility to connect to multiple repositories. We used the Star Wars RDF data as our only KB and it worked well for implementing a prototype. It would be interesting to query bigger KBs and even multiple KBs simultaneously. During the development of our prototype, we had the idea of creating a distributed network of KBs. The idea is to allow the user to ask a question, and the server queries all the connected KBs. All the connected KBs could possibly have their own vocabulary, and use word vectors to match the relations.

6.4 Interactive NLI

Another idea we did not have time to implement was a more interactive interface. When the user asks a question, and does not get any results back, the system should give some feedback. It could guess what might be wrong or hint at some change to get a better result. To further increase interactivity, and precision and recall, the system should also learn from user input. This requires large amounts of training data but will benefit the system greatly if models can be trained dynamically.

Chapter 7

Conclusion

We have developed a proof of concept remote NLI for querying Linked Data. With correctly structured data and a trained model, the NLI can in theory understand any context of data semantically. As described in Chapter 4, the implementation does not rely on a certain KB, but instead the system learns to understand the specified KB. The system learns to treat the data in a semantic manner, and allows the user to express their questions in natural language. Also, the user does not need any previous knowledge of SPARQL in order to query the data.

We lacked the possibility to explore more deeply the limitations of the system. In a real life scenario, the vast amount of training data necessary might be difficult to acquire. Nevertheless, our proof of concept demonstrates the possibilities of using an NLI in many real industries. We hope that our contributions will help industries who attempt to analyze tons of data, and want to further evolve their business.

Bibliography

[1] "Data - W3C." Available at https://www.w3.org/standards/semanticweb/data. Accessed 2020-02-12.

[2] "W3C - RDF."
Available at https://www.w3.org/2001/sw/wiki/RDF. Accessed 2020-04-23.

[3] "W3C - SPARQL."
Available at https://www.w3.org/standards/semanticweb/query. Accessed 2020-04-23.

[4] "World Wide Web Consortium (W3C)." Available at https://www.w3.org/. Accessed 2020-05-06.

[5] "Semantic Web - Wikipedia." Available at https://en.wikipedia.org/wiki/Semantic_Web. Accessed 2020-02-12.

[6] D. Allemang and J. Hendler, *Semantic Web for the Working Ontologist.* Denise E. M. Penrose, 2007.

[7] L. Ehrlinger and W. Wöß, "Towards a definition of knowledge graphs.," vol. 48, 2016.

[8] "What is a Knowledge Graph? - ontotext." Available at https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-graph/. Accessed 2020-02-13.

[9] M. Färber, F. Bartscherer, C. Menne, and A. Rettinger, "Linked data quality of dbpedia, freebase, opencyc, wikidata, and yago," *Semantic Web*, 2018.

[10] J. Lehmann, R. Isele, M. Jakob, A. Jentzsch, D. Kontokostas, P. N. Mendes, S. Hellmann, M. Morsey, P. Van Kleef, S. Auer, *et al.*, "Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia," *Semantic Web*, 2015.

[11] "About - DBpedia," 2019. Available at `https://wiki.dbpedia.org/about`. Accessed 2020-02-27.

[12] L. Feigenbaum, "SPARQL By Example - W3C." Available at `https://www.w3.org/2009/Talks/0615-qbe/#q13`. Accessed 2020-02-13.

[13] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*, pp. 160–167, 2008.

[14] "POS tags and part of speech tagging - Sketch Engine." Available at `https://www.sketchengine.eu/pos-tags/`. Accessed 2020-02-14.

[15] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," tech. rep., Stanford University, 2014.

[16] "Introduction to pipelines - Stanford CoreNLP." Available at `https://stanfordnlp.github.io/CoreNLP/pipelines.html#annotations-and-annotators`. Accessed 2020-02-27.

[17] J.-D. Kim and K. B. Cohen, "Natural language query processing for sparql generation: A prototype system for snomed ct," in *Proceedings of biolink*, vol. 32, 2013.

[18] M. Dubey, S. Dasgupta, A. Sharma, K. Höffner, and J. Lehmann, "Asknow: A framework for natural language query formalization in sparql," in *European Semantic Web Conference*, Springer, 2016.

[19] D. Damljanovic, M. Agatonovic, and H. Cunningham, “Freya: An interactive way of querying linked data using natural language,” in *Extended Semantic Web Conference*, Springer, 2011.

[20] C. Wohlin, “Guidelines for snowballing in systematic literature studies and a replication in software engineering,” in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, 2014.

[21] S. Keshav, “How to read a paper,” *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 3, 2007.

[22] C. Wang, M. Xiong, Q. Zhou, and Y. Yu, “Panto: A portable natural language interface to ontologies,” in *European Semantic Web Conference*, Springer, 2007.

[23] J. Lehmann and L. Bühmann, “Autosparql: Let users query your knowledge base,” in *Extended semantic web conference*, Springer, 2011.

[24] S. Ferré, “Sparklis: an expressive query builder for sparql endpoints with guidance in natural language,” *Semantic Web*, vol. 8, no. 3, 2017.

[25] N. E. Fuchs and R. Schwitter, “Specifying logic programs in controlled natural language,” *arXiv preprint cmp-lg/9507009*, 1995.

[26] S. Ferré, “Squall: a controlled natural language as expressive as sparql 1.1,” in *International conference on application of natural language to information systems*, Springer, 2013.

[27] D. Rusu, L. Dali, B. Fortuna, M. Grobelnik, and D. Mladenic, “Triplet extraction from sentences,” in *Proceedings of the 10th International Multiconference Information Society-IS*, 2007.

[28] E. Kaufmann, A. Bernstein, and L. Fischer, “Nlp-reduce: A naive but domain-independent natural language interface for querying ontologies,” in *4th European Semantic Web Conference ESWC*, 2007.

[29] A. Taylor, M. Marcus, and B. Santorini, "The penn treebank: an overview," 2003.

[30] B. Muschko, *Gradle in action.* Manning, 2014.

[31] D. Maynard, K. Bontcheva, and I. Augenstein, *Natural language processing for the semantic web*, vol. 6. Morgan & Claypool Publishers, 2016.

[32] L. Zhou, M. Shaikh, and D. Zhang, "Natural language interface to mobile devices," pp. 283–286, 2004.

[33] A. Bernstein, E. Kaufmann, and C. Kaiser, "Querying the semantic web with ginseng: A guided input natural language search engine," 2005.

[34] N. Eisinger and J. Maluszynski, *Reasoning Web: First International Summer School 2005, Msida, Malta, July 25-29, 2005, Revised Lectures*, vol. 3564. Springer, 2005.

[35] "Spring Framework." Available at `https://spring.io/why-spring`. Accessed 2020-05-12.

[36] B. Porter, J. van Zyl, and O. Lamy, "Maven–welcome to apache maven," *Maven–Welcome to Apache Maven*, 2018. Available at `http://maven.apache.org/`. Accessed 2020-05-12.

[37] "Gradle–gradle build tool." Available at `https://gradle.org/`. Accessed 2020-05-12.

[38] P. Webb, D. Syer, J. Long, S. Nicoll, R. Winch, A. Wilkinson, M. Overdijk, C. Dupuis, and S. Deleuze, "Spring boot reference guide," *Part IV. Spring Boot features*, vol. 24, 2013.

[39] D. M. Cer, M.-C. De Marneffe, D. Jurafsky, and C. D. Manning, "Parsing to stanford dependencies: Trade-offs between speed and accuracy.," 2010.

[40] T. Bocklisch, J. Faulkner, N. Pawlowski, and A. Nichol, "Rasa: Open source language understanding and dialogue management," *arXiv preprint arXiv:1712.05181*, 2017.

[41] "Github - Rasa Java-SDK." Available at https://github.com/rbajek/rasa-java-sdk. Accessed 2020-03-25.

[42] H. Pan, "Github - Evaluating OpenNLP." Available at https://github.com/Texera/texera/wiki/Evaluating-OpenNLP. Accessed 2020-03-26.

[43] "Redpill-Linpro." Available at https://www.redpill-linpro.com/. Accessed 2020-03-12.

[44] "ARQ - A SPARQL Processor for Jena." Available at https://jena.apache.org/documentation/query/index.html. Accessed 2020-03-13.

[45] A. Rodriguez, "Restful web services: The basics," *IBM developerWorks*, vol. 33, 2008.

[46] "Understanding memory and time usage - Stanford CoreNLP." Available at https://stanfordnlp.github.io/CoreNLP/memory-time.html. Accessed 2020-02-14.

[47] I. Fette and A. Melnikov, "The websocket protocol," 2011.

[48] "StanfordCoreNLP - Stanford CoreNLP." Available at https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/pipeline/StanfordCoreNLP.html. Accessed 2020-04-21.

[49] "Document - Stanford CoreNLP." Available at https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/simple/Document.html. Accessed 2020-04-21.

[50] "QuestionToStatementTranslator - Stanford CoreNLP." Available at https://javadoc.io/static/edu.stanford.nlp/stanford-corenlp/3.9.2/edu/stanford/nlp/naturalli/QuestionToStatementTranslator.html. Accessed 2020-04-21.

[51] "Open Information Extraction - Stanford CoreNLP."
Available at https://stanfordnlp.github.io/CoreNLP/openie.html. Accessed 2020-04-07.

[52] "SentimentModel - Stanford CoreNLP." Available at https://nlp.stanford.edu/nlp/javadoc/javanlp-3.5.0/edu/stanford/nlp/sentiment/SentimentModel.html. Accessed 2020-04-04.

[53] "RDF Repository Interface."
Available at https://rdf4j.org/javadoc/latest/org/eclipse/rdf4j/repository/Repository.html. Accessed 2020-04-21.

[54] "HTTPRepository."
Available at https://rdf4j.org/javadoc/latest/org/eclipse/rdf4j/repository/http/HTTPRepository.html#HTTPRepository-java.lang.String-. Accessed 2020-04-21.

[55] "RDF RepositoryConnection."
Available at https://rdf4j.org/javadoc/latest/org/eclipse/rdf4j/repository/RepositoryConnection.html. Accessed 2020-04-21.

[56] "CRFClassifier - Stanford CoreNLP."
Available at https://nlp.stanford.edu/nlp/javadoc/javanlp-3.5.0/edu/stanford/nlp/ie/crf/CRFClassifier.html. Accessed 2020-04-07.

[57] A. McCallum and W. Li, "Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons," 2003.

[58] B. Settles, "Biomedical named entity recognition using conditional random fields and rich feature sets," 2004.

[59] J. Rayfield, "A New Hope: The Rise of the Knowledge Graph - Ontotext," 2019. Available at https://www.ontotext.com/blog/the-rise-of-the-knowledge-graph/. Accessed 2020-04-08.

Appendix

A Setting up the Query Service

The project[1] is available open source and can be downloaded and tested. The following appendix will explain how to setup the project.

A.1 Setup GraphDB Repository

1. Go to Ontotext web page, create an account and download GraphDB Free.
2. Start GraphDB Free and make sure the port is set to 7200. To check the port, press the *Settings* button to see the current port.
3. Download the Star Wars RDF data, which we have already trained a model for. We accessed the RDF data from a blog post about Star Wars KGs [59].
4. Make sure GraphDB is up and running, and it should prompt you to `localhost:7200`.
5. Go to *Import-RDF-Create new repository*.
6. Give the Repository an ID, and set the ruleset to *No inference*.
7. Finally, go to *Import-RDF* and *Upload RDF files*, and choose the `data.ttl` file, containing the Star Wars RDF data. Lastly, click on *Import*.

[1] Source Code, https://github.com/acke80/StarQ

A.2 Setup Application Properties

In the project go to `resources/application.properties`. Change `repoURL` so the last part matches the name of your repository in GraphDB. All the other properties are set by default to work with the Star Wars repository.

A.3 Setup VM Options

The program needs at least 4GB of memory to run. This is due to the amount of CoreNLP annotators loaded into the pipeline. We added the following Java VM options: `-Xmx4g -Xms4g`. The memory usage may change as CoreNLP gets frequent new updates. At the time of writing we recommend at least 4GB of memory, but in the future more or less might be needed.

B Using the NLI

To use the Natural Language Interface, start the program and wait for it to load the properties and start the server. Go to `localhost:8080` and you should be prompted to the web page. If everything is working, the web page should say Connected.
The model has not trained on large amounts of data, so it will not recognize many Star Wars entities. We used the movie script from the first Star Wars movie: *A New Hope* to train the model, which means the NLP will only recognize certain entities that appear frequently in the first movie, for example: Darth Vader, Tatooine, and Han Solo.

B.1 Example Queries

In the input field, questions does not need to end with a question mark if only asking one question. Although, when asking multiple questions at the same time, separation with a question mark is necessary.

- What height is Vader?

- What is the orbital period of all the planets?
- Who are the residents of Tatooine?
- What is the description of all the characters?
- What are the films with Han Solo?
- What eye color does Luke have?
- What is the terrain on Alderaan?
- What is the cost in credits for all the starships?