FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Automated extraction of personal profiles from a university domain using web scraping and NLP methods
Student:	Bc. Tomáš Lenoch
Supervisor:	Ing. Stanislav Kuznetsov
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2023/2024

Instructions

The aim of the work is to design and implement a software application that utilises web scraping and NLP methods for extracting personal information, including working positions and affiliations, of university employees from the university domain website. The quality of the extracted profiles may decline due to an insufficient amount of information on the website. Therefore, the application must provide users with the capability to verify and adjust the extracted profiles manually.

Instructions for elaboration:

1. Conduct thorough research on web scraping techniques and NLP methods for information extraction.
2. Identify the sources of information on the university website that contain personal information, work position, and affiliations of university employees.
3. Develop and implement a web scraping system to extract relevant information from the university website with the help of NLP pre-trained models for data pre-processing and analysis.
4. Integrate the web scraping and NLP components into a comprehensive software application.
5. Develop user-friendly interfaces for the software application.
6. Test the software on a sample set of university websites to verify its effectiveness.
7. Document everything properly.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 18 February 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Reference:
1. Lan, Man & Yu, Zhe & Zhang, & Lu, Yue & Jian, Su & Lim, Chew. (2009). Which who are they? people attribute extraction and disambiguation in web search results.
2. T., István & Farkas, Han, Xianpei & Zhao, Jun. (2009). CASIANED: People Attribute Extraction based on Information Extraction.
3. Richárd & Jelasity, Márk. (2009). Researcher affiliation extraction from homepages.

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 18 February 2023 in Prague.

Master's thesis

AUTOMATED EXTRACTION OF PERSONAL PROFILES FROM A UNIVERSITY DOMAIN USING WEB SCRAPING AND NLP METHODS

Bc. Tomáš Lenoch

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Stanislav Kuznetsov
May 3, 2023

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Lenoch Tomáš. *Automated extraction of personal profiles from a university domain using web scraping and NLP methods.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Contents

List of Figures

List of Tables

List of code listings

First of all, I would like to thank my master's thesis supervisor Ing. Stanislav Kuznetsov, for his valuable comments and substantial advice. Furthermore, I would like to thank all my family and those who supported me while writing this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 3, 2023 .....

Abstract

This thesis deals with the development of a software application that can automatically extract personal profiles of employees from university websites, using web scraping and natural language processing (NLP) techniques. The extracted profiles include affiliations of the employees towards organizational units within the university. In addition, a user-friendly graphical interface is provided in the application to verify and modify the extracted profiles. The component-based design of the application allows for future adjustments to handle a more specific set of universities. The performance of the application is evaluated on the set of manually scraped university websites. The evaluation results suggest that the application can perform the required tasks. Further testing is required in the future due to the limited size of the reference set.

Keywords automated web scraping, personal attribute extraction, automated personal profile extraction, university employees

Abstrakt

Táto práca sa zaoberá vývojom softvérovej aplikácie, ktorá dokáže automaticky získavať personálne profily zamestnancov z webových stránok univerzít pomocou techník web scrapingu a spracovania prirodzeného jazyka (NLP). Získané profily obsahujú príslušnosť zamestnancov k organizačným jednotkám v rámci univerzity. Okrem toho je v aplikácii k dispozícii užívateľsky prívetivé grafické rozhranie na overovanie a úpravu získaných profilov. Dizajn aplikácie založený na komponentoch umožňuje do budúcnosti úpravy na spracovanie špecifickejšieho okruhu univerzít. Výkonnosť aplikácie sa hodnotí na množine dát ručne extrahovaných z univerzitných webových stránok. Výsledky hodnotenia naznačujú, že aplikácia dokáže vykonávať požadované úlohy. V budúcnosti je ale potrebné ďalšie testovanie vzhľadom na obmedzenú veľkosť referenčnej množiny.

Klíčová slova automatizovaný web scraping, extrakcia osobných atribútov, automatizovaná extrakcia personálnych profilov, univerzitní zamestnanci

Introduction

Universities have always played an important role in society by providing students with comprehensive and structured education, conducting research to advance knowledge, and contributing to economic and social development. To effectively fulfill these roles, the university is divided into smaller divisions, each with its responsibilities. This means that the organizational structures of universities are usually very complex.

Knowledge of this organizational structure can be a valuable resource for businesses looking to establish partnerships, identify potential talent, and develop new research and development projects. It can help them identify departments with specialized expertise that can improve their product and services. Moreover, knowing the exact personnel hierarchy of the organization of interest will also help businesses to identify potential candidates for recruitment or collaboration.

Unfortunately, despite the significant benefits of understanding the organizational structures, there is currently no standardized method for extracting and storing this information in a universally accessible format. Therefore, this thesis aims to develop a software application that can extract this structure and export it in an easily readable format. We aim to solve this problem by developing a software solution based on scraping websites of university domains with the help of natural language processing methods.

The thesis structure is divided into a theoretical and a practical part. In the theoretical part, we explore formal requirements that our application must satisfy. We also aim to describe current approaches and methods in the areas of web scraping and natural language processing. In the last chapter of the theoretical part, we review related works in this area. In the practical part, we address the design of the application and its components. We also evaluate the performance of the application on a set of university websites.

Part I

Theoretical

Chapter 1

Problem definition

The main objective of this thesis is to develop a software application that uses web scraping and natural language processing (NLP) to extract personal profiles of university employees from its domain websites. Each profile must include the person's full name and their work affiliation with the university and its parts. Additionally, it can include contact information, such as email or an office address.

Universities are organized into several sub-organizational parts, such as faculties, institutes, and departments. Each sub-organization typically has different objectives and areas of expertise. The main university website typically provides information about the organizational structure of the university, including the web links to sub-organizations websites. Each sub-organization website typically contains information about its activities, research interests, organizational structure, and employees. From these websites, it is then possible to extract this information using web scraping methods. With the use of NLP methods, the extracted information can be then transformed into valid personal profiles.

To ensure the universal functionality of the developed application, it must be capable of handling any university domain that offers an English version of its website. To achieve this, the application must meet the following functional requirements:

1. Transform any scraped website to its internal format, which allows the application to analyze and extract the required data in a structured way.

2. Generate a hierarchical tree that represents the organizational structure of the examined university This tree will provide a more profound understanding of the university's organizational structure.

3. Extract personal profiles from organizational units using NLP methods.

4. Provide a graphical interface that allows the users to confirm and edit the extracted data. Due to the high variability and lack of reference data, automatic verification of the extracted data is not feasible. Hence, the graphical interface allows the user to verify and correct any inaccuracies.

In essence, to solve this problem, we need to construct a knowledge base for the input university. The construction of this knowledge base includes solving subtasks of Information Extraction task (IE) such as Part of Speech Tagging, Named Entity Recognition and Relationship Extraction.

1.1 Data model

In light of the functional requirements presented earlier, we need to define a data model that will be able to represent the extracted data from university websites in a structured and machine-readable format. To achieve this goal, we defined a data model consisting of three entities: `Organization`, `Person` and `Affiliation`. The entity-relationship diagram of this model can be seen in Figure 1.1.

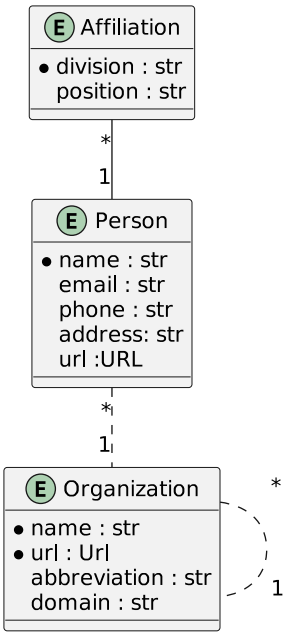

Figure 1.1 Entity diagram

Each `Organization` entity represents either the university itself as the main organization or as another sub-organizational unit within the university. It contains attributes such as the organization's name, domain, and URL of its website.

Each `Person` entity represents an employee of the university. It contains attributes such as the person's name, email, phone number, and website URL from which their data was extracted. Each `Person` entity is associated with a single `Organization` entity, which represents the organizational unit with which the person is associated with. Finally, each `Person` entity can have multiple `Affiliation` entities associated with it. The `Affiliation` entity has division and position attributes. The division attribute represents the organizational unit's name, and the position attribute defines the person's role within that unit. Every `Person` entity has an implicit `Affiliation` associated with it, generated from the `Organization` it belongs to. Additionally, other implicit `Affiliation` entities can be added recursively through parents of this `Organization`.

1.2 Hierachical tree

With the data model defined in the previous section, we can now store the organizational structure of a university as a hierarchical graph tree H. The tree nodes are instances of `Organization` entities. Since these instances can have associated `Person` instances, H can represent the complete knowledge base of a university. Therefore, H is the only output parameter of the application.

The input to the application is an `Organization` instance, which holds details about the target university. This instance will become the root of the H. In order to precisely locate a specific node in the tree, we will refer to i-th node at j-th level as $\mathrm{ORG}_{i,j}$. The level of a node is determined by its distance from the root of the tree. This implies that the root node is denoted as $\mathrm{ORG}_{0,0}$. Employing this notation, any node in the tree can be easily identified by specifying its level and position within that level. For example, an instance in the third node at the second level of the tree can be referred to as $\mathrm{ORG}_{2,1}$. Moreover, we can denote all nodes at level j as $ORG_{:,j}$.

The tree is iteratively constructed by the procedure `BUILD_TREE` shown in Algorithm 1. A queue-based breadth-first search approach is used to iteratively explore the organizational structure, extracting child organizations and adding them to the tree. Once all the founded organizations are extracted, personal profiles are extracted from all nodes of H.

Algorithm 1 Construction of output tree H

```
1:  procedure BUILD_TREE(university : Organization)
2:      H ← tree({university}, {})
3:      Q ← Queue({university})
4:      while ¬Q.empty() do
5:          active_node ← Q.pop()
6:          children ← scrape_and_extract_organizations(active_node)
7:          if ¬children.empty() then
8:              Q.insert(children)
9:              H.add_edges(active_node, children)
10:     for each org ∈ H.nodes() do
11:         org.people ← scrape_and_extract_people(org)
12:     return H
```

Methods on the lines 6 and 11 have similar functionality. They take `Organization` as an input parameter and return a collection of `Organization` or `Person` instances, respectively. They represent two pivotal processes in the application workflow.

Chapter 2

Information extraction

One of the first things we notice when looking at any data set is that it is either structured or unstructured. Structured data is typically organized in a specific format, such as a relational database. In contrast, unstructured data lacks an obvious structure and is often not easily readable by computers. However, even seemingly unstructured data, such as human-written text, may contain some structural elements, such as headings, paragraphs, or footnotes. Therefore, we can view this type of data as semi-structured. This is particularly evident when analyzing textual data found on websites, where data is often presented in its own unique structure [1].

The main objective of the IE task is to identify entities and relationships that are semantically defined within unstructured or semi-structured textual data. The extracted information must be saved in a format that will facilitate subsequent analysis and processing. While small data sets can be processed by humans, the typical size of meaningful data sets today is beyond the scope of manual processing [2].

Information extraction itself belongs to the area of Natural Language Processing. To properly describe the necessary sub-tasks of IE, we must explore some basic terms and techniques in this area.

2.1 Natural language processing

Using computational techniques to analyze linguistic data, such as documents or websites, is called Natural Language Processing (NLP). The main goal of NLP is to use insights from linguistics to build a structured representation of the natural language. This representation can be focused either on syntactic or semantic features of the text. Typically, an NLP system comprises a pipeline of components, where each component transforms input text in a more structured way. The initial components within the NLP pipeline address tasks that refer to the low-level text characteristics. In contrast, subsequent components are designed to analyze higher-level concepts and relationships. The components use different techniques to process the text. These techniques include rule-based methods like regular expressions and finite state automata, as well as machine learning and statistical models. [3]

We need to introduce some basic terminology to describe the functionality of each pipeline component. Before processing, the input text must be segmented into linguistic units, such as words, punctuation, numbers, or alphanumerics. These units are commonly referred to as *tokens* [4]. The process of splitting input text into tokens is called *tokenization*. This process may involve additional steps, such as converting all letters to lowercase and eliminating stop words, which are commonly used words like *the* and *a*. [5].

Based on this, we can define subsequent terms:

- **Sentence** – Ordered sequence of tokens.
- **Corpus** – Set of tokens or sentences.
- **Document** – Collection of sentences.
- **Dictionary** – Set of tokens in the corpus.

2.2 Part of Speech Tagging

Every language is constructed of its own vocabulary and grammar rules. The grammar rules dictate how words can be combined to form meaningful sentences. Each word plays a role in conveying the meaning of the sentence. In most European languages, we can label these roles as nouns, verbs, pronouns, prepositions, adverbs, conjunctions, participles, and articles. The task of classifying each word into one of these categories is called part-of-speech (POS) tagging. Besides the basic categories, additional categories may be added depending on the problem being solved.

According to [5], the classification of a word in a POS class is based on its grammatical relationship with neighboring words or the morphological properties of their affixes. POS classes can generally be divided into two main categories – closed and open. Classes in the closed category are those with relatively fixed membership, such as prepositions, articles, and conjunctions. The open category includes classes like nouns, verbs, adjectives, and adverbs, which are typically content words. This thesis will examine mostly nouns and noun phrases, as they usually refer to entities we seek, such as persons, organizations, and locations.

The POS tagging process consists of assigning POS class y_i to each x_i token of sequence $x = x_1, x_2, \ldots, x_{n-1}, x_n$ of length n and where $x_1, x_2, \ldots, x_{n-1}, x_n$ are word tokens. An example of a tagged sentence is shown on Figure 2.1.

Peter(**PROPN**) used(**VERB**) to(**PART**) work(**VERB**) for(**ADP**) Alfred(**PROPN**).

Figure 2.1 Example of POS tagging on sequence with length $n = 6$

Words can have ambiguous meanings, and the POS class assigned to them is determined by the grammatical and morphological structure of the sentence. Because of this, depending on the context in which it is used, the word may be assigned to a various POS classes. However, several words can be easily disambiguated based on the likelihood of their various tags. For instance, the word ”book” may refer to a physical object or a written work, but the last mentioned sense is significantly more prevalent. Given this, the common strategy is to choose the most common class for the ambiguous word from the training set.[5]

Once a sentence is POS tagged, we acquire insights into its grammatical structure, thereby allowing us to identify and extract all proper nouns from it. In order to recognize which semantic entity they refer to, we need to solve another sub-task of IE, called Named Entity Recognition.

2.3 Named Entity Recognition

Peter **(PERSON)** used to work for Alfred **(PERSON).**

Figure 2.2 Example of NER task

The main goal of the Named Entity Recognition (NER) task is to identify named entities like person names, organizations, or locations from the nouns and noun phrases in the sentence. For example, if we look at the sentence in Figure 2.2, we can see that two people entities were

recognized: ”Peter” and ”Alfred”. These recognized entities are useful in many other tasks besides IE, such as question-answering and sentiment analysis.

One of the major problems in the NER task is determining boundaries for named entities. Another problem is choosing the correct label for the recognized entity. In the Figure 2.3, we can see an example where the same word refers to a different named entity. [5]

I called my brother Cleveland**(PERSON)**
Cleveland**(GPE)** is in Ohio**(GPE)**.

Figure 2.3 Example of NER entity ambiguity: In the first sentence, Cleveland is recognized as a person entity and, in the second, as a geopolitical entity.

There are generally three main approaches to creating a NER solving system: rule-based, machine learning, and hybrid.

The rule-based approach is characterized by a collection of rules that experts have manually crafted. These systems are usually used in specific domains where the knowledge of the experts is highly beneficial. These systems are expensive to develop, not portable, and not applicable in a different domain. NER systems based on machine learning have an advantage over rule-based systems, as they can include more complex patterns and complex decision-making algorithms by learning from real-world data. As with any machine learning system, we divided them by type of training data into three main categories, which are supervised, semi-supervised, and unsupervised. The hybrid-based approach combines the best parts of the approaches mentioned above. These systems are more flexible and accurate than those that use only one approach. [6]

2.4 Relationship Extraction

Once named entities (NE) have been identified and extracted from text, various predefined relationships between them can be detected. Identifying these relationships is the primary objective of the Relation Extraction (RE) task. The RE task can be divided into two steps. The first step consists of identifying whether a relationship exists between the two entities. When this is determined, the relationship is classified into a predefined category in the second step. [7]

In long documents, relationships between entities are typically identified based on the relationships between sentences. However, recognizing these relationships can be challenging because we work with relatively short and unconnected sentences. Fortunately, we can make use of hyperlink connections between the web pages to extract relationships between entities throughout the entire website. This approach allows for the identification of relationships that would otherwise be missed when analyzing individual sentences.

2.5 Statistical Language Models

The current trend is to use statistical language models to solve NLP tasks, including IE, as they can effectively handle a wide range of tasks in this area. The main principle of these models is to assign a probability to every possible sequence of tokens from the corpus they have been trained on. Therefore, if we feed a language model a sentence, it will provide a probability for each token in the sentence. The probability is based on the context in which the token is used. [8]

The context of a token is defined as a set of its neighbor tokens. In the Figure 2.4, we can see that context of size 2 for the token ”Cleveland” in the example sentence contains tokens ”my” and ”brother”. At its core, the context of the token represents the history of its preceding tokens. Ideally, we want this history to be as long as possible.

I called my brother **Cleveland**.

Figure 2.4 context of size 2 for the token "Cleveland"

2.5.1 n-gram Models

There are different approaches to building statistical language models for NLP tasks, the simplest ones use n-grams to calculate the probability of tokens in a sentence. An n-gram is a sequence of n tokens, where $n = 1$ or $n = 2$ are referred to as uni-gram and bi-gram, respectively. The value of n is a hyperparameter of this group of models, which is set before the training process begins. [5]

During the training of n-gram models for NLP tasks, the goal is to estimate the probability of a token in a sentence based on its context, which consists of the $n - 1$ preceding tokens. The token and its context together form an n-gram in the model. The probability of each token is then determined by the frequency of all the different n-grams in the training corpus that contain it as the last token. A limitation of n-gram models is that they struggle to capture long-term dependencies between tokens when the value of n is small. If we increase the value of n to capture these dependencies, then we require a much larger training corpus. Another drawback of these models is that they cannot predict a token not included in the training corpus. [8].

2.5.2 Neural Language Models

In recent years, models based on neural networks have gained popularity in the area of NLP. These models can learn complex patterns and relationships within large amounts of textual data, making them highly effective for various NLP tasks. A specific type of statistical language model that uses neural networks to approximate the probability distribution of tokens in a sentence is called a neural language model.

These models can handle longer histories and generalize better over contexts of similar tokens compared to n-gram models. They also have a more accurate prediction of the next token. On the contrary, they are more complex, and therefore they are less interpretable. Additionally, they require much larger training datasets to work accurately. [5]

By [9], the neural language model based on feed-forward neural network architecture comprises an input embedding layer, multiple hidden layers, and an output layer that returns a probability distribution. The input embedding layer takes input data of a sequence of tokens and converts each token into a fixed-size vector representation called word embedding. The meaning of the words and their relationships with each other are captured by word embeddings, which allow semantically similar words to be located close by in the vector space. In the hidden layers, the neural network processes the input embeddings and learns to identify patterns in the training sequences. Each hidden layer consists of multiple neurons, and each neuron is responsible for learning a specific feature or pattern in the input sequence. These features could be as simple as identifying common prefixes or suffixes in words, or as complex as recognizing grammatical structures and semantic relationships between the words. The output layer produces a probability distribution over the training corpus vocabulary. Each neuron in the output layer corresponds to a word in the vocabulary, and its value represents the probability that the corresponding word is the next in the sentence. This probability is calculated by applying a softmax function to the output of the last hidden layer.

2.5.3 Word Embeddings

Using word embeddings enables neural language models to generalize better than n-gram models. This implies that encoding tokens not present in the training corpus into a word embedding

makes it possible to predict their probability. Formally, word embedding (or word vector) e_w of word w is a dense vector in a d-dimensional vector space, where d is significantly smaller than the vocabulary size. The typical value of d used is between 50 to 1000. Each element of the embedding vector e_w represents a learned feature of the word w that captures its meaning and context within the training corpus. [5]

In the embedding layer, neural language models can either learn embeddings in the training process or use pretrained static word embeddings generated from large corpuses containing millions of words. The most common methods for generating static word embeddings include word2vec [10], GloVe [11], or fastText [12].

2.5.4 Recurrent Neural Language Models

One of the primary limitations of feed-forward neural language models is their inability to handle inputs with variable-length. Consequently, they struggle to capture long-range dependencies in text, which is crucial for NER task. In order to recognize named entities in text, the model must understand the context and relationships between different words and phrases. Furthermore, named entities differ in length and the surrounding context words they occur with.

To address these challenges, models based on recurrent neural networks (RNN) are employed. Unlike feed-forward models that take a fixed-size input and produce a fixed-size output, RNN models can handle variable-length inputs and produce variable-length outputs. They can capture long-range dependencies in text. The model can maintain a memory of previous inputs through the use of recurrent connections. These connections allow information to flow from one time step to the next. Because of this, they can capture the context and relationships between different words and phrases over longer distances than a feed-forward model.

However, as described in [13], simple RNN models can only use information up to 5-10 time steps into the past because of their gradient-based training process. When gradients are back-propagated, their values decrease over each time step until they eventually vanish. This problem is known as the vanishing gradient problem, which limits the ability of the model to update weights effectively. To overcome this issue, an extension of RNN, named Long Short-Term Memory (LSTM) [14], has to be used.

LSTM was specifically designed to address this problem by introducing specialized gated neural units that can selectively remember or forget previous context over time. These units are composed of 3 types of control gates: forget gate, input gate, and output gate. The forget gate regulates how much of the previously remembered context should be forgotten. The input gate controls how much of the input coming to the unit should be stored in its context. Lastly, the output gate regulates the amount of the stored context that is sent as the output from the unit.

Although LSTM models have achieved great success in many NLP tasks, they still have some limitations. One of the main drawbacks of RNN networks, including LSTMs, is their poor suitability for parallel training due to their sequential processing of input tokens. To overcome this issue and take advantage of parallel computing, a new neural network architecture called Transformer [15] was introduced.

2.5.5 Transformers

Transformer is a type of neural network architecture that does not rely on recurrent connections, such as LTSM. The reason for this is that all inputs are passed to the network simultaneously, meaning there is no time-step concept for the inputs. This characteristic allows for the parallelization of its training process, which results in faster computation times even when dealing with large datasets. It is an encoder-decoder neural network, sometimes referred to as a sequence-to-sequence network. Therefore, it consists of three main components. An encoder, which is a neural network that encodes an input sequence of symbols $(x_1, \ldots, x_n)$ into a sequence of contextualized representations $z = (z_1, \ldots, z_n)$. These representations z are then passed to a decoder

that generates an output sequence of symbols $y = (y_1, \ldots, y_m)$. In language modeling, the input symbols are tokens, and the output symbols are probabilities of each token in the vocabulary, forming a distribution. [15]

The encoder and decoder are composed of stacks of transformer blocks, as shown in Figure 2.5. These blocks are made of linear layers, feed-forward networks, and self-attention layers. Self-attention is a pivotal concept in transformers that allows them to extract and utilize information from arbitrarily large contexts without the need for intermediate recurrent connections. In simpler terms, it allows the network to calculate the relevance of the i-th token in the input sequence to the other tokens in the sequence. These relationships are then stored in attention vectors.

However, when working with long input sequences, a single attention vector struggles to capture all the important relationships between the tokens. To resolve this issue, transformers use multi-head attention layers, which consist of multiple self-attention networks with different parameters, known as heads. Depending on its parameters, each head captures the relationships between the tokens differently, and thus it outputs a specific attention vector. These distinct output vectors are then combined into a final attention vector for the sequence, enabling transformers to capture diverse relationships between tokens.

Since the input sequence is not processed sequentially, the transformer does not have information about the absolute or relative positions of the tokens in the input. The solution to this problem is to generate separate embeddings of the same dimension as the input embeddings that capture the positions of the tokens in the sentence. By adding these positional embeddings to the input embeddings. An augmented representation, which incorporates both the semantic meaning of the token and its position in the sequence, is created.

The success of transformers can be attributed to their ability to parallelize training, process large datasets, capture various relationships between tokens, and retain positional information. With the introduction of pretrained transformer-based models, such as BERT [16] and RoBERTa [17], which have demonstrated state-of-the-art performance on a variety of NLP tasks, including POS tagging and NER. Transformers have become the preferred option for language modeling and related NLP tasks.

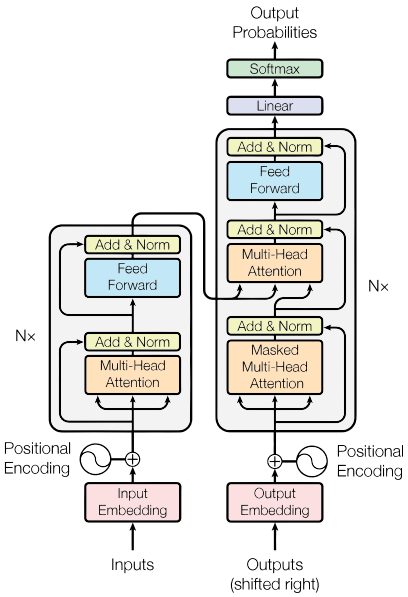

Figure 2.5 Transformer architecture [15]

Chapter 3

Web Scraping

The internet is a great source of information, but it can be a daunting task to extract and analyze data without the right tools and techniques. As more businesses and organizations seek to produce data-driven products, web scraping has become an increasingly important tool.

Internet websites can present data that is only accessible through a web browser and do not provide an option to save a personal copy of this information. In order to make this presented data more accessible, web scraping techniques are employed. Web scraping is an automatic process that extracts meaningful data from the HTML source code of an Internet website. The extracted data is then stored in a central local database or a spreadsheet [18].

In everyday conversation, the terms ”website” and ”web page” are often used interchangeably. In the context of web scraping, it is crucial to distinguish between the two. A web page is a single document that can be accessed online through a unique URL. In contrast, a website is a collection of related web pages whose URLs belong to the same web domain. Each web page is designed to display information in a specific format and may contain hyperlinks to other web pages, multimedia elements, and other interactive features.

Hypertext Transfer Protocol (HTTP) is a communication protocol that enables client applications to communicate with web servers. As defined in [19], the first phase of the web scraping process, called the fetching phase, consists of obtaining the source code of a web page. During this phase, the web scraping client sends an HTTP request to the web server and receives an HTML response in return. In the second extraction phase, requested data is extracted using libraries for HTML parsing, regular expressions, or XPath queries. XPath is an expression language used to navigate and select elements within an XML or HTML document [20]. In the final transformation phrase, extracted data is transformed into a more structured form suitable for storage and presentation.

3.1 Web Scraping Tools

A working web scraping program requires proper technology to execute all three web scraping phases. A common approach is to use multiple programming libraries that would help fulfill each phase. Ideally, these libraries should be written in the same programming language.

There are numerous programming languages available for implementing web scraping programs with the use of libraries. One of the most popular options is Python, which we will use to implement the application for this thesis. It natively supports communication through the HTTP protocol. Additionally, it offers HTML parsing libraries, such as *BeautifulSoup* [21] and *lxml* [22]. Moreover, another approach is to use more high-level frameworks that offer a more comprehensive and efficient approach to web scraping. One of these frameworks is *Scrapy* [23],

which is a fast Python web scraping framework that extracts structured data from websites. It is based on spiders, which are classes that define how to crawl and how to extract data from websites. They are designed to be flexible and customizable, allowing developers to handle a broad range of scraping scenarios. Additionally, *Scrapy* has a powerful item pipeline system that allows processing and manipulation of the scraped data.

3.2 Web Page Representation

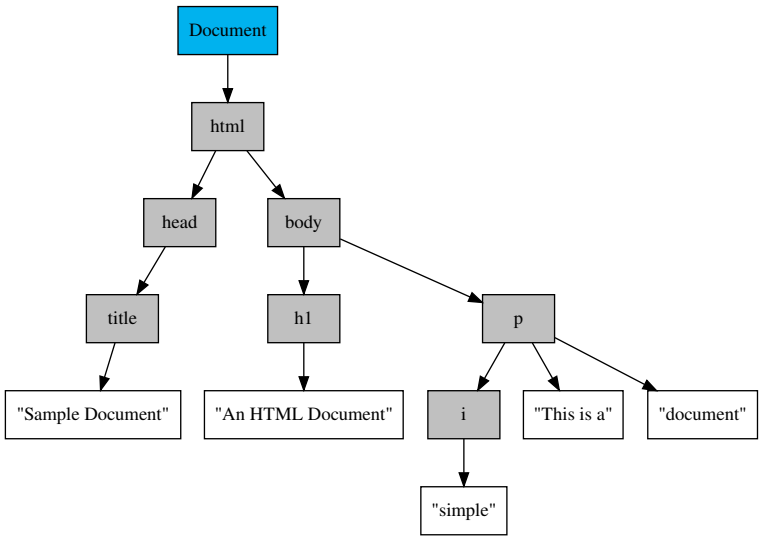

Figure 3.1 Example of the DOM tree [24].

When retrieving the source HTML document via HTTP protocol during the initial fetching phase, it is necessary to parse it into a format that a web scraping program can easily process. The parsing process involves analyzing the HTML source code and transforming it into a tree-like data structure that a web scraper program can easily navigate and access. This data structure represents the hierarchical structure of the HTML document, with each element and attribute mapped to its corresponding node in the tree. After parsing is complete, the web scraping program can then traverse the tree and extract the desired data for further processing or analysis.

3.2.1 DOM Tree

One of the most popular data structures for the representation of HTML documents is Document Object Model (DOM). The DOM represents the document as a tree-like structure called the DOM tree. Nodes in this tree represent each element, attribute, and piece of text in an HTML document. [25]

There are 3 main types of nodes: document, element, and textual. The document node is the root of the DOM tree and represents the entire document. Element nodes represent HTML tag elements such as `<html>`, `<body>`, `<div>`, and so on. These nodes represent the structural components of the HTML document and contain information about the element's tag name and attributes. The textual content of the tag elements is represented by textual nodes. Furthermore, more specific types of nodes can be used to represent the document structure in more detail. [24]

Libraries like *BeautifulSoup* and *lxml* provide a convenient way to automatically construct the DOM tree from HTML source and provide a flexible interface to navigate, manipulate, and modify the nodes of the tree. However, in some cases, manual construction of the DOM tree may

be necessary. This could happen, for example, if the desired structure of the tree is different from the one generated by the automatic parsing. In such a case, the DOM tree must be manually constructed by creating and linking nodes together into the desired hierarchy.

3.2.2 Semantic Tree

The goal of our thesis is to create a web scraper that can be used universally to scrape any given university website. To achieve this, we require a representation that captures the key features of a document regardless of its specific designed structure. We have found that the DOM tree provides too low-level representation. Therefore, we aim to use a representation that encapsulates high-level features of the document more abstractly. One such solution for high-level representation is described in [26]. Most websites on the internet are generated from templates, which ensures that the semantic schema of the website is consistent across all pages. The only thing that changes is the content of the individual pages. In this paper, authors propose an algorithm that automatically detects semantic structures in HTML documents. Their approach is based on the observation that page elements with semantic similarities exhibit spatial locality in the rendered view of the web page. They capture the spatial locality of the elements as a similarity of root-to-leaf paths in the DOM tree. The principle of the algorithm is based on restructuring the DOM tree by traversing it in a bottom-up manner and grouping nodes with similar paths.

Algorithm 2 Construction of the semantic tree [26]

```
1:  procedure PartitionTree(n : node of the DOM tree)
2:      if n is a leaf node then
3:          n.path ← path from leaf to root
4:      else if n has only one child node c then
5:          PartitionTree(c)
6:          group n and c to the new compound node m
7:          m.path ← c.path
8:          replace n with m and remove n, c from the tree
9:      else
10:         for each child node c of n do
11:             PartitionTree(c)
12:         FindPartition(n)
```

The Algorithm 2 reconstructs a DOM tree into a semantic tree by grouping nodes with similar paths in a bottom-up manner. It first traverses the tree from the root to the leaves using recursion. If a leaf node is reached, its path from the leaf to the root is stored. If a node has only one child, the algorithm proceeds to the child and groups the two nodes into a new compound node that has the same path as the child node. In case when a node has more than one child, the algorithm recursively processes each child and then calls the `FindPartition` procedure to group the children into compound nodes based on their path similarity.

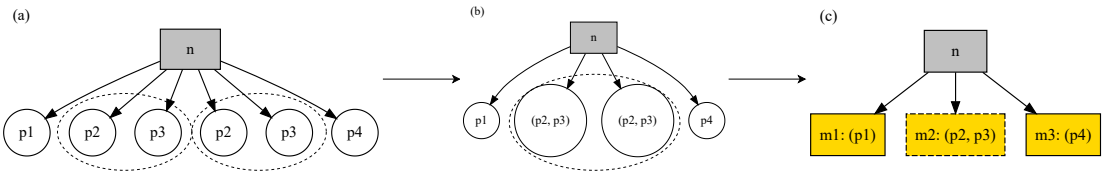

Figure 3.2 Visualization of the algorithm for grouping nodes in `FindPartition`

To group nodes, the algorithm of the procedure converts the paths of all nodes into a sequence of strings, respecting the order of the nodes. For example, the string sequence α for tree (a) in Figure 3.2 is $p1.p2.p3.p2.p3.p4$. The algorithm then identifies patterns in this string sequence by finding the maximal repeating substring. By the paper's definition, a substring γ is a maximal repeating substring in a string β if it appears at least twice in β, its length multiplied by the number of times it appears is at least half of the length of β, and it is the longest substring that satisfies these conditions. In the example, the pattern for α is $p2.p3$.

The algorithm then groups nodes with paths that match the pattern into a new compound node, with the pattern as its path value. In tree (b) of Figure 3.2, nodes with paths satisfying the pattern are grouped into a compound node with new path $(p2, p3)$. The algorithm repeats this process until no more patterns can be found. The final output is the tree (c), with compound nodes $m1$, $m2$, and $m3$. Note that $m2$ is composed of multiple nodes, while $m1$ and $m3$ are compound nodes with only one node each.

Chapter 4

Related works

The focus of this chapter is to examine related works that are relevant to our task. The most similar to our task is the Attribute Extraction (AE) task, which involves identifying and extracting specific attribute values associated with entities from unstructured textual data. In our case, we are interested in the personal AE task, which involves extracting personal attributes. To solve AE, many works were concluded in the third Web People Search campaign (WePS-3) [27] in which participating teams had to resolve person name disambiguation problem and AE for persons sharing the same name.

The solution from [28] achieved the best results in AE task. The authors developed a system that handles both problems based on biographical attribute extraction. They adopted the premise that relevant data for personal AE can be obtained from both textual parts and tables of websites while removing undesired components such as navigation menus. In addition, short text paragraphs with less than 60 characters and less than two verbs were removed. The authors trained their own NER model to extract attributes, but the accuracy score was low. To improve accuracy, they developed attribute-specific section selection modules and used simple string matching to create a database of positive and negative paragraphs for each attribute. They then extracted attributes from paragraphs containing at least one word from a set of positive words created from the positive paragraphs. The attribute extraction system described in the work has two main components: a candidate attribute extraction module and an attribute verification module. The approach involves marking potential attribute values in a paragraph and then identifying which of these candidate values have been found. They also grouped similar attributes into logical categories. This allows for a consistent approach to extracting attributes of the same type and assumes subordinate relationships among related attributes.

In a system called CASIANED, presented in [29], authors developed a personal AE system based on IE. The system extracts attributes from web pages. It consists of two function modules: the attribute candidate generation module and the attribute candidate verification module. The attribute candidate generation module identifies possible attributes for every attribute class on a web page by recognizing named entities and noun phrases. Similarly to [28], attribute classes are split into three categories based on named entity type. The candidate verification module then verifies these candidates through classification. Authors mention that commonly used NER systems lack performance when dealing with irregular text on web pages. Therefore, it is difficult to establish precise boundaries for attribute values. To address this issue, they use gazetteers generated from a prefilled knowledge base to recognize more complicated named entities. Furthermore, they state that it is a challenging task to develop an AE system that performs well on different kinds of websites. This is because they are usually noisy, irregular, and multi-topical.

The authors of [30] have introduced a proficient approach for extracting diverse categories of

target person attributes from unprocessed web pages. The extraction process involves the use of several techniques, such as NER, regular expression patterns, gazetteer-based matching, and manually constructed rules based on web page cleaning. The authors have implemented two web page cleaning algorithms – shallow and deep – which differ in the extent to which irrelevant content is removed from the web page. The evaluation of the results has shown that deep web cleaning enhances the precision of the extracted attributes, but it decreases the overall recall of the results.

The ArnetMiner [31] system aims to extract and mine academic and social networks. One of its parts automatically extracts researcher profiles from the web using a unified approach based on Conditional Random Fields. This approach lies in three steps. In the first step, the homepage of the researcher is found through analysis of web search results using SVM binary classifier. The second step consists of tokenization of the web page text and heuristically tagging each token. Their tokenization algorithm defines 5 types of tokens: standard word, special word, image, term, and punctuation mark. Special word tokens are assigned tags, such as Position, Affiliation, or Email, during the tagging process.

In the paper [32] authors have directed their attention towards the extraction of researcher affiliations from their respective home web pages, rather than focusing on personal AE in general. Their findings indicate that many of these home pages present their content in text-like form instead of more structured. They apply filtering method based on string-matching to remove irrelevant content from the web page. For the extraction process they opted to use entity-based approach that uses custom NER system with narrow selection of NE entity classes that are associated with university affiliations.

There is a range of techniques used by the authors to extract personal attributes, including NER, gazetteers, regular expression patterns and manually constructed rules based on web page cleaning. In addition, methods such as attribute-specific section selection and simple string matching are developed to improve accuracy. The evaluation of these works reveals challenges in developing AE systems that can perform well across different types of websites.

Part II

Practical

Chapter 5
Design

This chapter will explore the software design of our application. It will provide an overview of the architecture, the components involved, and how they interact with each other to build a hierarchical tree of an input university.

5.1 Technologies

We choose Python as the main technology for the implementation of our application because it offers all the necessary libraries for the purpose of this thesis.

For web scraping tasks, we use a combination of *Scrapy* and *BeautifulSoup* libraries. Furthermore, we use *spaCy* [33], which is a free, open-source Python library for NLP. It offers fast and efficient tools for tokenization, POS tagging, NER and other related tasks. We use *spaCy* with a pretrained transformer model. To be able to work with graphs, we use *Networkx* [34], which is a library for the creation and manipulation of graph structures.

In Chapter 1, we outlined the subtasks of the IE task that our application must perform in order to extract desired entities defined in the data model presented in Section 1.1. To accomplish this, we use the pretrained English language model `en_core_web_trf`, which is based on the Transformer architecture and available implicitly through the *spaCy* library. This model has been pretrained on the OntoNotes 5 dataset[1], a large corpus of English text containing over 2 million words across various genres, including news articles, academic papers, and conversational data. The model can produce state-of-the-art results in a wide range of NLP tasks, including the subtasks of POS tagging and NER that are required for our application.

5.2 Application Architecture

The architecture of the developed application in this thesis consists of three main modules. The *Scraping System* module includes all the business logic related to the scraping processes and the construction of the output hierarchical tree H, as defined in Section 1.2. It consists of four components, as shown in Figure 5.1.

The *Academic Spider* component is responsible for managing the crawling process of the currently processed website through *Scrapy* spider classes. Every time a new web page is visited, it is passed through the *Page Pipeline* component. The pipeline transforms the processed web page into a semantic tree format and assigns it one of the predefined `PAGE_TYPE` tags. These tags indicate whether the spider should scrape data from the processed web page and which scraping

[1] OntoNotes Release 5.0: `https://catalog.ldc.upenn.edu/LDC2013T19`

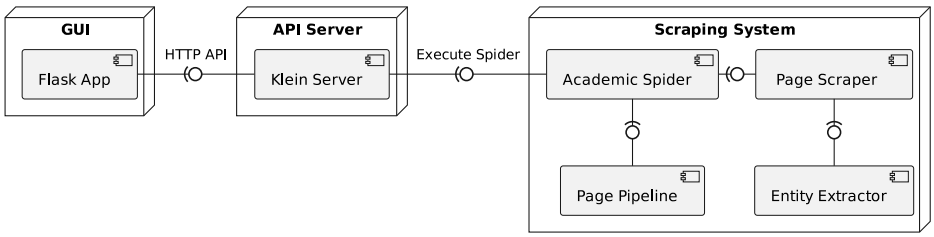

Figure 5.1 Architecture of the application

strategy class from the *Page Scraper* component should be employed for scraping. The scraping strategy classes use methods from the *Entity Detector* component to extract entities and their attributes from the processed web page.

Another crucial part of the application is the *GUI* module. It is implemented as a web application developed in the Flask [35] framework, which is a lightweight web framework that allows for rapid development of web applications. It allows users to view, edit, and export extracted `Organization` and `Person` instances.

The last module of the application is an *API Server*, which serves as a bridge between the web application and the *Scraping System*. It is implemented using Klein [36], which is a micro-web framework. The *API server* receives requests from the *GUI* and directly calls the *Scraping System* to scrape and extract the requested entities from the input website. Afterward, the API server transforms the extracted data into JSON format and sends it back to the web application as a response. In the next section, we will explore in detail each of the modules and their components.

5.3 Page Pipeline

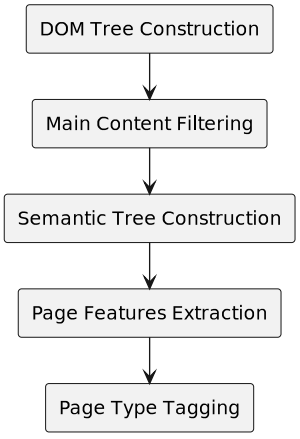

Figure 5.2 Scraping system component

The page pipeline component incorporates five stages, as shown in Figure 5.2. It takes the downloaded HTML source document of the processed web page as input. As the output, it returns a semantic tree as defined in Section 3.2 with an assigned set of page features and `PAGE_TYPE` tag. This collection of outputs is passed as input to the *Page Scraper* component.

5.3.1 DOM Tree Construction

In the first stage of our pipeline, we start by constructing a DOM tree from the input HTML source document. As we mentioned in Section 3.2.1, the DOM tree is a data structure that effectively represents a web page. However, we need to adjust the DOM tree parsed automatically by *BeautifulSoup* to meet our specific requirements.

In a typical DOM tree, leaf nodes represent textual content. However, in our adjusted tree, we will store the values of textual nodes as attributes of their parent element nodes, using an attribute named *text*. Additionally, we will remove the children of `<a>` element nodes and append their *text* attribute values to the *text* attribute of their parent node. This means that each `<a>` element node holds the textual content of its children in its *text* attribute. We will also ignore the document node in the root and set its child as the new root. With these adjustments, we simplified the structure of the DOM tree and made it more suitable for our purposes.

Furthermore, we will define two new attributes for the nodes: *category* and *path*. The *category* attribute holds information about the type of element tag, which will help us detect the branch that contains the main content section of the web page. The *path* attribute contains a sequence of nodes representing the path from the root to the node. This information will allow us to construct the semantic tree in the next stage of the pipeline.

Now we can refer to the adjusted DOM tree as D where $V(D) = \{d_1, \ldots, d_n\}$ is a set of its n element nodes with additional attributes defined above. In Figure 5.3, is shown an example of the adjusted DOM tree and its nodes. A full list of possible categories for tag element categories can be found in the documentation of source code.

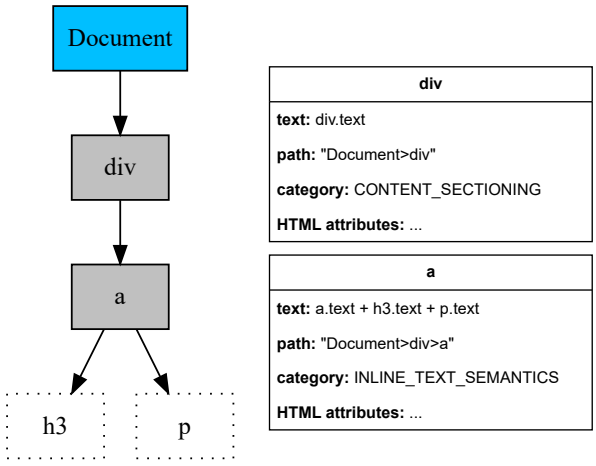

Figure 5.3 Adjusted DOM Tree

5.3.2 Main Content Filtering

The university websites are usually generated from predefined templates. Hence, there are branches in the DOM tree that are identical for different web pages of the website. These branches represent static sections of web page, such as the navigation menu, header, or footer section. The main goal of this *Page Pipeline* stage is to remove these branches from D as they are considered irrelevant to our research objectives. These branches only contribute to the size of D and contain extraneous data.

If the website follows *HTML5* [37] standard, the main content section and the static sections can be easily identified either by specific attributes of element nodes of D or by their special type. This allows for easy identification and removal of branches that represent static sections.

On the contrary, if a website does not follow this standard, a heuristic approach can be employed to detect the main content branch in D. We developed an approach based on [38], in which authors identify the first node of the main content branch using the chars-nodes ratio (CNR). Nevertheless, this approach encounters a challenge as we encounter web pages with varying amounts of text in the main section. When the amount of text present is relatively small, the navigation section is often misidentified as the main section. As a result, we have declined to use this approach when processing web pages that do not conform to the HTML5 standard. In such cases, we retain the original tree D, as it provides a more reliable representation of the web page.

5.3.3 Semantic Tree Construction

Once the irrelevant branches are removed from D in the previous stage, it can be transformed into a semantic tree described in Section 3.2. In the process of this transformation, semantically similar nodes are grouped into the new compound nodes. Let S be the semantic tree constructed from D by Algorithm 2, where $V(S) = \{s_1, \ldots, s_m\}$ is a set of $m \leq n$ compound nodes. A compound node of S is a non-empty set of nodes from $V(D)$, such that $\forall s_i, s_j \in V(S), s_i \neq s_j : s_i \cap s_j = \emptyset$. Note that all the nodes from D are contained in S.

5.3.4 Page Features Extraction

In the page features extraction stage, we extract a set of predefined page features that enable us to recognize web pages which contain information related to either organizations or employees. These features are extracted from the semantic tree S constructed in the previous stage. The semantic tree comprises nodes encapsulating web page elements with the same semantic structure. This provides us with a representation of the web page that is semi-independent from its original structure.

The values of extracted page features are based on identifying and matching NE and a predetermined set of keywords present in the text of the element nodes from S. To accomplish this, we define a function *text_matches* that takes an element node with *text* attribute as input and returns a collection of matched NE and keywords. As previously mentioned in Section 5.3.3, each element node in the adjusted DOM tree D is part of exactly one compound node in the semantic tree S. Therefore, we can calculate the total number of matched NE and keywords in a compound node by counting the outputs of *text_matches* for its member element nodes.

The page features can be divided into two distinct categories based on the types of matches used to obtain their values. These categories are NE-based features and keyword-based features. Based on this distinction, we introduce two new attributes for the compound nodes in the semantic tree S. These attributes are called *ne_tag* and *keyword_tag* and represent the most frequently occurring matched NE and keyword categories, respectively. Moreover, we add these attributes to the element nodes within a compound node, as more types of NE and keywords can be recognized in their *text* attribute value.

5.3.4.1 NE-based features

NE-based features are extracted by identifying and matching NE found within the text of all nodes in the semantic tree S. The NER recognition module we use can recognize numerous types of named entities. However, we limit our focus to two specific types that are most relevant to our task. These are names of people, which the module marks with the ”PERSON” label, and names of organizations, which are marked with the ”ORG” label.

The following list presents a set of predefined NE-based features with their descriptions. The ⟨NE⟩ symbol represents one of the required types of NE.

count of ⟨NE⟩ the total number ⟨NE⟩ entities recognized in every node of S.

count of ⟨NE⟩ nodes the total number of compound nodes whose *ne_tag* attribute value is ⟨NE⟩.

has ⟨NE⟩ in links `True`, if there exists more than one hyperlink element node that contains ⟨NE⟩, otherwise `False`.

has siblings in ⟨NE⟩ links `True` if in the compound node there are at least two hyperlink element nodes with the same *ne_tag* value as the compound node itself, otherwise `False`.

has ⟨NE⟩ in title `True`, if the title of the web page contains ⟨NE⟩, otherwise `False`

5.3.4.2 Keyword-based features

The recognition of NE in the text is not perfect. Although we only work with websites in English, our application should be able to process websites of universities around the world. Hence, we can encounter differences in vocabulary describing entities on the page, naming conventions or in use of abbreviations and acronyms. These differences make it challenging to accurately identify all NE with the pretrained model that we are using.

Fortunately, in the domain of universities, vocabulary for naming organizational units or work positions follows a specific set of guidelines and conventions. For this reason, it is possible to manually establish a vocabulary of keywords typically used to name these entities. We manually collected the list of common naming keywords and divided them into three categories, as shown in Table 5.1.

Category	Keywords
Work Position	professor, lecturer, teacher, instructor, assistant professor, associate professor, dean, researcher, scientist, manager, employee, member
Organizational unit	department, faculty, school, college, institute, center, unit, division, office, structure
Title of people page	people, organization, persons, staff, structure, employees

Table 5.1 Predefined keywords and their categories

The first category includes a set of keywords commonly used to describe the job titles of university personnel. The second category includes a set of keywords commonly used to describe organizational units within the university. The last category consists of keywords commonly used to describe titles of web pages that provide information about university employees. Based on these categories, we designed the following set of binary keyword-based features:

has keyword work position `True`, if there exists an element node that contains any of the words from the work position keyword set, otherwise `False`.

has keyword organizational unit `True`, if there exists an element node that contains any of the words from the organizational unit keyword set, otherwise `False`.

has keyword organizational unit in title `True`, if the title of the web page contains any of the words from the organizational unit keyword set, otherwise `False`

5.3.5 Page Type Tagging

In the last stage of the pipeline, the extracted page features are used to assign a `PAGE_TYPE` tag to the processed web page. The web page can be assigned one of the following `PAGE_TYPE` tags:

PAGE_ORGANIZATIONS is assigned to a page that contains a list of organizational units.

PAGE_SINGLE_PERSON is assigned to a page that provides details about exactly one person.

PAGE_PEOPLE_NO_DATA is assigned to a page that contains a list of people with hyperlinks to a page tagged with `PAGE_SINGLE_PERSON` tag, which contains details about the corresponding person.

PAGE_PEOPLE_DATA is assigned to a page containing a list of people with personal details and no corresponding hyperlinks.

PAGE_GENERAL is assigned to a page that does not satisfy the conditions of the previous tags.

The tag is assigned to the page by a rule-based algorithm. From the tag definitions, it can be deducted that certain page features are more relevant to some tags than others. Therefore, all tags assigned a subset of page features that are relevant to them. Moreover, the algorithm has a predefined set of conditions for each tag that values of selected page features must satisfy. The conditions are assigned weights based on their importance. The algorithm then calculates the relevancy score for each tag by taking the weighted average of the fulfilled conditions. Finally, the tag with the highest relevancy score is assigned to the page.

5.4 Page Scraper

In order to extract meaningful data from the web page and parse it into a hierarchical tree defined in Section 1.2, the system relies on a *Page Scraper* component, which implements a set of scraping strategies. These strategies are classes with multiple methods that extract data from the processed web page and parse it into a relevant set of entity instances. The entity type that is returned depends on the `PAGE_TYPE` tag assigned to the processed web page. The semantic tree S of the processed web page is passed as the input for these strategies. Each strategy class uses specific functions, called operations, from *Entity Extractor* module to extract necessary data from S. Once this data is obtained, strategies parse it and construct corresponding entity instances. We define the following scraping strategies:

Organization strategy processes a page with `PAGE_ORGANIZATIONS` tag, and outputs a set of `Organization` entity instances. It calls the necessary operations to obtain attribute data, from which `Organization` instances are constructed. Afterward, it heuristically filters irrelevant and duplicate instances.

People strategy process a page with `PAGE_PEOPLE_DATA` or `PAGE_PEOPLE_NO_DATA` tag. It calls the necessary operations to obtain attribute data, from which `People` instances are constructed. If the processed page has `PAGE_PEOPLE_NO_DATA` tag, only values of the required attributes are extracted, and the values of the remaining attributes are set after scraping the related page with *Person strategy*.

Person strategy process a page with `PAGE_SINGLE_PERSON` tag. It requires a `Person` instance into which are, after invoking necessary operations, assigned the extracted values of the remaining attributes.

5.5 Entity Extractor

This component consists of several functions called operations. These operations were designed as pure functions, meaning they do not have any side effects and always produce the same output

when given the same input. They take the semantic tree S or one of its compound nodes as required input arguments, with other support variables if necessary. They return data in a format that can be consumed either by another operation or strategies from *Page Scraper* component. All these functions apply heuristic approaches to identify and extract relevant elements in the compound nodes using recognized NE and keywords.

In the following sections, we will explore critical operations that are called from *Page Scraper* component strategies.

5.5.1 Attribute Operations

The operations described in this section aim to identify member tag elements within compound nodes that contain the required attributes for creating instances of `Organization` and `Person` entity classes, and to extract these attributes from them. If these attributes are missing, instantiating the entity classes will not be possible. For `Organization` instances, the required attributes are the *name* and *url*, while for `Person` instances, only the *name* attribute is required.

The operation `ATTRIBUTES_FROM_LINKS` is designed to extract required attributes from hyperlinks elements that contain a specific type of NE, passed as an argument, in their *text* attribute. It checks each node of S to determine if it contains any hyperlink elements. If a hyperlink element is found, the operation adds the part of the *text* attribute that was recognized as NE and the value of its *href* attribute to the output. This operation is used in the *Organization strategy* and the *People strategy* if the processed page has `PAGE_PEOPLE_NO_DATA` tag assigned.

When the People strategy is processing a page with `PAGE_PEOPLE_DATA` tag, it invokes the operation `ATTRIBUTES_FROM_TABLE`. The main purpose of this operation is to extract the required attributes for the `Person` entity. Additionally, it can extract other attributes if they are provided on the processed web page. The operation assumes that information about people is arranged in a table-like structure. Hence, it first identifies elements of this structure and then extracts attribute values for the entity from their corresponding text.

5.5.2 Affiliation Operations

Operations explored in this section were designed to extract affiliations for a `Person` instance. In the Figure 1.1, two attributes were defined for `Affiliation` entity, from which only *division* is required. Therefore, these operations output a set of `Affiliation` entity instances, which are then associated with the corresponding `Person` instance.

If a processed page is labeled with the `PAGE_SINGLE_PERSON` tag and is accessed through the *url* attribute of a `Person` instance, the *Person strategy* will execute the `AFFILIATIONS_FROM_PAGE` operation. This operation entails selecting element nodes from the compound node that possess a *ne_tag* attribute value of 'ORG'. Subsequently, a new instance of `Affiliation` is generated, with the identified portion of the text serving as the value for the *division* attribute. The operation then proceeds to search for the *position* attribute value, which may be located within the element's own *text* attribute or among the *text* attributes of its sibling element nodes. The matching of the *position* value involves comparing the embedding vectors of noun phrases identified in the *text* of element nodes to the embedding vectors of keywords from the work position category, as listed in Table 5.1.

5.6 Academic Spider

The last component of *Scraping system* is made of two Scrapy spiders that control the whole scraping process. Spiders use components described in previous sections to navigate the website of the input university domain and to build an output hierarchical tree H from the scraped data.

The main goal of the first spider is to scrape relevant web pages of $\text{ORG}_{0,0}$ website and to extract its child `Organization` nodes, which will be put on the first level of H. We name this spider *OrganizationSpider*, as it only extracts `Organization` instances. It uses the activity `SCRAPE_ORGANIZATIONS` shown in Figure 5.4, to extract `Organization` instances.

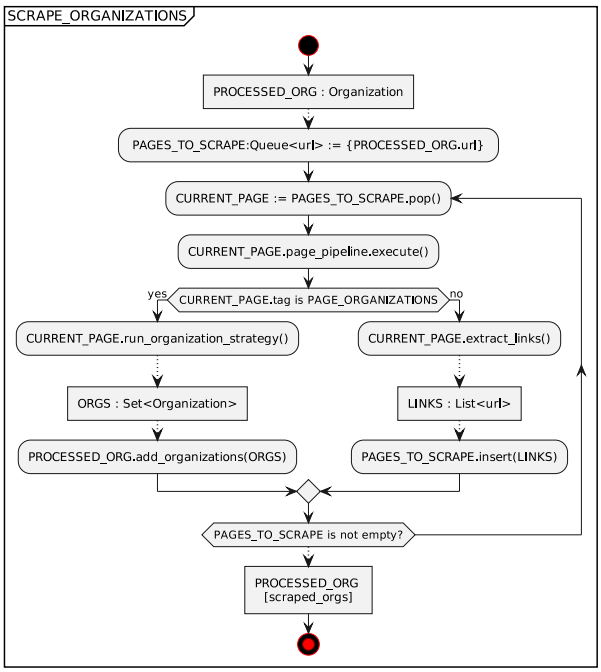

Figure 5.4 Diagram of `SCRAPE_ORGANIZATIONS` activity

The `SCRAPE_ORGANIZATIONS` activity incorporates methods from other components of *Scraping system* to extract required instances. It iteratively scrapes pages of processed `Organization` instance. Using pipeline from *Page Pipeline* component, it constructs its semantic tree and assigns it a `PAGE_TYPE` tag. If the page has `PAGE_ORGANIZATIONS` tag, it calls *Organization strategy* to extract instances. Finally, it assigns extracted instances to the processed `Organization` instance. The function `extract_links` collects internal URLs from each hyperlink tag on the page. Scrapy offers a mechanism that prevents duplicate requests for the same URL.

After the first level of H is filled with the n_0 child nodes $\text{ORG}_{0,1}, \ldots, \text{ORG}_{n_0,1}$ of $\text{ORG}_{0,0}$, each of them is passed as the input to the second spider of this component, called *PeopleSpider*. Organizations at this level represent bigger organizational units, such as faculties, colleges, or institutes. These organizations tend to have their websites hosted on a subdomain of $\text{ORG}_{0,0}$ domain. Hence, the *OrganizationSpider* must identify these subdomains and update the *domain* attribute of extracted instances accordingly.

Because organizations on lower levels represent smaller organizational units like departments, research groups, or offices, therefore, *PeopleSpider* tries to identify and extract these smaller organizations and their employees from the websites of $\text{ORG}_{:,1}$. It scrapes `Organization` instances using the same `SCRAPE_ORGANIZATIONS` activity as *OrganizationSpider*. To extract `Person` instances, it uses activity `SCRAPE_PEOPLE`, shown in Figure 5.5. In the `SCRAPE_PEOPLE` activity, pages of processed `Organization` instance are scraped, as in the `SCRAPE_ORGANIZATIONS` activity. Therefore, after the currently processed web page is assigned one of the people related `PAGE_TYPE` tags, it calls *People strategy* to extract `Person` instances. Moreover, when the page is assigned `PAGE_PEOPLE_NO_DATA` tag, the spider calls *Person strategy* to obtain `Affiliation` instances and additional attributes for each `Person` instance.

The full workflow of *PeopleSpider* is shown in Figure 5.6. In case when no organizations are

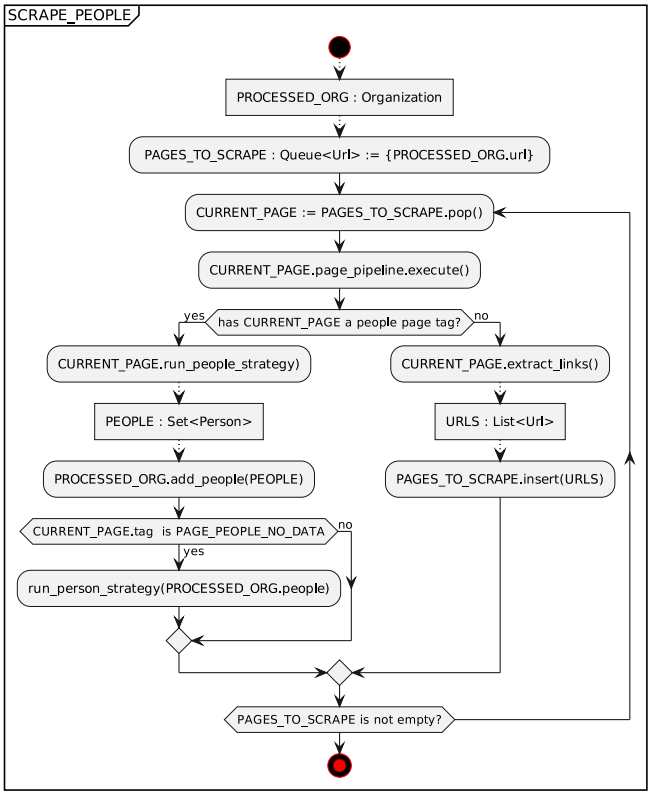

Figure 5.5 Diagram of `SCRAPE_PEOPLE` activity

found, it will try to scrape all `Person` instances from the website that are directly associated with the processed `Organization` instance.

5.7 API Server

The *API Server* is a crucial module in the application architecture, as it serves as the interface between the *GUI* and the *Scraping System*. It receives HTTP requests, processes them, and then calls the appropriate spider from the *Academic Spider* component to perform the requested action. It sends back a JSON response after the scraping process is completed. The *API Server* has only two endpoints, as there are two main spider classes in *Scraping System* module. Each endpoint expects to receive an `Organization` instance converted into JSON format before being passed as input.

Klein micro web framework was used to implement the only web server component of the module. Klein is built on top of *Twisted* [39], which is an event-driven networking engine that allows for high scalability and performance. Since Scrapy is also based on Twisted, using *Klein* as an API server provides good integration with *Scrapy* and can benefit from built-in networking features provided by *Twisted*. Moreover, Klein provides a simple and intuitive routing system for defining API endpoints, which makes it easy to create APIs with different routes for handling different types of requests.

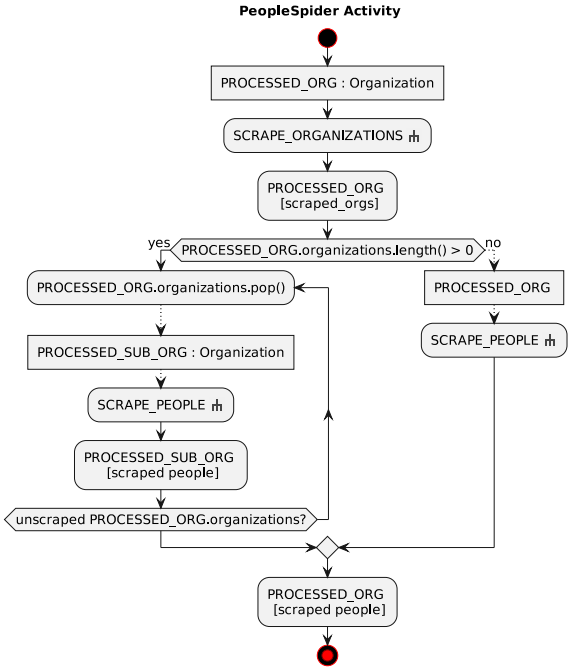

Figure 5.6 Activity diagram of *PeopleSpider*

5.8 GUI

The *GUI* module of the developed application is implemented as a web application using the *Flask* web framework. It provides an intuitive and user-friendly interface that allows users to view, edit and export the extracted profiles produced by the *Scraping System* module. The user interface is designed to be simple and easy to use, with a clear layout that makes it easy for users to navigate and access different features.

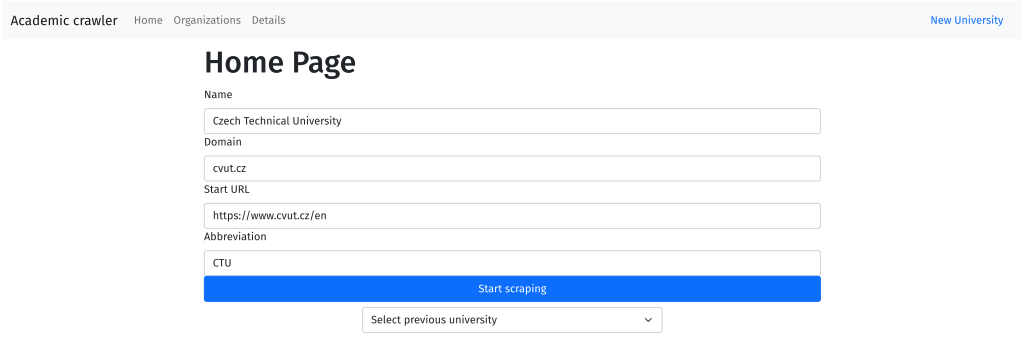

Figure 5.7 Home page of the web interface

The home page of the application, as shown in Figure 5.7, displays a form to which the user fills in information about the input university. After this form is submitted, `Organization` instance of the input university in the JSON format is sent to *API Server* that calls *OrganizationSpider* to extract first level `Organization` instances. The extracted instances are displayed on a separate page, as shown in Figure 5.8, that allows users to edit, add, and remove extracted instances.

Additionally, the user can choose instances that will be scraped in the next phase by *PeopleSpider*. Figure 5.9 shows the page displaying extracted `Person` instances associated to the first level `Organization` instance. Users can edit and remove these instances on this page with their associated `Affiliation` instances.

The web application also allows users to export extracted `Person` instances in JSON format for each first-level `Organization` instance or the whole input university.

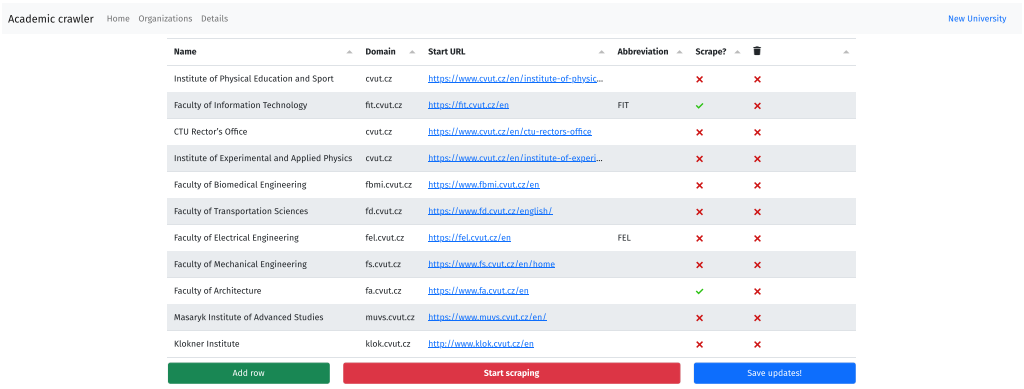

Figure 5.8 Page displaying first-level `Organization` instances

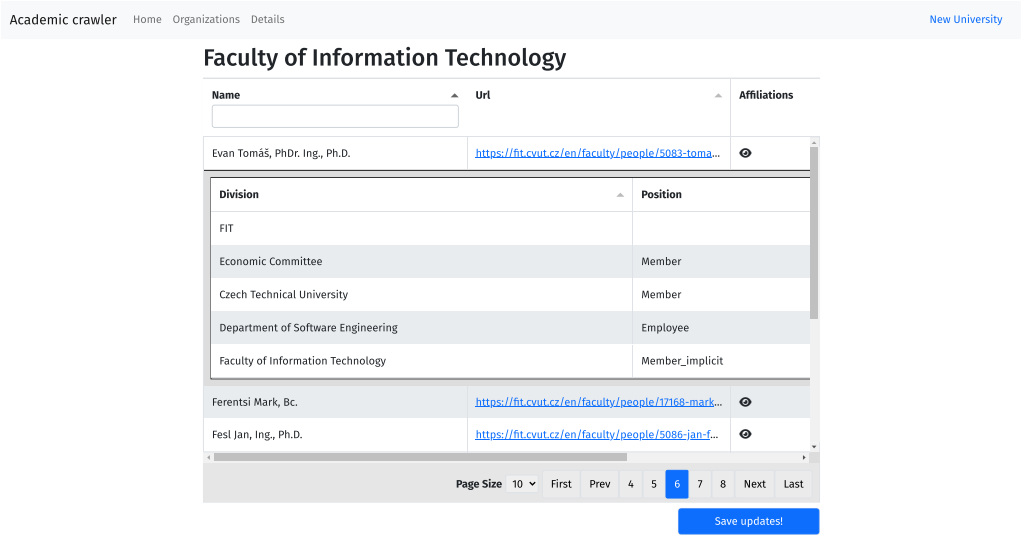

Figure 5.9 Page displaying `Person` instances

Chapter 6
Evaluation

In this chapter, we will evaluate the performance of the proposed application from Chapter 5. The application is designed to build a hierarchical output tree H over two different phases that utilize components from the *Scraping system* module. During the first phase, the *OrganizationSpider* from the *Academic Spider* component fills the first level of the H with organization nodes. In the second stage, the *PeopleSpider* component is utilized to build subtrees for $\text{ORG}_{:,1}$ nodes with `Person` instances assigned to the relevant nodes in these subtrees.

The evaluation process will consist of assessing the performance of these two phases separately. From these evaluation results, we can then derive the overall quality of the output hierarchical tree and the extracted personal profiles. This comprehensive evaluation process will allow us to identify weaknesses within the application and provide suggestions for potential enhancements. It is important to note that application was not designed to handle websites with some predetermined structure. Therefore, its overall performance is heavily influenced by the structure of the processed website, as well as by the accessibility and the quality of the desired data provided on the website.

In order to measure the correctness of a hierarchical tree H and its nodes, we need to construct a hierarchical reference tree R from the university website. This will allow us to compare corresponding levels of the trees using metrics such as precision, recall, and F1 score. Additionally, it will allow us to compare personal profiles assigned to the tree nodes in a similar manner.

6.1 Reference data

A comprehensive collection of hierarchical reference trees constructed from websites with diverse structures is crucial to evaluate our application accurately. However, due to the time-consuming and technically challenging nature of this task, it is not feasible to manually construct an extensive set of these reference trees. Nonetheless, our application can utilize the *Page Pipeline* component to identify web pages containing relevant data and consistently extract it in the required format. Thus, even if we can only evaluate our application on a small set of universities for which we can manually build their hierarchical trees, we can assume that its performance will generalize to other universities. Besides, only a relevant subtree of the hierarchical tree is required to evaluate the spiders. For the first phase, only the hierarchical tree up to the first level is necessary. To evaluate the second phase only subtrees of $\text{ORG}_{:,1}$ are sufficient.

Following this approach, we have collected a set of reference hierarchical trees for each evaluation stage. In total, we have collected four reference trees for both the first and second evaluation stages.

University	Size of $ORG_{:,1}$	Build $ORG_{:,1}$ reference trees
A	14	2
B	17	1
C	9	1
D	80	0

Table 6.1 Reference universities set

In the Table 6.1 are shown, selected reference universities with their parameters. The first parameter is the number of organization nodes at the first level of their hierarchical trees, shown in the second column. The second parameter refers to the number of complete hierarchical subtrees for $ORG_{:,1}$ nodes that were built manually. Regrettably, for university D we were unable to construct a complete reference hierarchical tree for any of its $ORG_{:,1}$ nodes. As for the other universities, we were able to build a total of only three reference trees. This is because most of the scraped websites did not provide the required data publicly or had structurally inconsistent websites.

6.2 Metrics

In order to quantify the performance of our application, the correctness of the output hierarchical tree H needs to be evaluated. This evaluation is based on a comparison by levels of H with the reference tree R. To perform this comparison, we have to establish some evaluation metrics. Each level of a hierarchical tree is a set of `Organization` instances. We can distinguish `Organization` instances in the set using their *name* and *url* attributes. Therefore, we can use implicit set operations between the levels. Let L_H, L_R be levels of H and R respectively. We define confusion matrix between L_H and L_R as follows:

	$\mathbf{L_R}$ **Positive**	$\mathbf{L_R}$ **Negative**
$\mathbf{L_H}$ **Positive**	$\lvert L_H \cap L_R \rvert$	$\lvert L_H \setminus L_R \rvert$
$\mathbf{L_H}$ **Negative**	$\lvert L_R \setminus L_H \rvert$	0

The top-left cell of this matrix represents the number of constructed instances included in the reference set. In contrast, the top-right cell represents the number of constructed instances absent from the reference set. The bottom-left cell represents the number of instances from reference not constructed by the application. The last bottom-right cell is 0 because it represents the number of instances that are neither constructed nor present in the reference set, which is impossible.

Various metrics can be derived from the confusion matrix, such as *precision* and *recall*. They are defined in the following manner:

$$\text{precision}(L_H, L_R) = \frac{|L_H \cap L_R|}{|L_H \cap L_R| + |L_H \setminus L_R|} \qquad \text{recall}(L_H, L_R) = \frac{|L_H \cap L_R|}{|L_H \cap L_R| + |L_R \setminus L_H|}$$

Another metric that can be derived from the confusion is called *F1*. This metric incorporates aspects of both *precision* and *recall metrics*, as it is calculated as the harmonic mean of their values. For levels L_H and L_R it is defined as:

$$\text{F1}(L_H, L_R) = \frac{2 * \text{precision}(L_H, L_R) * \text{recall}(L_H, L_R)}{\text{precision}(L_H, L_R) + \text{recall}(L_H, L_R)}$$

If we calculate the values of these metrics for all levels of H and R, and aggregate them, we will get the overall quality of the H compared to R. It should be noted that if H has fewer levels than R, they can be considered empty sets. This will, however, negatively distort the overall results of each metric. For this reason, we will only compare up to the highest level of H.

Besides the root node of a hierarchical tree, each node can be associated with a set of `Person` instances. With the sole difference being the type of instances included in the set, the correctness of this set can be evaluated using the same metric functions. Based on the data model from Section 1.1, every `Person` instance has at least one implicit `Affiliation` instance, created from `Organization` instance with which is associated. More implicit `Affiliation` are added recursively if associated `Organization` has a parent `Organization` instance. This implicit `Affiliation` instances are the most reliable because they are created from relationships between the web pages rather than from the content of a web page. The `Affiliation` instances created from the content highly depend on the amount of data provided on the single web page. Using this implicit `Affiliation` instance, we can construct a set $P_{i,j}$ of all `Person` instances associated with $ORG_{i,j}$. Therefore, this set includes each `Person` instance associated with the node directly or through its descendants.

6.3 Phase 1

In the first phase, we test the ability of the application to find and construct instances for level 1. This is provided by *OrganizationSpider* from the *Academic Spider* component. The evaluation results of this phase are shown in Table 6.2.

Name	$\|L_H\|$	$\|L_R\|$	precision	recall	f1
A	11	14	0.909	0.714	0.8
B	17	17	0.824	0.824	0.824
C	9	9	0.889	0.889	0.889
D	20	80	1	0.25	0.4

Table 6.2 Evaluation results of phase 1

Based on the results, we can see that *OrganizationSpider* achieved high *precision* in all evaluated levels. Moreover, its recall is impressive for the first three levels, being above 0.8.

However, although the precision is perfect for the first level of the D tree, its recall value is low. This is because the web page from which instances are extracted contains all the child organizations from the organizational hierarchy of D, including those from deeper levels. This outcome indicates that the reference tree does not reflect the actual organizational structure of the university D.

In conclusion, the *OrganizationSpider* has demonstrated its effectiveness in finding and constructing first-level instances for this set of reference universities.

6.4 Phase 2

The main goal of the second phase is to build a hierarchical tree for $ORG_{:,1}$ nodes. Therefore, in this section, we are not evaluating the hierarchical tree of the whole university but only the subtrees of its first-level nodes. Unlike in the first phase, in this phase, we are more interested in evaluating the extracted personal profiles rather than the organizational structure itself. As mentioned in Section 6.2, we can evaluate the set of `Person` instances that are directly associated with $ORG_{i,j}$ or its complete set $P_{i,j}$ that includes instances from descendant nodes. As we want to assess the ability of *PeopleSpider* to extract personal profiles from the website of the first-level `Organization` instance, the complete set for this instance provides the most relevant result for the evaluation.

In the Table 6.3 are presented results for the first level `Organization` instances of the previously presented universities. It is first noticed that *precision* values for $A_{0,1}$ and $B_{0,1}$ instances are low, indicating that most of the constructed instances were irrelevant. However, at least

Name	Size of constructed set	Size of reference set	precision	recall	f1
$A_{0,1}$	535	362	0.649	0.959	0.774
$A_{1,1}$	273	292	0.945	0.884	0.913
$B_{0,1}$	1988	2767	0.309	0.222	0.258
$C_{0,1}$	28	29	0.964	0.931	0.947

Table 6.3 Evaluation results of phase 2

the recall value is high for $A_{0,1}$, suggesting that most reference instances were found. For the instances $A_{1,1}$ and $C_{0,1}$ the *F1* value is high, which indicates good performance. Notably, we observe that results for instances $A_{0,1}$ and $A_{1,1}$ are significantly different, even though they belong to the same university. The reason for this inconsistency is the difference in the structure of their websites.

6.5 Summary and Possible Improvements

Based on the results of the previous sections, we can conclude that both spiders possess the ability to solve their tasks. Although, the limited number of reference trees prevents us from reaching a convincing conclusion about the general performance of the application. Nevertheless, these results certainly showed some shortcomings in the design and implementation of the spiders.

One of the shortcomings is the low precision in phase 2 due to the imperfections of the NER recognition model. This indicates that the model is likely to recognize not only the names of the employees but also other irrelevant noun phrases, causing the creation of irrelevant profiles. A possible solution is to fine-tune the model on the set of personal names that include their academic titles. Furthermore, low recall values for both phases are often attributed to inaccurately assigned `PAGE_TYPE` tags. This can be improved by replacing the rule-based tagging algorithm with a classifier trained on a set of correctly annotated web pages.

Chapter 7

Deployment

This chapter provides instructions on how to deploy our application. As mentioned in Chapter 5, the application was designed as a client-server application. The web application from *GUI* module acts as a client which is communicating with the server in *API Server* module that directly invokes requested spiders from *Scraping System* module.

All source code for the application can be found in the *src* folder on the enclosed medium. In its subfolder *exec*, we can find installation script *install.sh*, that will set up a new Python environment with all required libraries and models. After the environment is ready, we can execute script *run.sh* that will start both the Flask web application and Klein API server.

Conclusion

The primary objective of this thesis was to develop a software solution that automatically extracts the organizational structure and personal profiles of employees associated with each organizational unit from any given university website by utilizing web scraping and NLP techniques. The second objective was to develop a user-friendly interface that facilitates verifying and modifying the extracted personal profiles. The motivation for this thesis was the potential benefit businesses could gain from having access to the real-time organizational structure of a university of interest.

The application consists of several components that can be easily replaced, allowing for new approaches for web page transformation or profile and organization extraction in the future. This design allows the application to process a wide range of universities while also allowing it to be fine-tuned to process a specific set of universities.

Both objectives were successfully achieved as the application can build the organizational hierarchy and extract relevant personal profiles from the input university website. Using the developed user interface, the extracted personal profiles can then be exported in JSON format.

In cases where the data representing the organizational units and personal profiles is not clearly presented on the input website, the quality of the output is compromised.

Although the evaluation results suggest that the application is able to extract relevant profiles for the universities in the reference set, the small size of the reference set affects the representativeness of the results. In the future, it would be desirable to test the application performance on a broader sample of universities.

Bibliography

1. MANNING, Christopher; RAGHAVAN, Prabhakar; SCHUETZE, Hinrich. *Introduction to Information Retrieval*. 2009. ISBN 0-521-86571-9.
2. GRISHMAN, Ralph. Information Extraction. *IEEE Intelligent Systems*. 2015, vol. 30, no. 5, pp. 8–15. ISSN 1941-1294. Available from DOI: 10.1109/MIS.2015.68.
3. VERSPOOR, Karin; COHEN, Kevin. Natural Language Processing. In: 2013, pp. 1495–1498. ISBN 978-1-4419-9862-0. Available from DOI: 10.1007/978-1-4419-9863-7_158.
4. MADNANI, Nitin. Getting Started on Natural Language Processing with Python. *XRDS: Crossroads, The ACM Magazine for Students* [online]. 2007, vol. 13, no. 4, pp. 5–5 [visited on 2023-03-13]. ISSN 1528-4972, ISSN 1528-4980. Available from DOI: 10.1145/1315325.1315330.
5. JURAFSKY, Daniel; JAMES H. MARTIN. *Speech and Language Processing* [online]. 3rd ed. draft. 2023 [visited on 2023-03-10]. Available from: https://web.stanford.edu/~jurafsky/slp3/ed3book_jan72023.pdf.
6. NASEER, Salman; GHAFOOR, Muhammad; SOHAIB; KHALID ALVI, Sohaib; KIRAN, Anam; REHMAN, Shafique Ur; MURTAZA, Ghulam; CAMPUS, Jehlum; JEHLUM, Pakistan. Named Entity Recognition (NER) in NLP Techniques, Tools Accuracy and Performance. *Pakistan Journal of Multidisciplinary Research*. 2022, vol. 2, pp. 293–308.
7. SINGH, Sonit. *Natural Language Processing for Information Extraction* [online]. arXiv, 2018 [visited on 2023-04-22]. No. arXiv:1807.02383. Available from arXiv: 1807.02383 [cs].
8. EKMAN, MAGNUS. *Learning Deep Learning: Theory and Practice of Neural Networks, Computer Vision, NLP, and Transformers Using TensorFlow*. 2021. ISBN 978-0-13-747035-8.
9. BENGIO, Yoshua; DUCHARME, Réjean; VINCENT, Pascal. A Neural Probabilistic Language Model. *Advances in neural information processing systems*. 2000, vol. 13.
10. MIKOLOV, Tomas; CHEN, Kai; CORRADO, Greg; DEAN, Jeffrey. *Efficient Estimation of Word Representations in Vector Space* [online]. arXiv, 2013 [visited on 2023-04-06]. No. arXiv:1301.3781. Available from arXiv: 1301.3781 [cs].
11. PENNINGTON, Jeffrey; SOCHER, Richard; MANNING, Christopher. Glove: Global Vectors for Word Representation. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)* [online]. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1532–1543 [visited on 2023-04-06]. Available from DOI: 10.3115/v1/D14-1162.

12. MIKOLOV, Tomas; GRAVE, Edouard; BOJANOWSKI, Piotr; PUHRSCH, Christian; JOULIN, Armand. Advances in Pre-Training Distributed Word Representations. *arXiv preprint:1712.09405*. 2017. Available from arXiv: 1712.09405.

13. GERS, Felix A.; SCHMIDHUBER, Jürgen; CUMMINS, Fred. Learning to Forget: Continual Prediction with LSTM. *Neural computation*. 2000, vol. 12, no. 10, pp. 2451–2471.

14. HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long Short-Term Memory. *Neural computation*. 1997, vol. 9, no. 8, pp. 1735–1780.

15. VASWANI, Ashish; SHAZEER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N.; KAISER, Lukasz; POLOSUKHIN, Illia. *Attention Is All You Need* [online]. arXiv, 2017 [visited on 2023-04-07]. No. arXiv:1706.03762. Available from DOI: 10.48550/arXiv.1706.03762.

16. DEVLIN, Jacob; CHANG, Ming-Wei; LEE, Kenton; TOUTANOVA, Kristina. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* [online]. arXiv, 2019 [visited on 2023-04-23]. No. arXiv:1810.04805. Available from arXiv: 1810.04805 [cs].

17. LIU, Yinhan; OTT, Myle; GOYAL, Naman; DU, Jingfei; JOSHI, Mandar; CHEN, Danqi; LEVY, Omer; LEWIS, Mike; ZETTLEMOYER, Luke; STOYANOV, Veselin. RoBERTa: A Robustly Optimized BERT Pretraining Approach [online]. 2019 [visited on 2023-03-30]. Available from DOI: 10.48550/ARXIV.1907.11692.

18. SINGRODIA, Vidhi; MITRA, Anirban; PAUL, Subrata. A Review on Web Scrapping and Its Applications. In: *2019 International Conference on Computer Communication and Informatics (ICCCI)*. 2019, pp. 1–6. ISSN 2329-7190. Available from DOI: 10.1109/ICCCI.2019.8821809.

19. PERSSON, Emil. *Evaluating Tools and Techniques for Web Scraping* [online]. 2019 [visited on 2023-03-08]. Available from: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-271206.

20. *XML Path Language (XPath) 3.1* [online] [visited on 2023-03-08]. Available from: https://www.w3.org/TR/xpath-3/.

21. *Beautiful Soup Documentation — Beautiful Soup 4.4.0 Documentation* [online] [visited on 2023-04-22]. Available from: https://beautiful-soup-4.readthedocs.io/en/latest/.

22. *Lxml - Processing XML and HTML with Python* [online] [visited on 2023-04-22]. Available from: https://lxml.de/index.html.

23. *Scrapy 2.8 Documentation — Scrapy 2.8.0 Documentation* [online] [visited on 2023-03-08]. Available from: https://docs.scrapy.org/en/latest/.

24. FLANAGAN, David. *JavaScript: The Definitive Guide*. Sixth. O'Reilly Media, Inc., 2011. ISBN 978-0-596-80552-4.

25. *What Is the Document Object Model?* [Online] [visited on 2023-03-27]. Available from: https://www.w3.org/TR/WD-DOM/introduction.html.

26. MUKHERJEE, S.; GUIZHEN YANG; WENFANG TAN; RAMAKRISHNAN, I.V. Automatic Discovery of Semantic Structures in HTML Documents. In: *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.* [Online]. Edinburgh, UK: IEEE Comput. Soc, 2003, vol. 1, pp. 245–249 [visited on 2023-02-17]. ISBN 978-0-7695-1960-9. Available from DOI: 10.1109/ICDAR.2003.1227667.

27. ARTILES, Javier; BORTHWICK, Andrew; GONZALO, Julio; SEKINE, Satoshi; AMIGÓ, Enrique. WePS-3 Evaluation Campaign: Overview of the Web People Search Clustering and Attribute Extraction Tasks. In: *CLEF (Notebook Papers/LABs/Workshops)*. 2010.

28. NAGY, István. Person Attribute Extraction from the Textual Parts of Web Pages. *Acta Cybernetica*. 2012, vol. 20, no. 3, pp. 419–440.

29. HAN, Xianpei; ZHAO, Jun. CASIANED: People Attribute Extraction Based on Information Extraction. *City*. 2009, pp. 20–24.

30. LAN, Man; ZHANG, Yu Zhe; LU, Yue; SU, Jian; TAN, Chew Lim. Which Who Are They? People Attribute Extraction and Disambiguation in Web Search Results. In: *2nd Web People Search Evaluation Workshop (WePS 2009), 18th WWW Conference*. 2009.

31. TANG, Jie; ZHANG, Jing; YAO, Limin; LI, Juanzi; ZHANG, Li; SU, Zhong. ArnetMiner: Extraction and Mining of Academic Social Networks. In: *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* [online]. Las Vegas Nevada USA: ACM, 2008, pp. 990–998 [visited on 2023-03-14]. ISBN 978-1-60558-193-4. Available from DOI: 10.1145/1401890.1402008.

32. NAGY, István; FARKAS, Richárd; JELASITY, Márk. Researcher Affiliation Extraction from Homepages. In: *Proceedings of the 2009 Workshop on Text and Citation Analysis for Scholarly Digital Libraries (NLPIR4DL)* [online]. Suntec City, Singapore: Association for Computational Linguistics, 2009, pp. 1–9 [visited on 2023-05-02]. Available from: https://aclanthology.org/W09-3601.

33. HONNIBAL, Matthew; MONTANI, Ines; VAN LANDEGHEM, Sofie; BOYD, Adriane. spaCy: Industrial-strength Natural Language Processing in Python. 2020. Available from DOI: 10.5281/zenodo.1212303.

34. *Proceedings of the Python in Science Conference (SciPy): Exploring Network Structure, Dynamics, and Function Using NetworkX* [online] [visited on 2023-03-25]. Available from: https://conference.scipy.org/proceedings/SciPy2008/paper_2/.

35. *Welcome to Flask — Flask Documentation (2.2.x)* [online] [visited on 2023-04-10]. Available from: https://flask.palletsprojects.com/en/2.2.x/.

36. *Klein, a Web Micro-Framework — Klein 16.12.0 Documentation* [online] [visited on 2023-04-10]. Available from: https://klein.readthedocs.io/en/stable/.

37. *HTML5* [online] [visited on 2023-04-22]. Available from: https://www.w3.org/TR/2011/WD-html5-20110405/.

38. LÓPEZ, Sergio; SILVA, Josep; INSA, David. Using the DOM Tree for Content Extraction. *Electronic Proceedings in Theoretical Computer Science* [online]. 2012, vol. 98, pp. 46–59 [visited on 2023-02-21]. ISSN 2075-2180. Available from DOI: 10.4204/EPTCS.98.6.

39. *Twisted* [online] [visited on 2023-04-10]. Available from: https://twisted.org/.

Contents of Enclosed Medium

- `Readme.md` ..... installation instructions
- `src`
  - `application` ..... source code of *API* and *Scraping system* modules
  - `interface` ..... source code of *GUI* module
  - `thesis` ..... source code of thesis document in LaTeX
  - `exec` ..... scripts to install and run the application
- `text` ..... text of thesis
  - `thesis.pdf` ..... text of thesis in PDF format