BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

AUTOMATIC WEBPAGE CONTENT CATEGORISATION AND EXTRACTION

AUTOMATICKÁ KATEGORIZACE A EXTRAKCE DAT Z WEBOVÝCH STRÁNEK

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MICHAL REIN**
AUTOR PRÁCE

SUPERVISOR **Ing. DANIEL DOLEJŠKA**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

145581

Institut:	Department of Information Systems (UIFS)
Student:	**Rein Michal, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Application Development
Title:	**Automatic Webpage Content Categorisation and Extraction**
Category:	Speech and Natural Language Processing
Academic year:	2022/23

Assignment:

1. Study, analyze and describe the problem of automatic categorization and extraction of text data.
2. Identify and compare the technologies, tools and techniques currently used to address the issues of this thesis.
3. Design a system that enables automatic processing of web page files in order to categorize its content or extract additional data.
4. Implement the proposed system according to the supervisor's instructions.
5. Analyze, verify and adequately test the implemented system according to the supervisor's instructions.
6. Discuss the results obtained and their possible extensions.

Literature:

- DIOUF, Rabiyatou, SARR, Edouard Ngor, SALL, Ousmane, BIRREGAH, Babiga, BOUSSO, Mamadou and MBAYE, Seny Ndiaye, 2019. Web Scraping: State-of-the-Art and Areas of Application. In: *2019 IEEE International Conference on Big Data (Big Data)*. Online. Los Angeles, CA, USA: IEEE. December 2019. p. 6040–6042. [Accessed 20 October 2022]. ISBN 978-1-72810-858-2. DOI 10.1109/BigData47090.2019.9005594.
- GLEZ-PEÑA, Daniel, LOURENÇO, Anália, LÓPEZ-FERNÁNDEZ, Hugo, REBOIRO-JATO, Miguel and FDEZ-RIVEROLA, Florentino, 2014. Web scraping technologies in an API world. *Briefings in Bioinformatics*. September 2014. Vol. 15, no. 5, p. 788–797. DOI 10.1093/bib/bbt026.
- ALLEN, James, 1995. *Natural language understanding*. . 2nd ed. Redwood City, Calif: Benjamin/Cummings Pub. Co. ISBN 978-0-8053-0334-6. QA76.7 .A44 1995
- CHOWDHARY, K. R., 2020. Natural Language Processing. In: CHOWDHARY, K.R., *Fundamentals of Artificial Intelligence*. Online. New Delhi: Springer India. p. 603–649. [Accessed 20 October 2022]. ISBN 978-81-322-3970-3.
- HIRSCHBERG, Julia and MANNING, Christopher D., 2015. Advances in natural language processing. *Science*. 17 July 2015. Vol. 349, no. 6245, p. 261–266. DOI 10.1126/science.aaa8685.

- according to the supervisor's instructions

Requirements for the semestral defence:
Points 1, 2 and 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Dolejška Daniel, Ing.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	26.10.2022

Abstract

This thesis describes the development of a flexible system for automatically categorising and extracting content from web pages, with a focus on the darknet environment. We have designed a highly adaptable and scalable system capable of handling any type of content, while taking great care in considering the overall architecture, database structure, and processing pipeline. Using the state-of-the-art language model trained on the natural language inference task, we demonstrate the model's potential to categorise content effectively in a zero-shot environment. We also conduct an analysis of the performance of various hypothesis templates. To further enhance the data extraction process, we have integrated a named entity recognition model and templating methodology for content extraction and proposed an automated segmentation approach using OpenAI's ChatGPT model. In addition, we have developed a user-friendly web client application to enhance the system's accessibility and ease-of-use, evaluated the achieved results, and identified areas for further research and development in this field.

Abstrakt

Tato práce popisuje vývoj flexibilního systému pro automatickou kategorizaci a extrakci obsahu z webových stránek, se zaměřením na prostředí darknetu. Navrhli jsme vysoce přizpůsobitelný a škálovatelný systém, který dokáže zpracovávat různorodý typ obsahu, přičemž jsme dbali na kvalitu návrhu celkové architektury, struktury databáze a samotného algoritmu pro zpracování dat. Použitím nejmodernějšího jazykového modelu trénovaného na úkolu inference přirozeného jazyka demonstrujeme potenciál modelu efektivně kategorizovat obsah v zcela neznámém prostředí, přičemž jsme provedli analýzu výkonu daného modelu za použití různých hypotetických šablon. Dále jsme do systému integrovali model pro rozpoznávání pojmenovaných entit a metodologii šablonování pro extrakci obsahu, přičemž jsme navrhli automatizovaný přístup k segmentaci obsahu webových stránek za pomocí modelu ChatGPT od společnosti OpenAI. V neposlední řadě jsme vyvinuli uživatelsky přívětivou webovou aplikaci pro zlepšení dostupnosti a snadné použití systému, zhodnotili dosažené výsledky a navrhli možnosti pro další výzkum a vývoj v dané oblasti.

Keywords

content categorisation, natural language processing, natural language inference, named entity recognition, templating, microservice architecture, darknet

Klíčová slova

kategorizace obsahu, zpracování přirozeného jazyka, inference přirozeného jazyka, rozpoznávání pojmenovaných entit, šablonování, architektura mikroslužeb, darknet

Reference

REIN, Michal. *Automatic webpage content categorisation and extraction.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Daniel Dolejška

Automatic webpage content categorisation and extraction

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Daniel Dolejška. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Michal Rein
May 15, 2023

Acknowledgements

I would like to express my thanks and gratitude to my supervisor Ing. Daniel Dolejška for professional guidance throughout the preparation of this thesis and all of my family members for incredible amount of patience and support.

Contents

Chapter 1

Introduction

This thesis focusses on the crucial task of automatic webpage content categorisation and extraction, which holds significant importance in today's world where the majority of information is sourced from the web. With the tremendous surge in information available online, there is a growing need to automatically classify and extract information from web pages, especially from highly niche domains.

In general, webpage categorisation involves assigning a category or label to its content. This can be useful for tasks such as web search and information retrieval, where it is essential to quickly and accurately classify content into relevant categories that best suit the query of the user.

Automatic extraction involves extracting specific information or entities from a webpage, such as a title, author, or any other potentially interesting data and entities. This can be useful for creating data sets and knowledge bases that can be used for further analysis and evaluation of the content present on the web.

In order to solve the above-mentioned tasks effectively, it is necessary to use various natural language processing techniques, machine learning, and deep learning algorithms that use highly sophisticated methods to process textual content. By leveraging these cutting-edge techniques, it is possible to analyse and process the text data on entire web pages and to train language models that can precisely categorise and extract information from the content.

The subject of automatic webpage categorisation and extraction is an important and challenging topic, and this thesis will provide a detailed analysis of the methods and techniques that can be used to achieve satisfying results. In addition, our goal is to create a modular and scalable system called WebCat, which provides tools for analysing content from the web pages and the large web archives. With the use of natural language processing techniques, this system will be able to accurately analyse content and search for specific entities, anomalies, or categories. The modular and distributed design of the system will allow it to be easily adapted and extended for different applications and use cases, and its scalable architecture will handle large volumes of data, making it suitable for use in real-world scenarios.

Overall, our goal is to create a powerful tool that can help improve the efficiency and effectiveness of categorisation and information extraction from web pages, especially in challenging environments such as darknet forums and markets, or other domains that are highly unconventional and niche.

Chapter 2

Natural Language Processing

The study of natural language processing (NLP) is a subfield of computer science and artificial intelligence that focusses on the development of algorithms and techniques for the analysis, processing and generation of natural language data, such as text and speech. This field plays a crucial role in enabling computers to understand, interpret, and communicate with humans in a natural and intuitive way [15]. Using the principles of NLP, researchers and developers can create systems that can automatically analyse, understand, and generate human language, making it possible to develop a wide range of applications in areas such as machine translation, dialogue systems, information extraction, and many more.

Depending on the specific task and the type of data that are being analysed, various techniques and algorithms can be used. Some common techniques used in NLP include tokenization, lemmatisation, stemming, named entity recognition, topic modelling, and much more [30]. These techniques can be applied to a wide range of data sources, including text documents, social media posts, and speech recordings, to extract valuable information. In recent years, the construction of language models has shifted more from the classical statistical methods, such as the usage of the Naive Bayes and Hidden Markov Chains, towards the methods based on training deep neural networks. The main difference between these two approaches is how they analyse, process, and learn from natural language data.

Statistical methods use mathematical and statistical techniques to analyse data. These methods typically involve creating a statistical model of the language from the input data and using the model to make predictions or generate text. For example, a statistical model might use Hidden Markov Chains or Naive Bayes to learn a probability distribution of the language or topic and generate text based on the patterns and structure of the language data [32].

In contrast, neural network-based methods use artificial neural networks, which involves training a neural network on a large natural language data set and using the trained network to perform various downstream tasks. These methods are often more complex and computationally intensive than statistical methods, but they can provide more accurate and sophisticated results.

The choice between statistical methods or neural networks depends on the specific tasks and types of data analysed. Although statistical methods are often simpler and more efficient in calculation, they may not be as accurate as methods based on neural networks. Neural network-based methods are often more complex and computationally intensive, but can provide more accurate results, especially in some highly complex tasks.

Overall, NLP is an important and rapidly evolving field of computer science, mostly due to recent advances in artificial intelligence (AI), and has a wide range of applications in

many different areas. By using highly sophisticated techniques and algorithms, it is possible to gain valuable insights and information from natural language data and to develop more intelligent and human-like computer systems.

2.1 Common Terminology and Techniques

In order to make understanding NLP problems in the coming chapters more intuitive, it is important to have a basic understanding of the common terminology and techniques used in this field. This chapter will provide an introduction to some of the key terms and concepts, as well as an overview of the most commonly used techniques and algorithms.

Corpus

In the context of natural language processing, a corpus is a digital collection of text data that is used to train and evaluate NLP algorithms and models. A corpus typically includes a large number of text documents, such as news articles, books, or social media posts, and can be used to represent a specific language, domain, or topic. It can also provide valuable information on the characteristics and patterns of a specific language or domain. When selecting a corpus for creating a language model, it is important to consider several key properties [33], including

- **Size**: The corpus should be large enough to provide a sufficient amount of data.
- **Variety**: The corpus should include a diverse range of text types, including genres, styles, and formats that are ideally evenly distributed.
- **Annotations**: The corpus should be annotated with relevant labels, such as part-of-speech tags, named entities, or categories, to support more advanced NLP tasks. However, the selection of the annotation type depends on the task that is being solved.
- **Quality**: The corpus should be free of errors and inconsistencies and should be carefully curated to ensure its accuracy and reliability.

In general, a corpus is an essential tool for many tasks, as it provides a large and representative data set that can be used to train and evaluate algorithms and models.

Vocabulary

Vocabulary is a list of words that are used in a given text or corpus. The process of creating a vocabulary typically involves extracting words from the text data and organising them into a list, while each word appears only once. These distinct items in a vocabulary are also called „word types“ [38].

By creating a vocabulary from the text data, it becomes possible to represent the text data using a fixed and finite set of words or tokens, which can simplify the analysis and processing of the data. The finite property of the vocabulary also allows for indexing, which can be used to transform words into numeric representations that can further serve as input to the language models.

Many NLP techniques and algorithms work with the vocabulary. For example, a common technique is to build a statistical model based on the vocabulary of a corpus, which can then be used to classify new pieces of text or to generate text that is similar to the samples in the corpus, preserving the same style of writing.

Tokenization

Tokenization is the process of breaking a piece of text into smaller pieces, called tokens [26]. These tokens can be words or symbols and serve as basic building blocks for many NLP tasks. This process is typically done using a combination of rules-based and statistical methods. There are several types of tokenizers used in natural language processing, including

- **Word tokenizers**: For splitting the text into individual words.
- **Sentence tokenizers**: The text is split into individual sentences.
- **Regex tokenizers**: These tokenizers use regular expressions to identify patterns in text and split them into tokens based on those patterns.
- **Character tokenizers**: For splitting the text into individual characters.

Tokenization is an important preprocessing step for any NLP task, as it allows the text to be represented in a more structured and manageable form. Breaking the text into smaller pieces makes it easier to analyse and process the data and extract useful information and insights from the text. Individual tokens can be indexed using the vocabulary to create a numerical representation of the input; however, new or unseen tokens that are not present in the vocabulary are often replaced by a special token `[UNK]`.

Stemming and Lemmatisation

Both stemming and lemmatisation are processes to reduce inflected (or sometimes derived) words to their stem, base, or root form of the word [26]. This is typically done by removing the suffixes or prefixes of words that are used to indicate a grammatical function, such as tense, gender, or plurality. The difference between these two processes is that stemming is a heuristic process that removes the ends of words, while lemmatisation is based on morphological analysis and vocabulary-based algorithms [26]. The example of stemming can be seen in Figure 2.1, while lemmatisation is shown in Figure 2.2.

$$cooked, overcooked, cooking \implies cook$$

Figure 2.1: Example of words 'cooked', 'overcooked', and 'cooking' being processed by stemming and resulting in the word 'cook'.

$$am, are, is \implies be$$

Figure 2.2: Example of words 'am', 'are', and 'is' being lemmatized and resulting in the word 'be'.

This processing can be useful for tasks such as text classification, where the exact form of a word may not be important, but its basic meaning or concept is. When words are reduced to their root form, it becomes easier to group and classify words based on their underlying meaning, rather than their specific inflected form. The root form of words can also be used to construct a robust and compact vocabulary for algorithms based on the frequency of the term, which will be discussed in the next chapters.

Overall, stemming and lemmatisation are important preprocessing steps in many natural language processing tasks, as they can help to reduce the complexity of the text data and make it more applicable for analysis and processing by NLP algorithms.

2.2 Word Representation

Word representation refers to the way words are represented in a numerical or structured form. This representation is important, as it allows words to be input to the NLP algorithms, which require numbers as their input rather than a sequence of bytes representing characters, in which way words are being represented in today's computers. The most sufficient form of word representation for natural language processing tasks is a vector, as it allows the capture of properties of words across many dimensions that a simple scalar value is not capable of. Furthermore, any linear operation can be applied to these vectors, which makes it possible to measure similarity between pairs of words or construct higher-level representations of entire contexts, such as paragraphs or documents themselves. There are two main categories of word representations:

- Local representations (can also be referred to as one-hot), where every word in a fixed vocabulary of size D is represented by a binary vector $\vec{v} \in \{0,1\}^{|S|}$. Each dimension of the vector $\vec{v}$ corresponds to one word in the vocabulary, and only one value in this vector will be set to one, depending on the word it should represent [29].
- Distributed representations, where each word is represented by a vector $\vec{v} \in \mathbb{R}^{|k|}$, where $\vec{v}$ can be sparse or dense (hand-made features or latent representation) [29]. In common terminology, these representations are referred to as embeddings.

The common fact for every word representation method is that they are crafted or trained based on words that occur in some training corpora. One of the key issues includes the representation of words which were not present in the reference corpus. It is unrealistic for models to include every word which exists in the language, furthermore, new words are being introduced continuously, which would require creating new embeddings on regular basis. For this purpose, only a subset of the most frequent terms are included in the vocabulary, together with the special `[UNK]` symbol, which stands for any word outside of the vocabulary.

Word Embeddings

Word embeddings are numerical representations of words that capture their semantic and syntactic properties. These representations are typically created by training a neural network on large corpora and using the trained network to generate numerical representations of words. Once the model is trained, the transition of the word index in the vocabulary space can be directly mapped to the vector space. There are also highly effective classical methods for creating embeddings, often classified into two main families [35]:

- **Local context window methods**, for example, skip-gram model [27].
- **Global matrix factorisation methods**, for example, LSA (latent semantic analysis) [8].

Embeddings made in such a way can then be used to represent words in a more sophisticated and accurate way and are often used for most of the tasks in the NLP domain. The desirable feature of high-quality embeddings includes capturing semantic similarities between words themselves. As mentioned above, the magnitude of these similarities can be measured using the distance metric $F(\vec{w_1}, \vec{w_2}) = s$, where F is a function (for example, the cosine similarity function), $\vec{w_1}$ and $\vec{w_2}$ are embeddings of two different words and s is the measured

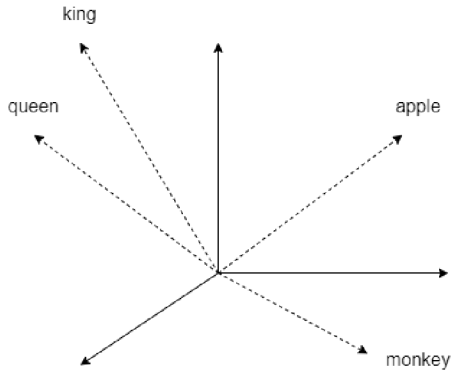


Figure 2.3: The 3D projection of the K-dimensional latent space of the embeddings.

similarity. Figure 2.3 shows a latent space of the embeddings projected in the 3D space with the vocabulary
$V = \{apple, monkey, king, queen\}$. As can be seen, the words „king“ and „queen“ are semantically more similar than the rest of the words in the vocabulary and thus their vectors are much closer to each other.

Word2vec

Word2vec is based on the idea that the meaning of a word can be inferred by the words that frequently appear near it in a text [28].

This approach to creating word embeddings uses a shallow neural network to learn the vector representation of words from large corpora. One way in which word2vec can be trained is that the model takes a word as input and tries to predict the surrounding words in the context of the sentence. This is known as the continuous skip-gram model. Alternatively, the model can be trained to predict the original input word while knowing the surrounding words, which is known as the continuous bag-of-words (CBOW) model.

This approach has been shown to produce high-quality word embeddings that capture both the semantic and syntactic properties of words, and it is a popular and effective method for preprocessing textual data in many different NLP applications.

GloVe

GloVe (Global Vectors for Word Representation) is a global log-bilinear regression method to train word embedding models [35]. It is similar to Word2Vec but uses a different approach to learning word vectors. Instead of predicting surrounding words, GloVe learns word vectors by directly modelling the co-occurrence probabilities of words in a corpus of text. The training process involves constructing a co-occurrence matrix, which counts how often each word co-occurs with every other word in the corpus. The rows and columns of this matrix represent words, and the elements in the matrix represent the number of times each pair of words co-occurs. This matrix is then factorised using a technique called singular value decomposition, which yields the word vectors.

2.3 Document Representation

The document can be seen as the highest linguistic unit of the natural language [24]. The goal is to create a fixed size representation **d**, which contains information about the content, which is typically composed of words $D = \{w_1, w_2, w_3, ..., w_n\}$, where D is the document. As with the word representations described in Section 2.2, the division into categories follows the same pattern as the representations of words:

- **Local representation (one-hot)**, typically bag-of-words, or representation based on term frequencies. The order of tokens in the document is not preserved, and the methods rely on the meaning of individual words alone, especially nouns.
- **Distributive representation**, typically transforming context into a latent space of dimensionality much smaller than the size of the vocabulary, with the help of neural network architectures capable of encoding sequences of tokens.

Bag of Words

The bag-of-words (BOW) technique is a method of representing textual data in natural language processing. It involves representing a piece of text as a fixed-length vector of word counts, where each dimension of the vector corresponds to a specific word in the text.

To create a bag-of-words representation of a document, the first step is to define a vocabulary $V = [w_1, w_2, w_3, ..., w_{|V|}]$ that contains a fixed number of the most frequent words that are present within a document, or by using a predefined list of words. The hot representation of each word is $w = [0, 0, ..., 1, 0, ..., 0]$, as defined in 2.2. The resulting bag-of-words representation can be defined as

$$d = \sum_{k=1}^{l} w_i,$$

where l is the length of the document. It is clear that this representation alone only counts the word frequencies in the document, and in real-world scenarios, bag-of-words is being used as a tool to generate word count features [24].

TF-IDF

The TF-IDF (term frequency-inverse document frequency) is a numerical representation of a word's importance in a document or a collection of documents. It is commonly used in information retrieval tasks [29].

In TF-IDF, the term frequency (TF) of a word is the number of times that word appears in a document, which corresponds to the output representation of the bag-of-words method. The inverse document frequency (IDF) of a word is a measure of how rare or common that word is in a collection of documents. The product of these two values is used to represent the importance of a word in a given document. The relative frequency of the term TF is calculated as

$$TF(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}},$$

where $f_{t,d}$ is the raw count of the term t in the document d, and the denominator is the sum of all terms in the document d. The part of IDF can be calculated as

$$IDF(t, D) = \log \frac{N}{1 + |\{d \in D : t \in D\}|},$$

where N is the total number of documents in corpus D. The denominator in this case counts the number of documents in which the term t appeared at least once. The final equation is the product of the TF and IDF parts:

$$TF\text{-}IDF(t, d, D) = TF(t, d) \cdot IDF(t, D)$$

The idea behind TF-IDF is that words that are more common in a given document are considered more important, while words that are common across all documents are considered less important. This allows the representation to capture the relevance of a word to a particular document, while also taking into account the overall context of the corpus. This concept is especially important, as words with high frequencies across all documents (e.g. „the“, „a“, „more“, etc.) are being penalised and considered less relevant.

This form of representation is often used in NLP tasks such as document classification, clustering, and search, as it can provide a more accurate representation of a document's content compared to other methods that only consider the raw frequency of words.

Neural Representation

Distributive representation of documents using neural autoencoders refers to the process of learning a low-dimensional, dense vector representation of a document while preserving its syntactic and semantic information. The motivation behind this approach is to overcome the limitations of traditional bag-of-words models, which fail to capture the meaning and context of a document [22].

The architecture of a neural autoencoder for document representation consists of two main components, an encoder and a decoder. The encoder maps the input document to a low-dimensional vector and the decoder generates a reconstruction of the input document from this vector. The training objective of the autoencoder is to minimise the reconstruction error between the input document and the output generated by the decoder. Mathematically, this can be formulated as minimising the following objective function:

$$\min_{\theta} |D - g_\theta(f_\theta(D))|^2,$$

where θ are the parameters of the encoder and decoder, $f_\theta(D)$ is the encoding function, and $g_\theta(f_\theta(D))$ is the decoding function. Note that concrete implementations of both encoder and decoder parts can vary depending on the selected neural network architecture and the form of input data. Figure 2.4 shows the general high-level overview scheme of this architecture.

The training can be done in an unsupervised fashion, hence the corpus does not need to contain annotations for input data. Once trained, the autoencoder can be used to obtain distributed representations of new documents by passing them through the encoder and using the resulting vector as a representation of the document. This representation can later be used for various NLP tasks, such as classification or summarization.

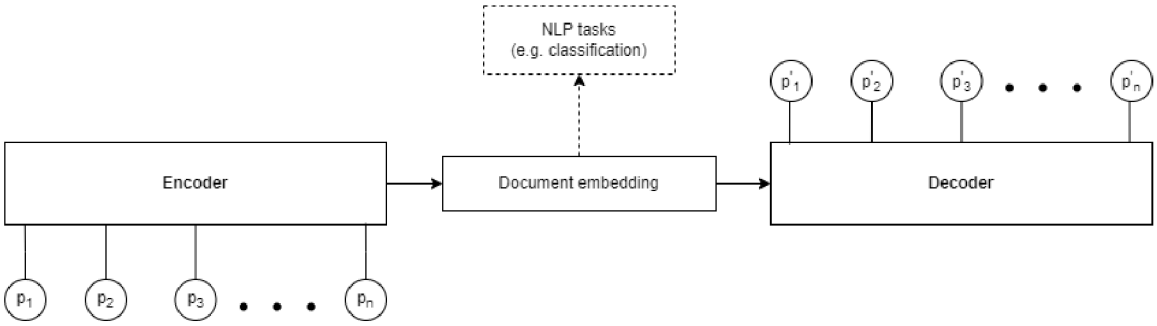


Figure 2.4: The general encoder-decoder architecture, where p_n is an element from the input sequence in the form of word/paragraph embedding vector and $p_n^{'}$ stands for the predicted word/paragraph embedding vector from the decoder.

2.4 Artificial Neural Networks for NLP

Artificial neural networks (ANNs) have become a popular tool in natural language processing due to their ability to automatically learn complex patterns in data. ANNs are a type of machine learning model that is inspired by the structure and function of the brain, consisting of a large number of interconnected processing nodes, or neurons. In NLP, these neurons can be trained to process and analyse linguistic data, such as words, sentences, and paragraphs, to perform tasks such as language translation, text classification, or sentiment analysis. In this chapter, we will explore the basic principles of ANNs and how they can be applied to NLP tasks.

Neuron

A neuron, also known as a „node“ or „unit,“ is a basic building block of a neural network. It receives input from other neurons, processes this input using a set of mathematical operations, and produces an output that can be passed on to other neurons or used as the final prediction of the network [1]. Neurons are organised into layers, with the input layer receiving the raw data and the output layer producing the final prediction, and the layers in between being called „hidden layers“. The computation performed by a node is typically a simple mathematical operation, such as a dot product of the input with a set of weights and an optional bias term, followed by an activation function. The base scheme is shown in Figure 2.5.

Activation Function

The activation function is a fundamental component of artificial neural networks. It is a mathematical function that is applied to the output of each neuron in the network, and its primary purpose is to introduce non-linearity into the network, which is crucial for the network's ability to learn and model complex patterns in data [1]. Common examples of activation functions that are widely used in neural networks include the hyperbolic tangent function, the sigmoid function (Figure 2.6b), and the rectified linear unit (ReLU) function (Figure 2.6a). Sometimes, the activation function is also known as the squashing function, as it can be used to limit the amplitude of the output [12]. These functions are chosen based on the specific requirements of the network and the nature of the input data.

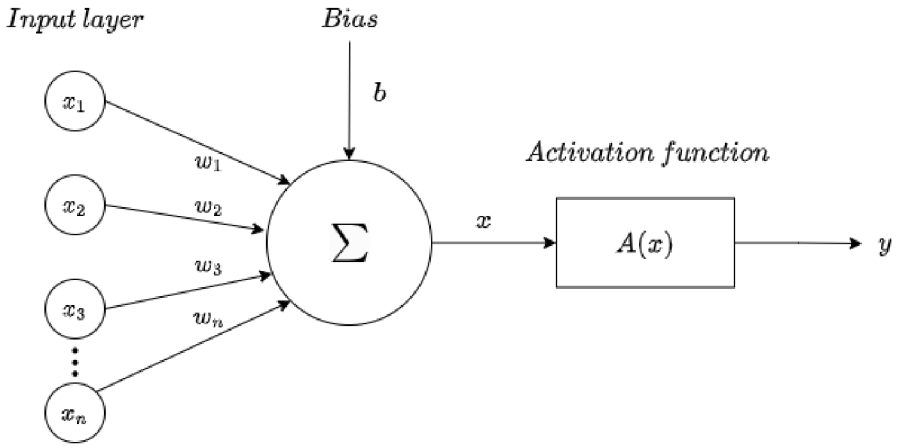
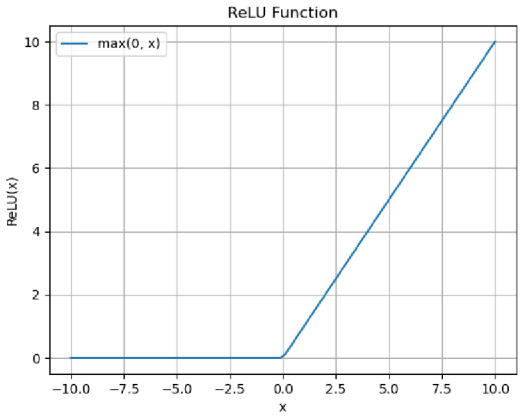


Figure 2.5: An example of a single neuron, the base unit of neural network models, where x_n are input values (they could be produced by other connected neurons), w_n are weights, x is an element-wise product $(\sum_{i=1}^{n} x_i w_i) + b$. The activation function is denoted as A and its output as y.

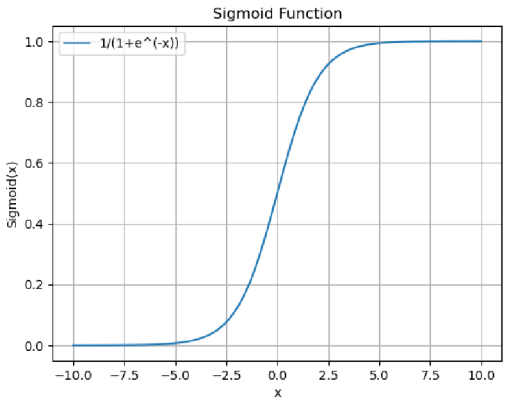
Training Process

The training process of a neural network involves adjusting the weights of the network to make predictions that are as accurate as possible on a given data set. This is typically done using a backpropagation algorithm [1]. This algorithm starts by performing a forward pass through the network, where the input is passed through the network to obtain the predicted output. The error (also called loss) is then calculated by comparing the predicted output with the desired output, also called the „ground truth“. The loss is then propagated back through the network and the weights are adjusted in order to reduce the overall error. This process is repeated for a number of iterations until the error converges to a minimum.

There are many error functions that can be used for various tasks. For example, one of the typically used error functions for a regression problem can be the mean squared error (MSE) function. MSE measures the squared Euclidean distance between the predicted and desired output and is defined as



(a) ReLU Function



(b) Sigmoid Function

$$E_{MSE} = \frac{1}{2n}\sum_{i=1}^{n}(y_i - t_i)^2$$

where n is the number of training examples, y_i is the output predicted for the i^{th} training example and t_i is the desired output for the i^{th} training example.

The weights are adjusted using the gradients of the error function with respect to the weights, computed using the chain rule of differential calculus. Gradients are computed in the backward direction, starting from the output node, which is called the backward phase of the algorithm. The detailed description of the entire gradient computation process can be further studied in [1].

Once the gradients are computed, it is necessary to somehow update the weights. Taking into account the simple perceptron, whose parameters correspond to $\overline{W} = (w_1, ..., w_n)$, the simple technique called gradient descent can be applied as follows:

$$\overline{W} \Leftarrow \overline{W} - \alpha(\frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_n}) \Leftrightarrow \overline{W} \Leftarrow \overline{W} - \alpha\frac{\partial E}{\partial \overline{W}}$$

where E is an error (loss) function and α is a learning rate parameter that determines the significantness of the updates. The notation above shows an update for only one neuron in the entire network, and to update the entire model, this update has to be done for all neurons.

Transfer Learning and Fine-tuning

Transfer learning refers to the idea of using knowledge or information learnt by a model on a task to improve the performance of the model on a different but related task. This can involve using the weights and biases of a pre-trained model as the starting point for training a new model on a different data set, or it can relate to using the learnt features from the pre-trained model as input to a new model. Transfer learning can be a powerful tool in deep learning since it allows models to leverage the knowledge learnt from previous tasks to improve their performance on new tasks and can often lead to better results than training a model from scratch. This approach can also be used when annotated data for certain domains are not available, but highly effective models for the same task have already been trained in different domains [2].

Fine-tuning is a type of transfer learning that refers to the process of adapting a pre-trained model to a new task or data set, by adjusting the weights and biases of the model and adding additional layers to the network. Fine-tuning is a common approach in deep learning, where pre-trained models are often used as the starting point for developing new models since it can save a significant amount of time and resources compared to training a model from scratch. There is a broad underlying theory on transfer learning that can be further studied in [34].

Architecture Types

There are a few popular types of neural network architecture that can be used for NLP tasks, and the best one to choose can depend on the specific problem to be solved or the environment in which it has to work. For NLP tasks, in general, the input of the neural network is commonly a sequence of tokens or vectors (embeddings). For this purpose, architectures capable of processing sequences were designed, and nowadays some of the most used architectures are [40]:

1. **Recurrent neural networks** (RNNs), which are well suited to processing sequential data, such as text. RNNs can be unrolled in time to process long sequences and can include variations such as long-short-term memory (LSTM) and gated recurrent units (GRUs) that help improve their ability to learn long-term dependencies.

2. **Convolutional neural networks** (CNNs), which are designed to process data with a grid-like structure, such as images or text represented as a sequence of words in a sentence. CNN's use convolutional layers, which apply a set of filters to the input data, to extract local features and create a hierarchical representation of the data.

3. **Deep contextualised language models**, which try to capture the context of the entire input sequence. One of the proposed methods was ELMo (Embeddings from Language Model), which was constructed from bidirectional LSTMs to produce context-rich embeddings [40]. Another successful proposal was BERT (Bidirectional Encoder Representations from Transformers), which is a type of neural network architecture based on self-attention mechanisms. These types of language model are considered highly universal, and their pre-trained variants are often fine-tuned for many different downstream tasks [4].

Recurrent Neural Networks

A recurrent neural network is a type of artificial neural network that is well suited for processing data with sequential dependencies, such as time series or natural language. Unlike a traditional feedforward neural network, which maps a fixed input to a fixed output, an RNN takes a variable-length input and produces a variable-length output, allowing it to make predictions for various sequence-to-sequence tasks, such as machine translation or text generation [1].

At the core of an RNN is the recurrent unit, which processes a single element of the input sequence and maintains a hidden state that captures information about the elements that have been processed so far. This hidden state is passed from one recurrent unit to the next, allowing the model to use the information it has learnt from previous input elements to make more informed predictions about the next element in the sequence.

One of the key advantages of RNNs is that they can learn temporal dependencies in the data, allowing them to make predictions based on information that is far away from the current input element. This makes them well-suited for tasks such as language modelling, where the meaning of a word can depend on the words that come before and after it in a sentence. In the standard RNN architecture, the information is passed forward, meaning that one input in a sequence has context only from its predecessors. To also gain context from successors, a modified version of RNN called Bidirectional RNN can be used to capture context from both sides. However, this modification greatly increases the computational time needed to perform both passes throughout the network.

There are also some difficulties when it comes to training. One of the main challenges with the training process is the problem of vanishing/exploding gradients, where the gradients computed during backpropagation tend to become smaller/greater with each time stamp as they are propagated back through time, making it difficult for the network to learn long-term dependencies [1]. This can be addressed using techniques such as gradient clipping or long-short-term memory (LSTM) units, which use gates to control the flow of information through the network and help prevent the gradients from vanishing.

Convolutional Neural Networks

A convolutional neural network (CNN) is a type of artificial neural network that is designed to process data with a grid-like structure, such as an image or a sequence of words. CNNs use convolutional layers, which apply a set of filters to input data, to extract local temporal features [1].

The convolutional layers operate on small patches of the input data, applying the same set of filters at each location to create a feature map. These feature maps are then combined using a nonlinear function, such as ReLU, to create a new set of feature maps that capture more complex patterns in the data. This process is repeated multiple times, each successive layer of CNN learning to extract more abstract and higher-level features from the input data. In the context of NLP, processing text data in this way can be called the N-Gram feature detector [48], as the convolutional filter can be viewed as a local context window.

In addition to convolutional layers, CNNs also typically include pooling layers, which are used to downsample feature maps and reduce the spatial dimensions of the data. This can help reduce the computational complexity of the network and make it more robust to small translations or deformations in the input data.

Transformers

Transformers are a type of neural network architecture that was introduced in the paper „Attention is All You Need“ [43]. Unlike traditional RNNs, which process input sequentially in a way that is difficult to parallelise, transformers are more parallelisable, which makes them particularly well suited for tasks that involve large-scale training.

At the core of the transformer architecture is the self-attention mechanism [43] which allows the model to „attend“ to different parts of the input sequence at different times, weighting the different input elements based on their relevance to the current output prediction. This allows the model to learn dependencies between input elements that are far apart in the sequence, and to create a more global representation of the input data.

In addition to the self-attention mechanism, transformer models also use multi-head attention [43], which allows the model to learn multiple different representations of the input data in parallel, and feedforward layers, which map the input data to a higher-dimensional space before making predictions.

BERT

Bidirectional Encoder Representations from Transformers (BERT) is a language representation model that uses unlabelled text data to produce embeddings, while capturing the context from the left and right directions of the input text [9]. It is based on the Transformer architecture and can be fine-tuned on a variety of downstream natural language processing tasks, such as question answering, sentiment analysis, and named entity recognition, to

achieve state-of-the-art performance [9]. In addition, several extensions of BERT have been developed that use different pre-training strategies such as RoBERTa [23], ALBERT [17], DeBERTa [13], ELECTRA [7] and others, to further improve its performance.

BERT uses the masked language modelling (MLM) objective during the pre-training process. The idea behind this objective is to randomly mask the input tokens by replacing the original token with a special `[MASK]` token, and let the model predict the original tokens according to the context [9]. Using this technique, the model can learn highly complex general language representations to generate embeddings that can be used to fine-tune the new model on any downstream task. In addition to MLM, the model is also jointly trained on the next sentence prediction (NSP) objective. At training time, the input sequence consists of two sentences, separated by the `[SEP]` token, while the second sentence may or may not be the following of its predecessor. The goal is to predict whether the second sentence follows the first. For this purpose, the special `[CLS]` token has been introduced, which is present at the beginning of each input sequence. The output representation of the `[CLS]` token can be used as input for downstream tasks that require a general encoded context of the entire input sequence, for example, sentiment analysis or categorisation task. More details about BERT can be found in [9], the original BERT paper.

2.5 Text Classification

Text classification is the task of assigning predefined categories or labels to a given text. It is a fundamental problem in natural language processing and has various applications, such as sentiment analysis, spam detection, topic categorisation, or named entity recognition. In general, there are multiple scopes for classification, as the task can be performed at the following levels [16]:

1. **Document level**: In document-level classification, the task is to assign categories or labels to a given complete document. For example, documents can be classified by its type to distinguish among academic articles, news articles, legal documents, etc.

2. **Paragraph level**: In paragraph-level classification, the task is to assign categories or labels to a given paragraph within a document. Examples include classifying customer reviews or identifying the main topic of a paragraph within an article.

3. **Sentence level**: In sentence-level classification, the task is to assign predefined categories or labels to a given sentence within a paragraph. This context could be used for use cases such as sentence-level grammar checking.

4. **Sub-sentence level**: In sub-sentence level classification, the task is to assign categories or labels to specific parts of a given sentence or the tokens themselves. Examples include named entity recognition, part-of-speech tagging, and coreference resolution.

The task can be divided into multiple pipeline stages that are followed to create a model. General steps include text preprocessing, feature extraction, classification, and evaluation. The following is a brief overview of each step in the pipeline [16]:

1. **Text preprocessing**: This step involves cleaning and preparing the text data for further processing. The techniques used in this step include removing stop words, stemming, and lemmatisation, and removing special characters and numbers.

2. **Feature extraction**: This step involves extracting features from the text data that will be used for classification. Techniques used in this step include bag-of-words, term frequency-inverse document frequency (TF-IDF), and word embeddings such as word2vec and GloVe. In state-of-the-art approaches, the encoder part of a transformer can be used to extract these features.

3. **Classification**: This step involves using the extracted features to classify the text data into different categories. Classical techniques include decision trees, the Naive Bayes Classifier (NBC), the Support Vector Machine (SVM), or the k-nearest neighbours clustering algorithm. Deep neural methods include convolutional neural networks (CNNs), recurrent neural networks (RNNs), or transformer-based architectures, such as BERT or RoBERTa with a classification layer (head).

4. **Evaluation**: This step involves measuring the performance of the classification model using metrics such as precision, recall, and the F1 score, which are common metrics for any classification task.

The following sections will discuss in more detail the relevant approaches we used in our work with respect to the specific tasks we focused on. The problem domains described in Sections 2.6 and 2.7 are specific cases of text classification.

2.6 Content Categorisation

In this chapter, we describe the common methods for the content categorisation task. The first and second methods are a common approach to classification in many domains that work with artificial neural networks, while the third is unique to NLP.

Softmax Classifier

The softmax classifier is a commonly used classification method in machine learning that involves combining a neural network model (in our use case, the encoder) with a classification head that uses a softmax function to produce the final prediction. The encoder is responsible for transforming the input data into a lower-dimensional representation, while the classification head is responsible for predicting the class labels based on the encoded features. The encoder can be any neural network architecture, but we focus mainly on state-of-the-art encoders based on the BERT architecture described in Section 2.4.

Once the encoder has been pretrained, a classification head is added to the network. The classification head typically consists of a fully connected layer followed by a softmax activation function, however, the head itself can be even more complex neural network on its own. The fully connected layer takes the encoded features as input and maps them to a set of class probabilities, while the softmax function normalises the class probabilities to ensure that they sum to 1. The class that has the highest probability is then considered the final prediction. More information on softmax itself or other functions used for output distributions can be found in [11].

During training, the optimiser adjusts the weights of the model according to a cross-entropy loss function, which measures the difference between the predicted class probabilities and the true class labels. There are two important properties of this type of classifiers to consider:

1. **Fixed labels**: The model will learn to predict a concrete number of labels depending on the size of the final output layer.

2. **Supervised training**: In order to train this type of classifier, a suitable data set has to exist in the desired domain that includes examples for all the required classes.

Sigmoid Classifier

The sigmoid classifier is a classification method that can be used when a sample can belong to multiple classes at the same time. The methodology is the same as with the softmax classifiers, as the difference is in the handling of the final output layer of the classification head. Unlike softmax, each label is treated independently and a separate sigmoid activation function is used for each class to produce a probability that the class is present. During training, the optimiser adjusts the weights of the model according to a binary cross-entropy loss function for each class. Note that this approach can be easily converted to a simple binary classification problem, when the output contains only a single class.

Natural Language Inference

Natural Language Inference (NLI) is a fundamental task in natural language processing that involves determining the relationship between two given text fragments. The goal of NLI is to infer whether the relationship between the two text fragments is one of contradiction, entailment, or neutrality.

The two given text fragments are typically referred to as the „premise“ and the „hypothesis.“ The premise is the first text fragment that provides context or information, while the hypothesis is the second text fragment that contains an assertion or claim that is being evaluated in relation to the premise.

To perform NLI, a machine learning model is trained on a large data set of text pairs, each labelled with one of the three possible relationships. The model learns to identify the linguistic features and semantic relationships that are indicative of each relationship and then applies this knowledge to new pairs of texts to make predictions. The common data set used to train these models is the Standford Natural Language Inference (SNLI) Corpus [5], which includes triplets of the premise, hypothesis, and label for each entry. For each premise in the data set, there are a total of 3 premise-hypothesis pairs for each of the following labels: $(entailment, neutral, contradiction)$. In Table 2.1, there are all 3 hypotheses for the premise *„Two doctors perform surgery on patient.“* that has been taken directly from the SNLI[1] data set.

Hypothesis	**Label**
Doctors are performing surgery.	Entailment
Two doctors are performing surgery on a man.	Neutral
Two surgeons are having lunch.	Contradiction

Table 2.1: An example of the training sample taken directly from SNLI data set for premise „Two doctors perform surgery on patient.“.

If the hypothesis is cleverly constructed, the model can serve as a classifier to predict the categories. An example of a classifier made of this type of model can be seen in Figure

[1] https://nlp.stanford.edu/projects/snli/

2.7. However, this approach has some specific drawbacks caused by the properties of the training objective. Although the common softmax classifier would contain a part called classification head, which contains the outputs of all defined categories, the NLI method can output the prediction for only one category at a time. This leads to the fact that the classification of a single premise requires inference to be made N times, where N is the number of categories. This drawback is compensated for the ability to choose any desired label and even change categories at run-time when needed.

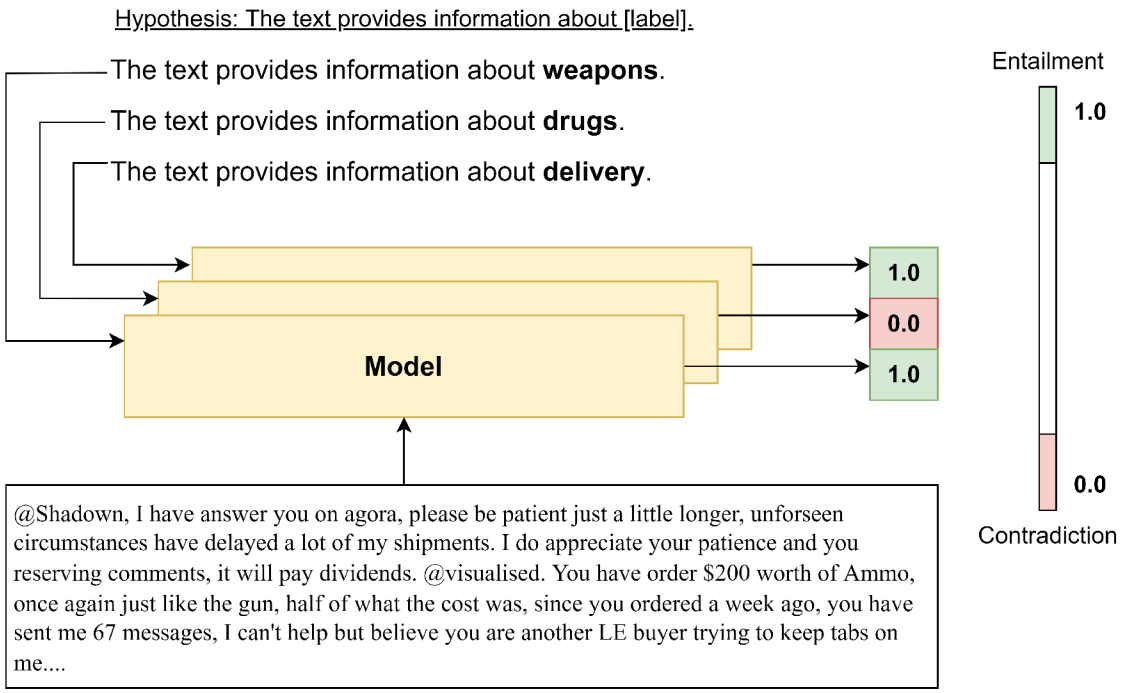


Figure 2.7: An example of a classifier made from the model trained on the NLI task.

2.7 Named Entity Recognition

Named Entity Recognition (NER) is a fundamental task in natural language processing that involves identifying and categorising named entities in text, such as names of people, organisations, locations, and other trained categories. It belongs to tasks recognised as a sequence tagging [14], which is a form of text classification performed at the sub-sentence level, as already divided in the section 2.5. Figure 2.8, visualises the example output of the NER model.

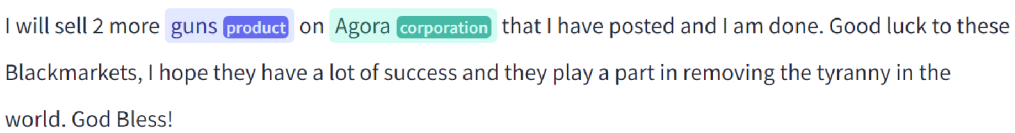


Figure 2.8: An example of NER performed on a piece of text from the darknet market domain. The words „guns“ and „Agora“ were classified as entities belonging to the *product* and *corporation* categories respectively.

The goal of NER is to correctly label each word or phrase in a given text with its corresponding entity category. This requires a machine learning model that can learn to recognise the patterns and features that are indicative of each category of entities. Such models typically use techniques such as conditional random fields (CRF) [14], support vector machines (SVM) [10], or deep neural networks, such as LSTM [14] or BERT [9] to perform classification. In our work, we focus mainly on state-of-the-art approaches that use models based on the transformer architecture, namely BERT.

2.8 Webpage Content Extraction

Webpage content extraction is the process of automatically identifying and extracting relevant content from web pages, which can include textual information, such as headlines, body text, and metadata, as well as multimedia elements, such as images, videos, and audio files. Extracted content can be used for a variety of applications, including web search, data analysis, and content curation.

Web pages are typically created using an XML-based markup language, such as HTML, and the structure of a web page is defined by the Document Object Model (DOM), which represents the page as a hierarchical tree of elements. To extract content, various techniques and tools are available, such as regular expressions, XPath, CSS selectors, and machine learning-based approaches. In this chapter, we explore the fundamentals of how webpages are defined and structured and what tools are available for extracting their content.

HTML

HTML (HyperText Markup Language) is a markup language that is used to structure and format the content of web pages. It employs a system of tags and attributes to define various elements on a webpage, such as headings, paragraphs, images, and links. These elements can be styled and arranged using CSS (Cascading Style Sheets) to control the webpage's appearance and layout. Web browsers parse the HTML code to display its content to the user.

The language was first officially released in 1991 [47] as a way to format and structure the content of web pages. Since then, it has become the standard language for web development and is used by virtually all websites. The current standard is HTML5, which introduced numerous quality of life features such as audio and video support, new semantic tags, basic rendering with `<canvas>`, validation of forms, and more [25].

HTML consists of a set of tags and attributes that can be used to define different elements. These elements define different components of a webpage, such as paragraphs, images, and hyperlinks. Typically, an element includes a start tag, inner content, and an end tag. For instance, the `<p>` and `<div>` tags are examples that follow this pattern. However, there are also exceptions that do not contain any content and do not require an end tag. These are referred to as empty elements, such as the `<br>` and `<img>` tags [46]. Although the `<img>` tag is used to display an image, information about the image is provided through its attributes.

The attributes are used to provide more information about an element and always appear in the starting tag. Typically, an attribute is composed of a name/value pair [44]. For instance, the `<img>` tag requires a link to the image and an optional description, which appears instead of the image when it fails to load. An example can be seen in Figure 2.9.

```
<img src="../images/cat.png" alt="Cat sitting on a mat.">
```

Figure 2.9: An example of an HTML `<img>` tag that is used to insert an image into a webpage. The tag includes the `src` attribute that specifies the location of the image file. The `alt` attribute provides alternative text that is displayed if the image does not load.

Tags can be combined and nested to create complex and structured web pages, as shown in Figure 2.10. In addition to defining the structure and content of a web page, HTML can also be used to add interactivity using technologies such as JavaScript. This allows developers to create dynamic and interactive web experiences that can respond to user input and change over time. We will not describe these interactive elements any further as it is out of the scope of this work and irrelevant to the context of the content extraction problem.

```
<html>
  <head>
    <title>My Webpage</title>
  </head>
  <body>
    <h1>Header 1</h1>
    <p>This is some text in a paragraph tags.</p>
    <div>
      <ul id="myList">
        <li>List item 1</li>
        <li>List item 2</li>
        <li>List item 3</li>
      </ul>
    </div>
  </body>
</html>
```

Figure 2.10: A minimal example of a webpage defined by the HTML code. The `html` tag is a root element of the webpage that contains `head` and `body` child elements. The purpose of the `head` element is to provide metadata about the webpage, while `body` encapsulates the visible content to the user.

Document Object Model

The W3C (World Wide Web Consortium) Document Object Model (DOM) is a standardised way to work with XML-like documents and is supported by most modern web browsers and programming languages [37]. It defines a set of interfaces, methods, and properties that can be used to interact with the document. This standard can be separated into three parts:

- **Core DOM** - shared among all document types.
- **XML DOM** - specification for XML documents.

- **HTML DOM** - specification for HTML documents.

Since we are mostly interested in the content extraction from web pages, in the following section, we will focus particularly on the HTML DOM standard.

HTML DOM

HTML DOM is a programming API for HTML documents that defines all elements as objects and organises them into a tree structure. The root of the tree is the `document`, which contains all other nodes appearing on the webpage [37]. The HTML DOM standard defines three main categories [45]:

- **Properties** of all HTML elements.
- **Methods** to access all HTML elements.
- **Events** for all HTML elements.

Every element in the document is represented by a node in the tree, with an element node potentially having child nodes, which can be other element nodes or text nodes. The attributes of an element are also represented as separate nodes. The example of Figure 2.10 visualised as a DOM tree can be seen in Figure 2.11.

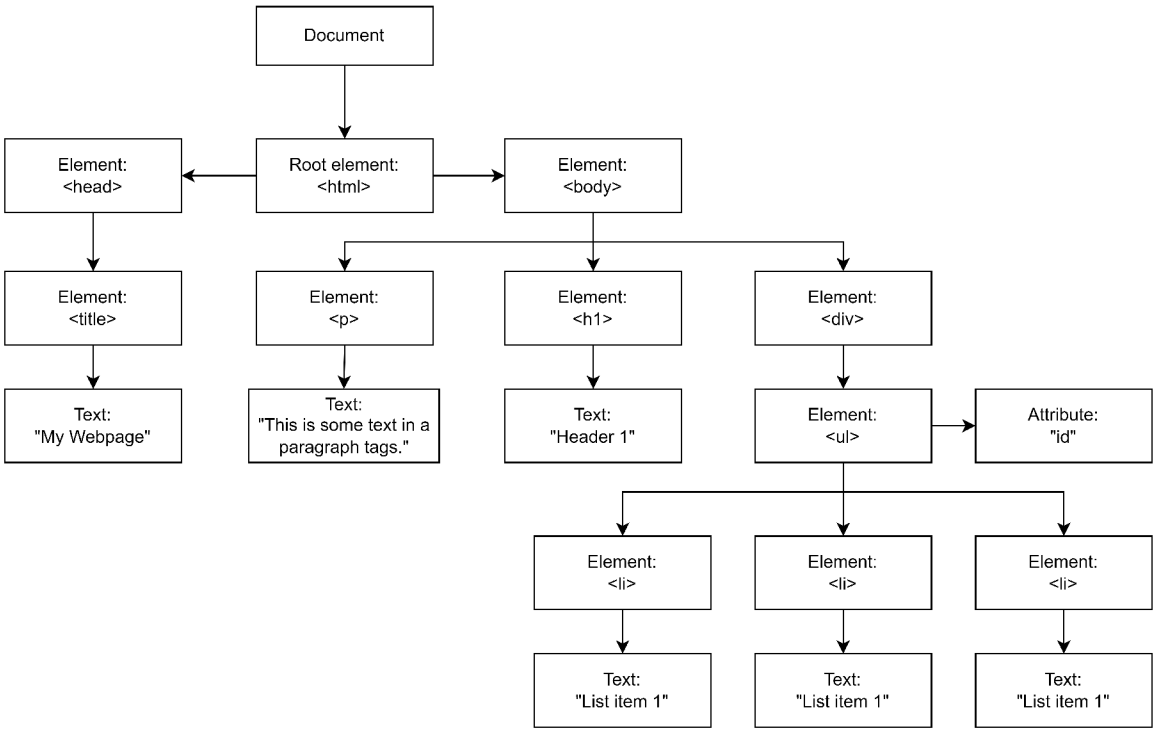


Figure 2.11: Visualisation of the HTML DOM tree, based on the code from Figure 2.10.

```
html/body/div/ul/li[position() > 1]
```

Figure 2.12: An XPath query that addresses all items except the first one in the list from Figure 2.10. The output of this query are two `<li>` elements with the values *„List item 2“* and *„List item 3“*.

XPath

XPath refers to the XML Path Language, which uses non-XML syntax to address various parts of the XML documents [31]. It can be used as a powerful way of navigating through the DOM of any XML-like language, including HTML. XPath uses a path notation, similar to URLs, to move through the hierarchical arrangement of an XML file. To enable it to be used in URIs and XML attribute values, it adopts a syntax that is different from XML itself [31].

XPath also has a number of useful features that make it a popular choice for XML processing. For example, it supports a wide range of operators and functions and makes it easy to perform complex XML data queries. An example can be found in Figure 2.12.

Available Tools

Various tools are available for parsing and scraping websites using almost every modern programming language. Typically, these tools allow access to web pages, parsing them into a DOM tree following the W3C standard, and providing useful methods for searching and extracting content from the DOM. As Python programming language is widely used for deep learning, we have decided to focus mainly on the technological stack based around the Python ecosystem. The following libraries can be used to parse web pages in the Python programming language:

- **BeautifulSoup**[2]: A Python library to pull data from HTML and XML files. It is a simple and powerful tool to extract data from web pages and is commonly used for scraping and data extraction. However, the `BeautifulSoup` library does not support XPath expressions for document navigation, which can be a disadvantage in certain scenarios.
- **html5lib**[3]: The pure Python library that is focused on the HTML5 standard, providing a simple and easy API for parsing up-to-date web pages.
- **lxml**[4]: Python library for parsing and processing HTML and XML documents. It is a binding of the `libxml2` and `libxslt` C libraries and offers a powerful and highly efficient way to extract data from web pages.

All of these libraries provide useful features for parsing web pages in Python. However, given the requirements for high processing speed and XPath support, most operations regarding the DOM manipulation will be done using the `lxml` library. The explanation of the need for XPath support will be further discussed in later chapters.

[2] https://beautiful-soup-4.readthedocs.io/en/latest/
[3] https://html5lib.readthedocs.io/en/latest/
[4] https://lxml.de/

Chapter 3

System Design

Until this point we have covered key topics that are necessary to build the automatic content categorisation and extraction system of web pages. In this chapter, we outline the basic structure of the proposed system. When designing the system, it is important to keep in mind the flexibility, maintainability, and performance of the system, as it should handle the wide range and amount of web pages with different structure and content types.

3.1 Requirements Analysis

The main purpose of the system will be the categorisation of content from web pages, specialised in darknet forums and marketplaces. Content analysis will be performed at the level of posts or comments, and extracted entities and topic categories will be stored in the database. While crawling these websites on the TOR network can be a difficult task by itself, the system will not be responsible for scraping web pages, but will work mainly with already crawled and archived data. However, it is important to allow for a possible extension of the system in this regard.

The archived forums typically contain a series of snapshots from different time periods and many files, each representing a webpage accessible from any other element in the website structure. Given the nature of the data, the system must be scalable to process such quantities of data in a reasonable time. Furthermore, many pages crawled in this way will not be useful for the analysis, as not all pages contain the interaction between users or the content of interest. In fact, when analysing large-scale archives such as Darknet Market Archives[1], most of the files produced by the crawler are not usable for this task at all.

For the analysis task, state-of-the-art deep learning NLP methods will be used. However, these methods are highly computationally expensive and require a significant amount of resources. Therefore, it is necessary to plan the distribution of the computing power wisely and split the entire processing pipeline into the following phases:

1. **Finding valuable files**: The system will iterate over each file and mark potentially valuable files for further processing.

2. **Processing valuable files**: Files marked as valuable will be further subjected to a more complex and expensive analysis.

These phases could be viewed as two separable map-reduce tasks, which would allow the system to use computational resources in an effective and distributive manner.

[1] https://www.gwern.net/DNM-archives

3.2 General Architecture

The general architectural pattern used to build this system is client-server architecture. This pattern allows for the construction of highly flexible and performant applications, which can be easily deployed in both local environments and cloud environments with scalable computing resources on demand. The basic scheme of this architecture can be seen in Figure 3.1.

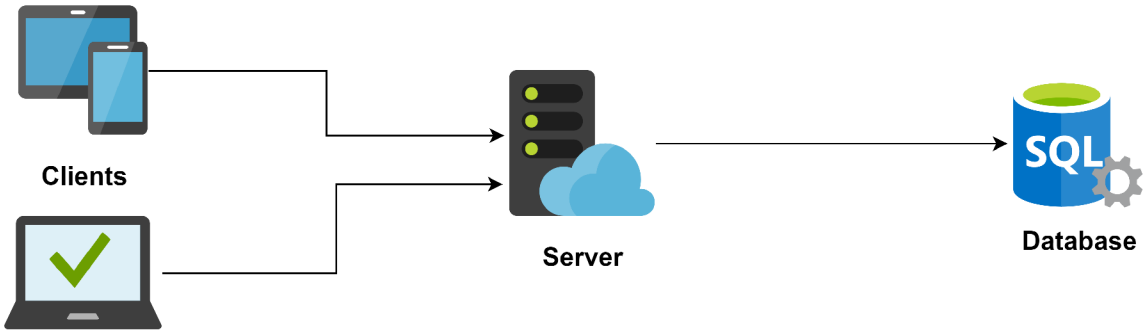


Figure 3.1: Client-server architecture.

Microservice Architecture

One of the modern architectures based on the client-server pattern is a microservice architecture, also known as microservices. Microservices is a structure that models an application as a series of services that can be independently deployed, are loosely coupled, are organised around business capabilities, and are owned by a small team [36]. Each of these services can be viewed as a small application that is responsible for a single function or process that can be easily described and understood. These services can communicate with each other to cooperate and perform more complex tasks. Communication is typically done through well-defined APIs[2] over the network [39]. An illustrative example of an application composed of multiple microservices can be seen in Figure 3.2.

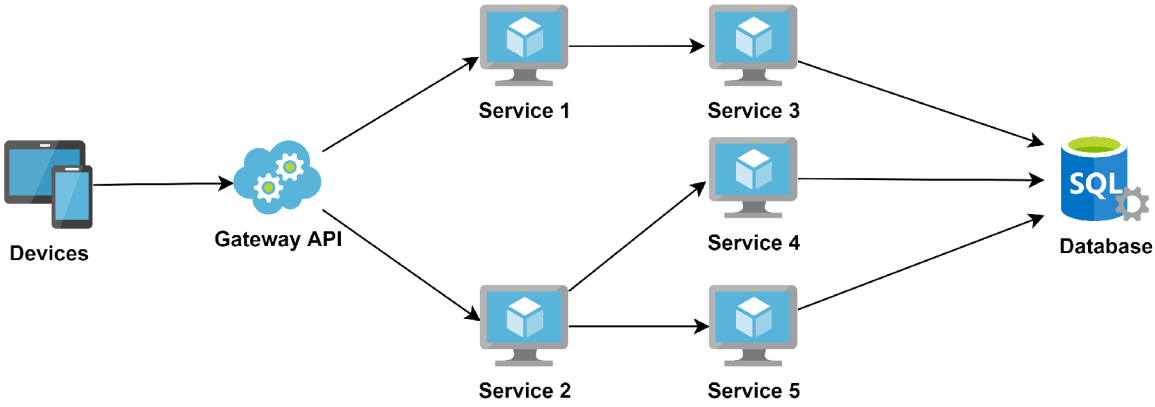


Figure 3.2: Illustrative example of a microservice architecture.

[2]Application Programming Interfaces

We propose using this architecture to build our system, because it is easily maintainable, scalable, and can be deployed anywhere with the help of containerisation technologies, such as Docker[3].

The initial design of the system modelled as a component diagram can be seen in Figure 3.3. At a high level, the system can be viewed as a collection of multiple easily manageable components, where each component has clear and simple responsibilities for certain tasks and uses other components to achieve a common goal. The analogy with the microservice architecture is naturally achieved as each component can be seen as a single independent service.

1. **Client**: The client application is responsible for sending requests to the back-end server and visualising the processed data to the user. The client can be almost anything that can construct an HTTP request and send it over the network to the API.

2. **Backend Server (API)**: The back-end server can serve clients, providing them with data, and accepting requests for processing.

3. **Coordinator**: The coordinator is responsible for managing the workload and the workers. Data will be split into batches and sent to workers for processing. After processing is complete, the coordinator must retrieve all the results and ensure that the results are stored in the database.

4. **Worker**: Workers take orders from the coordinator and use the NLP engine with the parser to perform content categorisation and data extraction.

5. **NLP Engine**: Provides access to deep learning models for content analysis and other features based on NLP methods.

6. **Parser**: Parse files into a manageable format suitable for further analysis.

7. **Database**: Handles requests for data storage and serves as an information provider for the back-end server and his clients.

3.3 Database Structure

The database is a key component of the system and must be versatile enough to handle a variety of web page data. The data will be almost exclusively textual content that is included within the elements that make up the entire web page. These elements can be grouped together in a free way to compose more complex and meaningful pieces of content. For example, if the subject of interest is forum posts, the main content of each post will most likely be composed of the author's username, title, and message. In general, any set of elements can represent an atomic piece of content, which means that the user must not be limited to the content structure or format to be parsable by the system itself. Furthermore, not all elements may contain data that contribute to categorisation analysis in any way. For example, the username does not include information relevant for categorisation, but can be useful for filtering content based on their authors.

[3] https://www.docker.com/

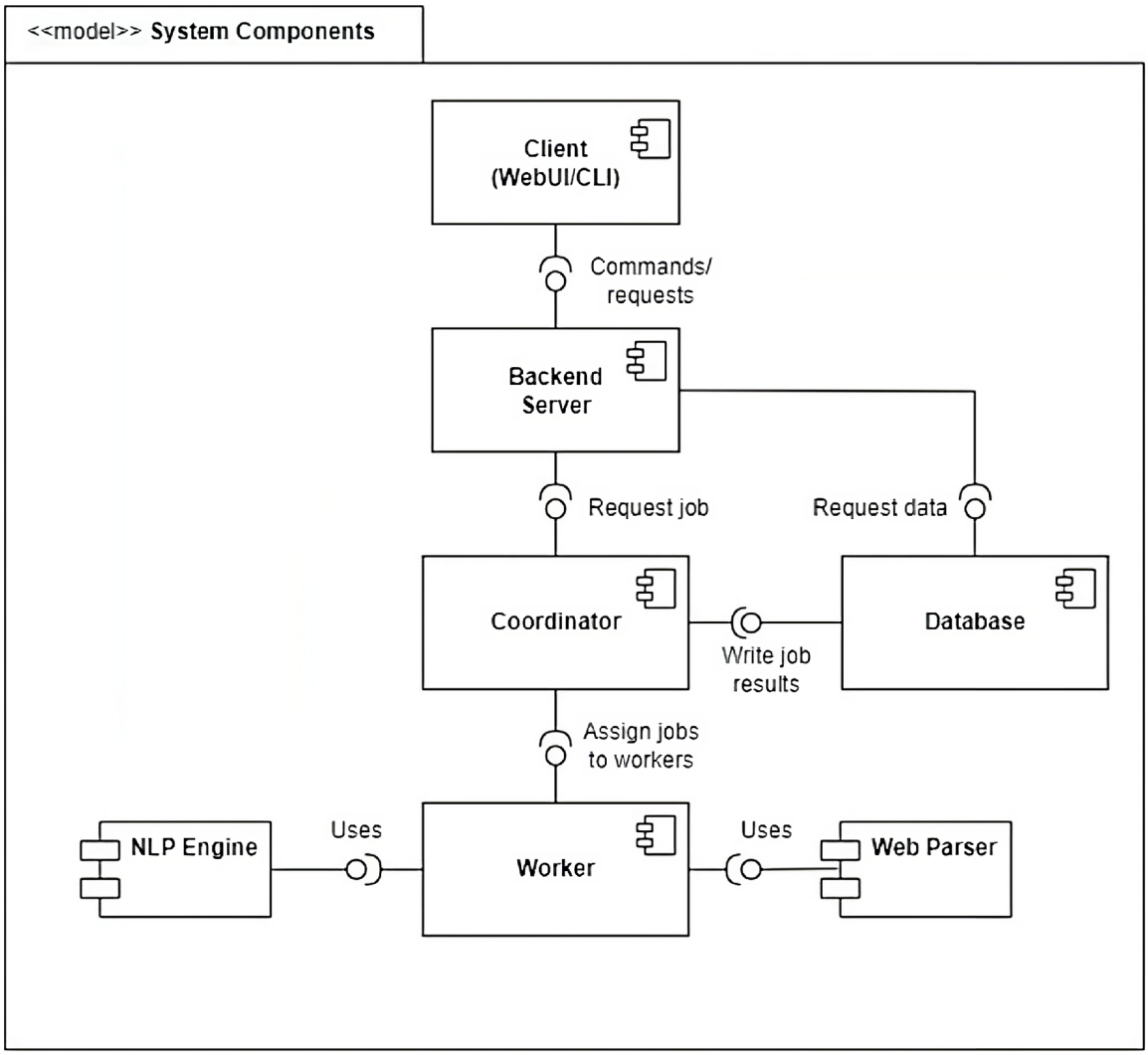


Figure 3.3: The overall architecture of the proposed system, modeled as a component diagram.

Entity Relationship Diagram

To achieve the required properties, the Entity Relationship Diagram 3.4 has been proposed to serve as the basis for the database structure. Based on the requirements of the system and the database, we have identified the following entities:

1. **Content**: In terms of the system, it is the main unit of information. The `content` entity groups all elements of interest together and links them to a specific file. Furthermore, the property `hash` is introduced to prevent storing redundant pieces of content that were already parsed and analysed in the past. The `hash` should be calculated from the overall structure and textual content of all linked elements. This can potentially happen while parsing large web archives, which contain snapshots of the sites in a time series.

2. **Attribute**: A single attribute is equivalent to the node (or a subtree) in the DOM tree structure. The main properties of an attribute entity are `content` and `tag`. The

`content` property stores the textual content extracted from the node, and `tag` is an auxiliary property that can be used for different purposes, such as enumeration of paragraphs, analysis errors for the concrete element, etc. Each attribute can also have named entities and categories associated with them, and it knows its type determined by the `Attribute Type` entity.

3. **Attribute Type**: An entity that determines the type of `Attribute` entity. Properties `tag` and `name` represent machine-friendly and user-friendly names, respectively.

4. **Named Entity**: The named entity represents a concrete entity found within some attribute's content. It is a product of the analysis process called Named Entity Recognition. Each named entity belongs to an attribute entity and knows its type.

5. **Named Entity Type**: The type of named entity defined by the properties `name` and `tag`, representing user-friendly and machine-friendly names, respectively.

6. **Category**: Available category in which the attribute can be categorised. The relationship between `Attribute` entity and `Category` must contain the relation `Confidence Score` associated with a certain category, which emerges as a product of the categorisation analysis part.

7. **File**: Represents a file that has been parsed by the system.

3.4 Design Summary

So far, we have made important design choices for the system that aims to categorise the content of the web pages. Using state-of-the-art deep learning NLP methods, a user can perform content analysis and extract entities and categories that are stored in a database. The system works with data already crawled and archived, and the architecture is designed to be scalable to handle large amounts of data in a reasonable time.

The system architecture is based on the client-server pattern and the proposed design uses microservices. The system is composed of multiple components, each component having clear and simple responsibilities for certain tasks and using other components to achieve a common goal. Components include a client, a back-end API server, a coordinator, worker, NLP engine, parser, and database.

The database must be versatile enough to handle a variety of web page data, and the data will be almost exclusively textual content that is included within elements that make up the entire web page. These elements can be grouped together in a free way to compose more complex and meaningful pieces of content.

In conclusion, the proposed system is designed to provide a scalable, efficient, and flexible solution for content analysis and categorisation of darknet forums and marketplaces, or any other web page content from other domains. The microservice architecture is ideal for this task, and the proposed design provides a clear and simple approach to achieving the goals of the system.

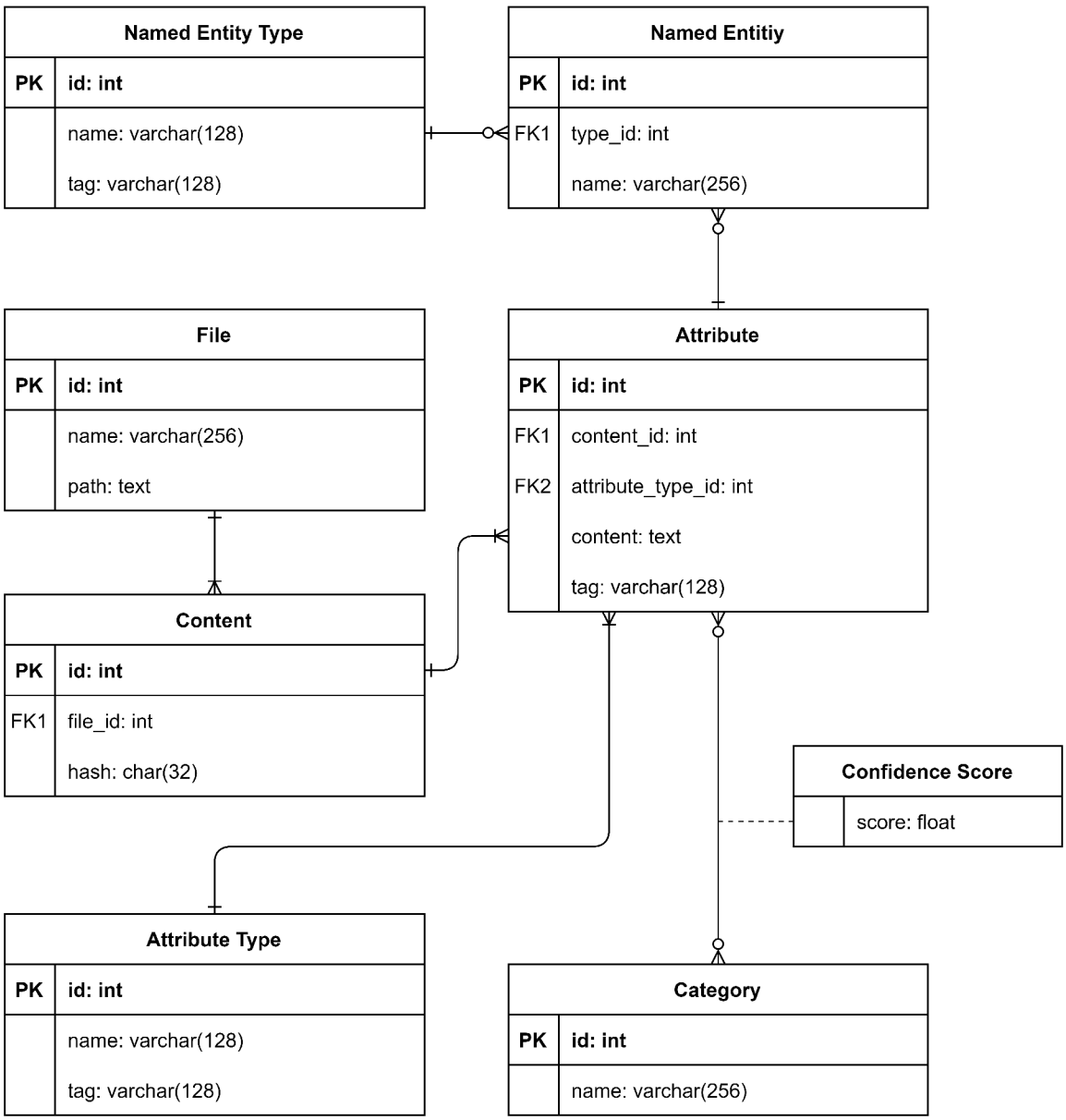


Figure 3.4: The entity relational diagram for the system database.

Chapter 4

System Implementation

In this chapter, concrete implementation choices are described in more detail, covering minor changes and the expansion of the system capabilities. The system is implemented in an object-oriented manner, following the object oriented principles, which allows for the usage of the object-oriented design patterns. These patterns are used to implement the components of the system, as they help to keep the code simple and reusable.

4.1 Environment and Tools

The main programming languages used for implementation are Python and Typescript. Python is used for the server-side implementation, while Typescript is used for the client-side implementation. The server-side implementation is based on the `Flask`[1] framework, while the client-side implementation is based on the `React`[2] framework. We use an open-source relational database implementation called `PostgreSQL`[3], and the database is accessed using the `SQLAlchemy`[4] object relational mapping (ORM), which is a technique that maps data between an object-oriented programming language and a relational database. To make the development process feasible in an iterative manner, we use `Alembic`[5] tool for database migrations to enable comfortable upgrade and fix of the database scheme.

For deployment and orchestration, `Docker` is used to containerise the system components and `Docker Compose` to orchestrate the containers. This enables the system to be deployed on any machine that has `Docker` installed and allows the system to be orchestrated as a microservice architecture, where each component is deployed as a separate container.

Regarding the versioning of the source code, we use `Git`[6] to manage and track changes made to our codebase. `Git` is a popular distributed version control system that allows efficient collaboration and easy management of different code versions. Our Git repository is hosted on `GitHub`[7], which is a `Git` hosting platform that provides a variety of useful features for managing software development projects.

[1] https://flask.palletsprojects.com/
[2] https://react.dev/
[3] https://www.postgresql.org/
[4] https://www.sqlalchemy.org/
[5] https://alembic.sqlalchemy.org/en/latest/
[6] https://git-scm.com/
[7] https://github.com/

4.2 Project Structure

The project directory consists of the following main files and directories:

- **README.md**: Includes additional information about the project and files or directories that are not specifically listed here.
- **config.py**: A configuration file used to initialise the `Flask` application. It is shared across all microservices to provide a connection string for the database.
- **docker-compose.cpu.yaml** and **docker-compose.gpu.yaml**: A Docker Compose files for running the project on a CPU and GPU machine, respectively.
- **docker-compose.cpu.debug.yaml** and **docker-compose.gpu.debug.yaml**: A Docker Compose files to run the project on a CPU and GPU machine with the support for external debugging. This allows for easy and comfortable attachment of any external debugger (for the `VS Code` IDE, see the `.vscode/launch.json` configuration with already predefined attach options for all services).
- **webcat**: A package containing the code for the main logic of the system. It includes sub-packages for the `analyser`, `database`, `model_repository`, `parser`, `pipeline` and `template_engine`. This package is available to all services.
- **webcat_api**: A directory containing the code for **gateway API**.
- **webcat_client**: A directory containing the code for **web client**.
- **webcat_scheduler**: A directory containing the code for **scheduler service**.
- **webcat_worker**: A directory containing the code for **worker service**.
- **webcat_templates**: A directory containing the code for **templating service**.

Every directory with the webcat prefix is a fully independent implementation of a microservice running in the `Docker` environment. The package `webcat` itself is distributed among all these services, except `client`, which provides access to the implementation of the business logic for each service. The `client` is implemented in the texttttReact[8] framework, while `api`, `scheduler` and `worker` are `Flask`[9] applications. It is worth mentioning that the `PostgreSQL`[10] database runs together with other services in the Docker environment, and its image is pulled from the Docker image registry while the project is being orchestrated by the `Docker Compose`[11] tool. The running application scheme can be seen in Figure 4.1. Note that given the properties of the microservice architecture, multiple workers can be instantiated to fully occupy available computational resources, which could be beneficial in powerful cloud computing environments.

[8] https://react.dev/
[9] https://flask.palletsprojects.com/en/2.3.x/
[10] https://www.postgresql.org/
[11] https://docs.docker.com/compose/

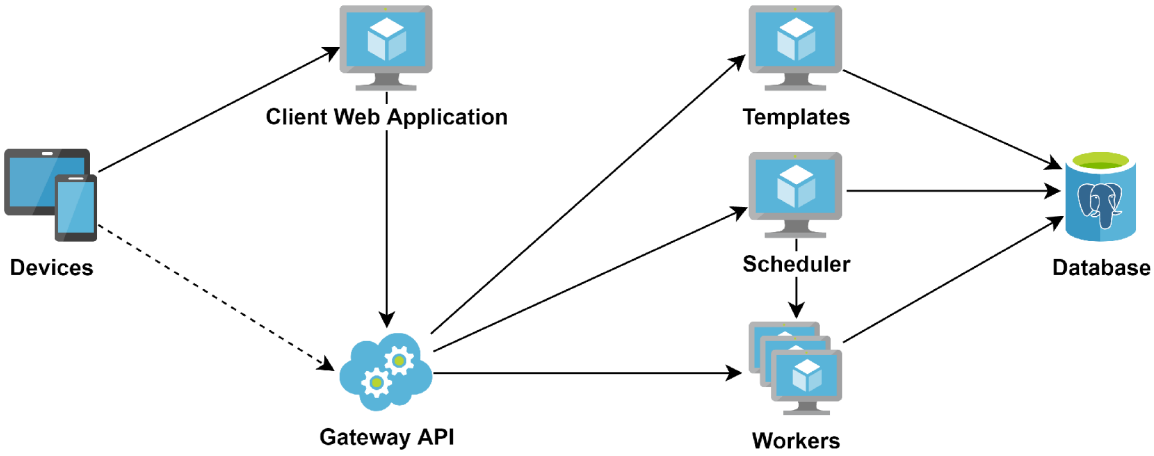


Figure 4.1: Scheme of the fully orchestrated WebCat application.

4.3 Services

This section will focus on each individual service and describe its purpose, responsibilities, and cooperation with other services. Each service exposes the REST API, providing unique functionalities for other services. Data exchange between services is always done with JavaScript Object Notation (JSON) payloads.

As mentioned above, all services are run separately as containers in a `Docker` environment. The base images used to build the containers are as follows:

- `python:3.11-slim`[12]: Used by all `Flask` applications.
- `node:18.15`[13]: Used by the `React` web client service.

Gateway API

The gateway API is the main and only entry to the features of the system, as it is exposed by the `Docker` environment to the public. As illustrated in Figure 4.1, the Gateway API service communicates with all other services to achieve the desired results requested by clients. In the majority of cases, this application serves as a proxy server whose responsibility is to forward requests to other services hidden within the virtual network of the containerised environment.

Currently, the API also provides direct access to processed content stored within the database with the various filtering options. However, this functionality is planned to be moved into its separate service, as it violates the facade principle, which we would like to fully enforce eventually in the near future.

Interface

As was mentioned before, the gateway API serves as a proxy for most of the features. For that reason, this section is split into 2 parts. The first part contains the list of all endpoints that are only forwarded for other microservices and will be discussed further in more detail in the sections dedicated to the specific service they belong to. The second part covers

[12] https://hub.docker.com/_/python
[13] https://hub.docker.com/_/node

the unique feature for the gateway service itself, and thus is described in more detail. The gateway API provides the following HTTP proxy endpoints for clients to use:

- `POST /api/v1/webcat_interactive`
- `GET /api/v1/webcat_files_parser`
- `POST /api/v1/webcat_files_parser`
- `GET /api/v1/webcat_templates/templates`
- `POST /api/v1/webcat_templates/templates`
- `DELETE /api/v1/webcat_templates/templates`
- `POST /api/v1/webcat_templates/element_types`
- `GET /api/v1/webcat_templates/element_types`
- `DELETE /api/v1/webcat_templates/element_types`
- `POST /api/v1/webcat_templates/engine`
- `GET /api/v1/webcat_templates/engine`

As mentioned above, the API also provides access to the results stored within the database, while allowing for the construction of filtering queries.

`GET /api/v1/webcat_data_provider`

Description: Lists all categories and named entity types recognised by the system. This information can be useful for clients to retrieve all categories and named entity types that can further be used as filtering options.

Response: A JSON object containing the following keys:

- `categories`: An array of strings with names of all categories.
- `entity_types`: An array of strings with names of all named entity types.

`POST /api/v1/webcat_data_provider`

Description: Provides access to the results in the database and accepts filtering options for the required content.

Parameters: The request body should contain a JSON object with the following keys that specify the filtering options:

- `categories`: An array of the required categories. If no categories are specified, all categories will be considered.
- `cat_threshold`: A floating number value between 0 and 1 that sets the minimal confidence threshold assigned to a category to be retrieved by the system.
- `entity_types`: An array of all the required named entity types that must be present in the results. If not specified, these types will not affect the filtering.

- `file_paths`: An array of file paths from which the results should originate. A special symbol `*` can be used anywhere in the path as a wildcard, matching all files in the subpath.
- `authors`: Special option for filtering by authors. An array of authors should be specified, and all content results containing an attribute with subword `*author*` as the attribute's type tag will be matched against the list of authors. All names containing the author's name as a subword will be matched as well.

Response: A JSON array with serialised database results compliant with the specified filtering options.

`DELETE /api/v1/webcat_data_provider`

Description: Removes a result entry from the database.
Parameters: The request body should contain a JSON object with the following keys:

- `id`: An identifier of the removed content entry.

Response: Status code of the operation.

Worker Service

The responsibility of the worker service is to handle any of the following requests:

- **Interactive input**: The user can send a raw text input for analysis that is handled as a single content fragment and processed through the processing pipeline. This feature can be useful for testing different hypotheses or labels and verifying the desired outcomes in an interactive way.
- **Processing files**: The worker will process the given file or multiple files in the directory. The processing pipeline will adjust different parsing strategies depending on the file type.

The most important part of the worker is the processing pipeline, which has to be configured and initialised depending on the configuration included within the incoming request. The worker has been designed with an emphasis on flexibility and expandability, thus processing pipeline was split into multiple configurable sections that follows the strategy pattern of the object-oriented design principles. Due to this approach, different parsing strategies and models can be loaded and swapped for each request. In addition, other file types, models, or preprocessing steps can be easily implemented and integrated into the system. The processing pipeline and workflow are discussed further in Section 4.4.

Furthermore, the worker service can be instantiated as either a CPU-based or GPU-based service. Although both types are inter-changable and capable of processing all sorts of requests, for larger models, it is recommended to use a GPU-based variant that can speed up the inference times significantly.

Interface

`POST /webcat_interactive`

Description: This endpoint provides access to the interactive worker service that can be used to analyse the input of raw text provided by the user.

Parameters: The request body should contain a JSON object with the following keys:

- `hypothesis_template`: A string with a hypothesis template.
- `labels`: An array of strings with categorisation labels.
- `input`: Raw text data that should be analysed.
- `models`: A JSON object that contains information on the selected models.

Response: A JSON object containing information about categories and named entities found within the text input.

`GET /webcat_files_parser`

Description: Lists all available models integrated by the system.

Response: A JSON object containing information about the available models for both classification and NER tasks, containing name, description, size, model repository path, task, and default flag for all available models.

`POST /webcat_files_parser`

Description: Sends a request to parse file(s) to the worker service.

Parameters: The request body should contain a JSON object with the following keys:

- `hypothesis_template`: A string with a hypothesis template.
- `labels`: An array of strings with categorisation labels.
- `path`: Path to the file(s) that should be analysed.
- `recursive`: Explore files inside sub-directories as well.
- `save`: Save the results to permanent storage (database).
- `models`: A JSON object that contains information on the selected models.

Response: A JSON array containing all extracted content fragments with the assigned categories and named entities.

Templates Service

This service provides access to features related to templates, both their creation and their management. A detailed description of the templating process and the use case of templates will be further discussed in Section 4.5. This microservice can be split logically into main three resources it provides:

- **Templates Management**: Provides a basic CRUD interface for templates.
- **Element Types Management**: Provides a basic CRUD interface for the element types, which is the amotic part of each template. This enables the possibility to define custom content attributes that can be used in the templates to extract specific parts of content based on the needs of the user.
- **Template Engine**: Provides access to the templating engine, which implements an automatic way of creating templates.

Interface

`GET /templates`

Description: Lists all available templates stored in the database.
Response: A JSON array containing all templates used by the system.

`POST /templates`

Description: Create a new template.
Parameters: The request body should contain a JSON object with the following keys:

- `template`: An object of the Template class created by the user or template engine.

Response: Created object with the assigned identifier.

`DELETE /templates`

Description: Remove the template from the database.
Parameters: The request body should contain a JSON object with the following keys:

- `id`: An identifier of the template.

Response: Status code of the operation.

`GET /element_types`

Description: Lists all available element types stored in the database.
Response: A JSON array containing all the element types recognised by the system.

`POST /element_types`

Description: Create a new element type.
Parameters: The request body should contain a JSON object with the following keys:

- `name`: User-friendly name for the created type.
- `tag`: Unique tag that will be used by the system internally.
- `analysis_flag`: Determines whether or not the content fragment associated with this type should be analysed by the models.

Response: Created object with the assigned identifier.

`DELETE /element_types`

Description: Remove the element type from the database.
Parameters: The request body should contain a JSON object with the following keys:

- `id`: Identifier of the removed element type.

Response: Status code of the operation.

`GET /engine`

Description: Lists all available template engines provided by the system.
Response: A JSON array of available template engine objects with the following keys:

- `name`: Name of the template engine.
- `description`: Description of the template engine.
- `requires_key`: A flag indicates whether the template engine requires an access key or not (for example, to the remote API, such as the OpenAI API).

`POST /engine`

Description: Creates a template proposal from the provided file.
Parameters: The request body should contain a JSON object with the following keys:

- `engine`: Name of the engine that should process the file.
- `file_path`: Path to file that is used for the template creation.
- `key`: If the engine requires keys, they must be provided.

Response: Template proposal object created by the template engine.

Scheduler Service

The responsibility of the scheduler service is to manage access to the computational heavy resources and keep records of the system usage. The managed resource is the worker instances, which use language models that are computationally expansive. The scheduler keeps track of all workers connected to the system and maintains the reservation table stored within the database.

For now, the scheduling system is fairly simple and works on the first-in-first-out (FIFO) principle. When the reservation request is received, the first available worker will be reserved for the task. Note that it is the responsibility of the calling service to release the reserved worker by calling an appropriate endpoint on the scheduler service when the task is completed.

The capabilities and scheduling strategies can be easily expanded in the future, depending on the additional requirements on the system. One possible improvement includes smart reservations based on the requirements and complexity of the pipeline, which would distinguish between the need for CPU-based or GPU-based workers.

Interface

`GET /webcat_scheduler`

Description: Lists all available workers connected to the system.
Response: A JSON array containing all workers and their statuses.

`POST /webcat_scheduler`

Description: Create a new reservation for the worker.
Parameters: The request body should contain a JSON object with the following keys:

- `type`: Either „cpu“ or „gpu“, determines the type of worker.
- `file_path`: Path to the file(s) that should be analysed.

Response: Status of the reservation request, together with the information about reserved worker.

`DELETE /webcat_scheduler`

Description: Release the worker from the reservation table.
Parameters: The request body should contain a JSON object with the following keys:

- `id`: Identifier of the worker being released.

Response: Status code of the operation.

Web Client Service

To provide a comfortable way to interact with the system, we have implemented a graphical user interface (GUI) that provides natural access to all features provided by the gateway API. As already mentioned, the client is implemented in the `React` framework which allows for building responsive and dynamic user interface components. The `React` is a client-side framework, which means that the entire application runs in the client's web browser. The app itself must be served by a web server, as for now we use the default development server, but migration to a more production-suited web server implementation, such as `NGINX`[14], will be necessary in the near future. The web client is logically split into the following sections:

- **Iteractive Parser**: Provides access to the processing pipeline in an interactive way that allows testing of different hypothesis templates, category labels, and different models. The user can input any raw text data into the input window and inspect the results produced by categorisation and NER models.
- **Files Parser**: This section provides the same functionality as the Interactive Parser, but requires one to provide path to files instead of the raw input. When the processing task is completed, the results are displayed together with summary statistics.
- **Template Maker**: This feature enables the creation of templates. The user can choose between automatic and manual modes for template creation. The automatic approach uses the template engine to automatically create templates from files, while the manual mode allows users to load HTML files directly onto the website and provides tools for content annotation.
- **Template Manager**: It is a template management tool that provides an overview of the templates that are actively used by the parser.
- **Data Viewer**: A useful tool for inspecting the results processed. The user can choose from a variety of filtering options and search within the results.

Screenshots of all sections are provided in the appendix A.

4.4 Data Processing Workflow

The data processing workflow of our system begins when the user interacts with the web client application, which communicates with the API. The user provides the system with the path to the file(s) to be processed and the configuration of the pipeline that will analyse the content. If the path provided is a directory, the system automatically discovers all files within that directory. The API then requests an url of a free worker from the scheduler service, which keeps a table of worker statuses in the database. If a free worker is available, the scheduler reserves the worker by changing its status from „free“ to „busy“ and returns the URL of the worker with a success status code.

Once the worker reservation has been made successfully, the API sends a request to the reserved worker, which loads a pipeline based on the configuration provided by the user. The configuration contains information on the deep learning models that should be used for processing, together with additional information, such as category labels, the hypothesis template, and whether the final results should be stored in the database.

[14] https://www.nginx.com/

The pipeline includes a parser that requires templates for content extraction to be fetched from the database created by the templating engine, which will be discussed further in the section on templating. The parser then parses the files in parallel using the `Joblib` library and returns the fragments of the content found. The analyser component of the pipeline first uses a deep learning model trained on the textual entailment task to categorise each attribute of the content that is marked for analysis by the flag. Then it uses a named entity recognition (NER) model to find named entities, and finally, the pipeline stores the content in the database using ORM and returns a serialised response with the analysed content in JSON format.

Once the job is completed, the API requests the release of the worker from the scheduler service and returns the results to the client. The entire processing pipeline provided by the worker can be seen in Figure 4.2.

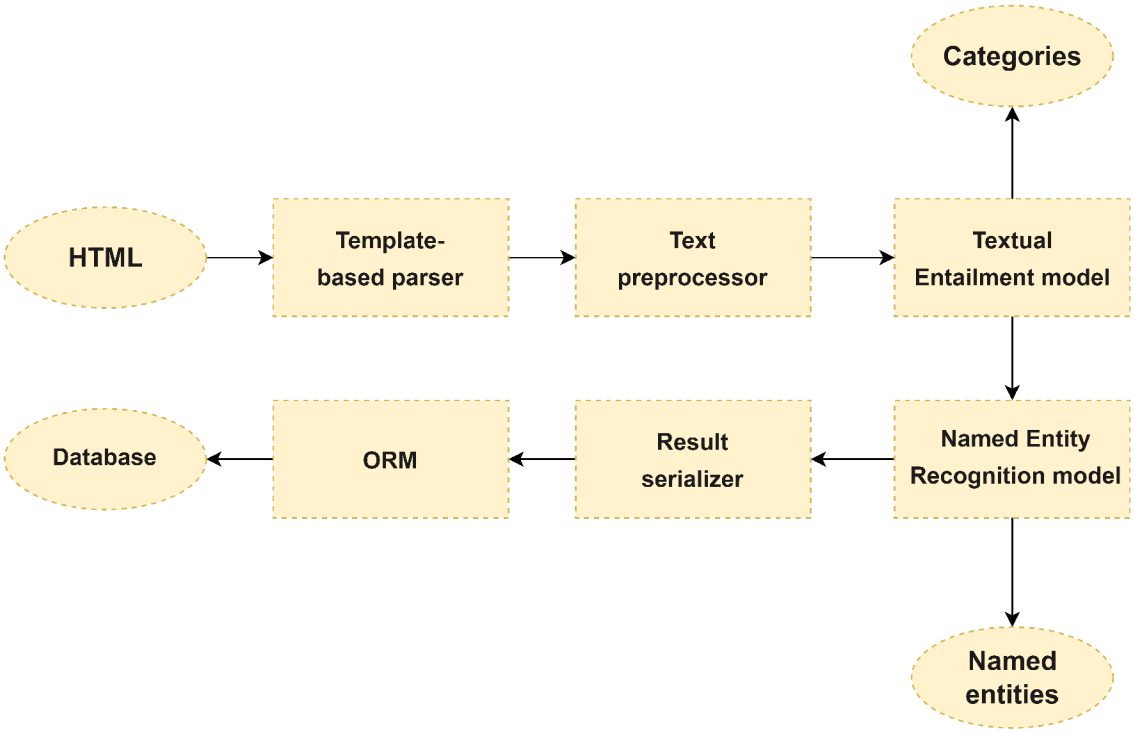


Figure 4.2: Processing pipeline executed by the worker service.

4.5 Templating

Templating is a key method of our system's data extraction process. To extract structured data from unstructured sources such as web pages, we match them against predefined templates. A template is a set of elements that must be present in the source file to be used for extraction. These templates are stored in our database, and the parser will attempt to match each template to the files that are being processed. The template entities stored in the database can be described by the ER diagram in Figure 4.3.

The `Template` entity keeps the date of creation and the path to the origin file. Each template contains at least one element, but for the system to work efficiently and accurately,

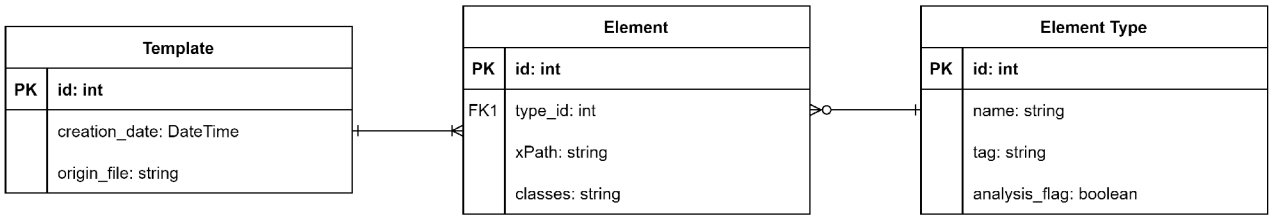


Figure 4.3: ER diagram of the Template and associated entities.

it is highly anticipated to create templates with as many `Element` entities as possible. The `Element Type` is an entity that tags an element with additional information about its content. For example, `Element Type` can denote messages, titles, usernames, profile links, etc. It also contains an `analysis_flag` attribute, which indicates whether the element's content should undergo further analysis or not.

In our system, templates can be created in multiple ways. The user can either create templates manually with our special tool implemented in the client application or the system can try to create a template in a fully automatic way.

Manual Template Creation

Creating templates manually is the easiest way to define templates because it allows the user to annotate specific parts of the web pages to extract any pieces of information that the user is interested in. We have implemented a tool in our web client which allows parsing any HTML file and rendering the structure of a web page into an interactive window. Each element of the original web page is expandable, and textual nodes are highlighted for better clarity. Each element can be annotated by selecting the desired annotation type from the selector above the interactive window and then clicking on the element which should get annotated. The screenshot of our web client can be found in the Appendix A.4.

Automatic Template Creation

We tried to implement a convenient way of creating templates from HTML files without the need for manual annotation. We focus only on the domain of web forums, where the content is logically split into posts or messages. The main structure of the post is always very similar to all other posts present on the web page. The key idea was to identify these repeating structures and find patterns, which would indicate the parent tag wrapping the repeating subtree with a high degree of similarity.

The experiments were carried out using clustering techniques, for example, the K-Mean clustering method. We calculate these clusters by constructing the vocabularies of the element tags and their classes V_t and V_c. Then we define a feature vector

$$F = [t, C, D_t, D_c, d, n, l],$$

where t is the index of the element tag in the vocabulary V_t, C is a vector of indices of elements classes of the vocabulary V_c, D_t is a set of all descending tag indices from the vocabulary V_t, D_c is a set of all descending class indices from the vocabulary V_c, d is the depth of the element within the DOM tree, n is the number of child elements, and l is the length of the text content present within the element, normalised by the text length from the entire web page.

```
I want you to act as a data extraction tool and extract post titles,
post authors, and post messages from raw text extracted from HTML
code of a forum website. The raw text is separated by newline characters
and the desired output is a JSON array of objects in the format
[{"post-title": "some title", "post-author": "some author",
"post-message": "some message"}]. Your task is to segmentate the
raw text and extract the required information from each segment. Do not
modify the input in any way, the output values must be identical with the
input lines.
The input is: [TEXT]
```

Figure 4.4: The prompt used by the ChatGPT-3.5 for performing the segmentation task on the extracted text data. The `[TEXT]` token appearing at the end of the prompt is being replaced by the extracted text.

While clustering methods are able to form clusters with one of them representing the set of posts, without further validation and guidance from the user it is difficult to identify the correct cluster and further segmentate the content without complex and highly biased heuristics, which made this method unusable for practical cases.

4.6 ChatGPT-driven Template Creation

Given the recent advances in generative language models and the release of highly successful OpenAI[15] GPT-4 and ChatGPT-3.5 models, we carried out experiments on the effectiveness of these language models in the analysis of textual data from unstructured data sources, such as web pages.

The main focus was on content segmentation, as we strived to identify content fragments and map them to specific HTML elements. After experimenting with the different prompts and input data formats, we have come up with a satisfactorily working solution. Suprisingly, passing the raw HTML code to the language model did not work, although we assumed that it should be the most sufficient form, as it excels at overall code generation.

The working solution is based on passing the raw textual data extracted directly from the parsed HTML, represented by DOM. We intentionally preserved only the newline characters that denote the boundaries of the elements, while removing all newline characters from the text content itself. The prompt we have used can be seen in Figure 4.4.

To further reduce the number of tokens and reduce the overall cost and inference time, the context window has been introduced as part of the input preprocessing for the ChatGPT model. To segmentate text and identify the repeating content fragments, such as posts or messages, and their logical structure, it is not necessary to pass the entire content of the webpage. In our method, we set a fixed maximum number of lines, as well as the maximum limit for characters that each line can include. We will refer to these parameters as `MAX_LINES` and `MAX_CHARACTERS_PER_LINE` respectively. In general, the starting lines of text extracted in this way will include many unnecessary data, such as the names of the navigation elements. To address this redundancy, we have decided to skip these lines to further save more tokens and will refer to this parameter as `INPUT_OFFSET`. No particular

[15] https://openai.com/

strategy for setting the offset has been used, as the resulting algorithm does not take the location of the content into account. In general, we aim to identify the repeating content fragments that can occur multiple times in the webpage, and thus the relative positioning of these fragments does not matter. The resulting algorithm is described in Figure 4.5.

To provide more clarity about the input format that the model receives, we have prepared the visualisation that can be seen in Figure 4.6. The entire red area is the raw textual content extracted from a forum web page. The blank lines serve as a delimiter between content that was part of a separate textual node. When formatting the text in this way, we can instruct the language model to pay attention to these gaps, as they potentially represent an important delimiter between some logical sections of content.

Parameters `MAX_LINES` and `MAX_CHARACTERS_PER_LINE` are subject to further testing and optimisation to minimise the total token spend, leading to faster response times and a lower cost of the overall process. In general, the goal is to provide enough clues for the model so that it can annotate the content as accurately as possible.

4.7 Categorisation Model

One of the most important parts of our system are the deep learning models used for content categorisation. We have focused on the natural language inference approach described in Section 2.6. This method is suitable for the domain of our focus, the darknet environment, as we lack a sufficient amount of annotated data. Since our work was mainly focused on the design and implementation of the automatic system for content categorisation and extraction, we have decided to outsource the models for now and constructed a flexible framework that makes it easy to add any additional models in the future, allowing the possibility for future research and development of custom models.

As for the models, we work mainly with the already pre-trained model of a Ph.D. candidate researcher Moritz Laurer (VU Amsterdam) [20], which is based on the Microsoft `DeBERTA V3` architecture [13]. We have integrated two variants of the pre-trained models, `DeBERTA-v3-large`[16] and `DeBERTA-v3-base`[17].

Another model of different architecture types that we integrated was `bart-large-mnli`[18] based on the Facebook `BART` [21] architecture. However, given the poorer performance of this model on the MNLI data set and significantly longer inference times, we have chosen `DeBERTA-v3-large` as our base model for the categorisation task. For completeness, the accuracy on the MNLI data set according to [19] and [21] is the following:

Model	**Accuracy MNLI_m** (%)	**Accuracy MNLI_mm** (%)
DeBERTa-v3-large	**91.2**	**90.8**
DeBERTa-v3-base	90.3	90.3
BART-large	89.9	90.1

Table 4.1: Comparison of Language Model Accuracy on the MNLI Dataset

[16] https://huggingface.co/MoritzLaurer/DeBERTa-v3-large-mnli-fever-anli-ling-wanli
[17] https://huggingface.co/MoritzLaurer/DeBERTa-v3-base-mnli
[18] https://huggingface.co/facebook/bart-large-mnli

1. Extract all text from the webpage and remove all newline characters from the content of elements. Save the mapping between the text and their corresponding HTML elements.
2. Skip the first `INPUT_OFFSET` lines of extracted text.
3. Cut the number of lines to a maximum length of `MAX_LINES`.
4. Cut each line of text to a maximum length of `MAX_CHARACTERS_PER_LINE`.
5. Send the text data in the prompt from Figure 4.4 to `OpenAI API` and parse it using the `ChatGPT-3.5-turbo` model.
6. Extract the **identified segments** from the model output.
7. If multiple entities were found, group values by **segments**. The **segment** refers to the type of the value, for example, title, author or message.
8. For each **identified segment**:
   - 8.1.1 Match the textual values annotated by the model with the corresponding HTML element from the stored mapping.
   - 8.2.2 Count the number of elements with equal XPath that were returned for current segment.
   - 8.3.3 Select the element with the highest number of occurences. If multiple elements have the same count, keep all of them. These elements are referred to as **candidate elements**.
9. For each **identified segment**:
   - 9.1.1 Use the stored XPath of each **candidate element** to retrieve all matching elements from the entire web page.
   - 9.2.2 The **candidate element** that was able to locate the highest number of nodes is selected as the **template element**.
10. Construct the template proposal from all **identified segments** and their corresponding **template elements**.

Figure 4.5: The algorithm for automatic content segmentation from HTML file, with the usage of `ChatGPT-3.5-turbo` model.

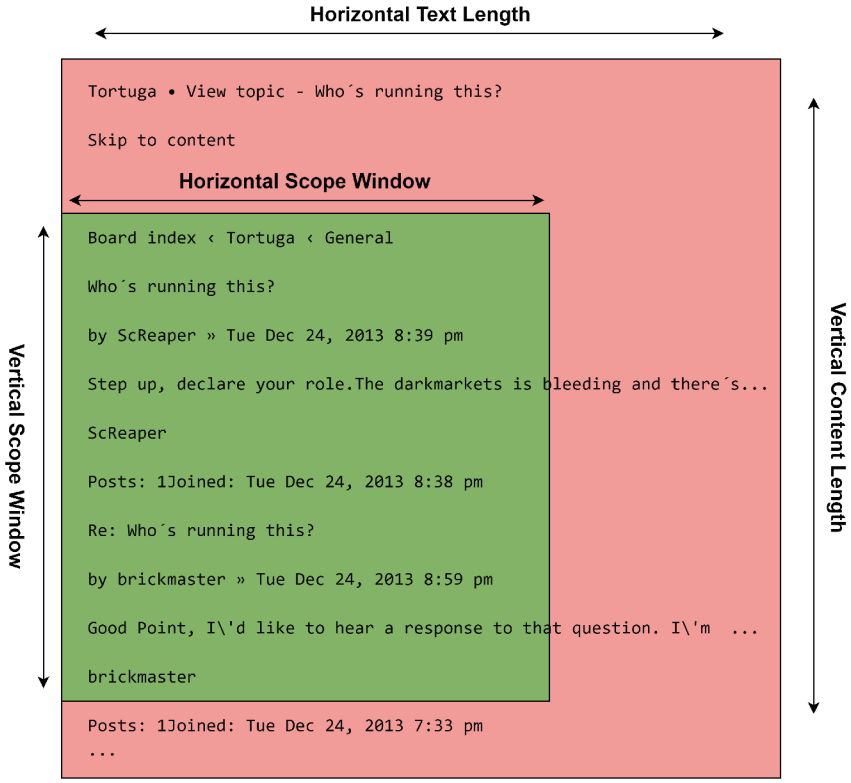


Figure 4.6: Visualization of the context window seen by the ChatGPT 3.5 model.

4.8 Named Entity Recognition Model

We have decided to add the NER model to our processing pipeline to enhance the data extraction process, producing additional data that can be useful for further analysis. Familiarisation with the NER problem took place in Section 2.7.

For the NER model, we chose a similar path as with the categorisation models and outsourced the pre-trained model. However, we had to consider a different approach to the selection of the suitable model, as the NER task is not as flexible as NLI and the set of entities that the model is capable of recognising had to be chosen carefully, as well as the data domain it has been trained on.

We have found a promising and performant NER model[19] provided by a tool `T-NER` [41]. This model is based on the `RoBERTa` architecture and is trained on the specialised `TweetNER7` data set [42]. The `TweetNER7` is made up of annotated posts (tweets) from the Twitter social network. The annotations contain the following labels: *corporation*, *creative_work*, *event*, *group*, *location*, *product* and *person*.

This data set is ideal for our use case of finding named entities within posts on the darknet forums, as the domain is still different but quite similar in certain ways. One challenge we encountered while testing other models was the inability to detect usernames, which are quite different from the names of real people. Models fine-tuned on the domain where usernames are commonly used naturally perform significantly better in classifying usernames than other models trained on data sets containing annotations of the real names only. Another advantage of the Twitter data set created by the authors is that it provides

[19] https://huggingface.co/tner/twitter-roberta-base-dec2021-tweetner7-random

annotations for entities labelled *product*, which is not common for other conventional NER data sets. The ability to detect certain products in conversations can be a great source of information for further content filtering, especially in the market environment, where users discuss or review products a lot.

Chapter 5

Results

In this chapter we summarise the results and outline possible directions for further work. The outcome of this work is a system called WebCat, which is a containerised application that can be deployed nearly anywhere and can be used for categorisation and extraction of the content located on the web pages. The source code and installation instructions are included on the attached memory medium. More information about its content and the link to NextCloud storage can be found in the Appendix B.

5.1 Comparison of Different Hypothesis Templates

The key component of the NLI classifiers for categorisation are the hypothesis templates. The selection of the hypothesis can affect the results in a significant way, and we have conducted experiments to search for the most suitable template that can be used for the text categorisation task. We have chosen the `DeBERTa_v3-large` model as the base model for our experiments, as the model performs the best of our inspected models in the NLI task in general, as shown in Table 4.1.

In the absence of suitable data sets for the darknet market domain, the experiments were carried out on a total of 2000 samples from the data set **newsgroup**[1] that was introduced in [18]. The newsgroup data set is a well-known benchmark data set that is used for text classification tasks. It contains messages from various news groups posted in the early 1990s. The data set includes more than 20.000 documents, classified into 20 different topics, such as *politics*, *sports*, and *religion*.

We have modified this data set by selecting and aggregating categories into its supercategories, whenever possible, to eliminate the overlap of categories. For example, categories such as *hardware*, *graphics*, or *ms-windows* were merged into the `computer` super category, as the overlap of the common domains causes noise in our evaluation process, since the model can output multiple categories, but the data set contains only one ground truth label for each sample. When the sample is classified into concrete category, but the model also identifies some of the subcategories, the precision is negatively distorted. For this reason, we have decided to split the data set only into the following diverse labels: *medicine*, *computer*, *religion*, *sport*, *motorcycles*, *guns*, *forsale*, *cars*, and *politics*.

The evaluation of text classification algorithms often involves measuring the F1, Precision, and Recall scores. These scores are used to assess the performance of a model in terms of its ability to correctly classify instances into their respective classes. F1 score is the

[1] https://huggingface.co/datasets/newsgroup

harmonic mean of precision and recall, which makes it a balanced metric for models that are good at either precision or recall but not both. Precision is the ratio of true positives to the total number of predicted positives, while recall is the ratio of true positives to the total number of actual positives. These scores can be calculated using the following formulas:

$$Precision = TP/(TP + FP)$$

$$Recall = TP/(TP + FN)$$

$$F1 = 2 * ((Precision * Recall)/(Precision + Recall))$$

where TP represents true positives, FP represents false positives and FN represents false negatives.

#	**Hypothesis Template**	**F1**	**Precision**	**Recall**
1	The text examines the topic of **x** in depth.	0.782	0.720	0.855
2	The topic of this text is **x**.	**0.841**	**0.801**	0.884
3	The text provides information about **x**.	0.804	0.707	0.933
4	The text discusses **x**.	0.838	0.771	0.917
5	The text contains information relevant to **x**.	0.650	0.489	**0.969**
6	The text provides insights into **x**.	0.694	0.553	0.931
7	The text sheds light on **x**.	0.755	0.638	0.924
8	The text explores **x** in detail.	0.796	0.722	0.888
9	This text covers the topic of **x**.	0.827	0.742	0.933
10	The central theme of this article is **x**.	0.789	0.736	0.850
11	The text provides a comprehensive analysis of **x**.	0.375	0.356	0.397
12	The text covers the topic of **x** extensively.	0.798	0.713	0.905
13	The primary subject matter of this article is **x**.	0.796	0.737	0.867
14	The text delves deeply into the topic of **x**.	0.813	0.726	0.925
15	The main idea presented in this text is **x**.	0.796	0.737	0.870
16	The example is about **x**.	0.823	0.773	0.880
17	The focus of this writing is on **x**.	0.811	0.753	0.879

Table 5.1: Performance of different hypothesis templates on a newsgroup data set for the `DeBERTa_v3-large` model variant.

As can be seen in Table 5.1, the selection of the hypothesis template can have a strong influence on the behaviour of the classifier. In general, the metric considered should mainly be the F1 score, as it provides the optimal balance between precision and recall metrics. However, the preferred metric can also depend on the task and the environment itself. If the main focus is on minimising the FP predictions (i.e., cases where the model does not output the positive label at all), the preferred metric could be the recall. However, as can be seen in Table 5.1, the hypothesis template with the highest recall performs much worse than others in terms of precision. This observation indicates that the model predicts more than two labels as positive on average, most likely producing unnecessary noise.

We recommend either the model with the highest F1 score or the recall depending on the criteria described above. However, we are also interested in the `#3` hypothesis with the second-highest recall, as the general precision is much better compared to the `#5` hypothesis

for the only slight loss in the recall metric. For this reason, the `#3` hypothesis has been set as a default for the `DeBERTa_v3-large` model, but users are encouraged to try different well-performing hypotheses to see which performs best in their domain of interest.

For comparison, we have also evaluated the performance on the identical set of hypothesis templates of the `DeBERTa_v3-base` model and the results are shown in Table 5.2. The results clearly demonstrate the general robustness and better performance of the `large` variant, as none of the proposed templates used by the `base` model was able to outperform or compete with the results in Table 5.1. Furthermore, the `#1` and `#11` examples show the proneness of the `base` model to the inability to perform the categorisation task, when certain unsuitable hypotheses are selected, demonstrating the generally better robustness of the `large` model.

#	Hypothesis Template	F1	Precision	Recall
1	The text examines the topic of **x** in depth.	0.225	0.213	0.239
2	The topic of this text is **x**.	0.744	0.645	0.880
3	The text provides information about **x**.	0.778	0.697	0.880
4	The text discusses **x**.	0.766	0.666	0.902
5	The text contains information relevant to **x**.	0.667	0.521	**0.926**
6	The text provides insights into **x**.	0.510	0.480	0.543
7	The text sheds light on **x**.	0.558	0.481	0.666
8	The text explores **x** in detail.	0.628	0.585	0.679
9	This text covers the topic of **x**.	0.764	0.667	0.894
10	The central theme of this article is **x**.	0.723	0.655	0.783
11	The text provides a comprehensive analysis of **x**.	0.006	0.006	0.006
12	The text covers the topic of **x** extensively.	0.542	0.501	0.591
13	The primary subject matter of this article is **x**.	0.727	0.678	0.783
14	The text delves deeply into the topic of **x**.	0.536	0.511	0.564
15	The main idea presented in this text is **x**.	0.743	0.680	0.818
16	The example is about **x**.	**0.781**	**0.714**	0.862
17	The focus of this writing is on **x**.	0.668	0.594	0.762

Table 5.2: Performance of different hypothesis templates on a newsgroup data set for the `DeBERTa_v3-base` model variant.

5.2 Strengths and Limitations

As for the evaluation of system performance in our domain of interest, which is an environment of darknet forums, due to the lack of data sets that would be suitable for evaluation purposes, we will identify and summarise the strengths, weaknesses, and behaviour in the real environment based on our observations and experience gained while using the system.

We have tested the system in the environment of the `dnmarchives` [6] data set, particularly in the archives of darknet market forums such as Utopia, Bungee54, Abraxas, Darkbay, or Greyroad. The concrete categories that we have used for testing purposes were selected to represent the diverse controversial topics that are generally considered to appear on the darknet. We have configured the processing pipeline to categorise forum posts into following categories: *drugs*, *hacking*, *fraud*, *cryptocurrency*, *shipment delivery*, *weapons*, *counterfeit*, *childporn* and *cryptography*.

Some of the following examples contain highly sensitive and controversial topics, however, we believe that it is important to demonstrate how our system operates in the real environment and its potential to detect malicious content in general. The example demonstrating the operation of the classifier can be seen in Figure 5.1. We identified the mapping between the text content and the categories and marked them with matching colours to provide a better insight. We believe that the example shows both the strengths and limitations of the categorisation process. In general, the classifier is highly prone to inferring categories based on the occurrence of the words from the desired domain rather than considering the whole context. This can be clearly visible in the case of the detection of the *drugs* category, where the word *Dope*, present within the username, is associated with this category. Another frequently occurring anomaly is the association of the *cryptocurrency* label with the numbers, dates, and hyperlinks in general. We have made an effort to preprocess the text and remove all dates and hyperlinks from the text input. However, many cases, such as the price tags, which are present within the example, are difficult to treat, and the question of their processing is the subject of further research and development.

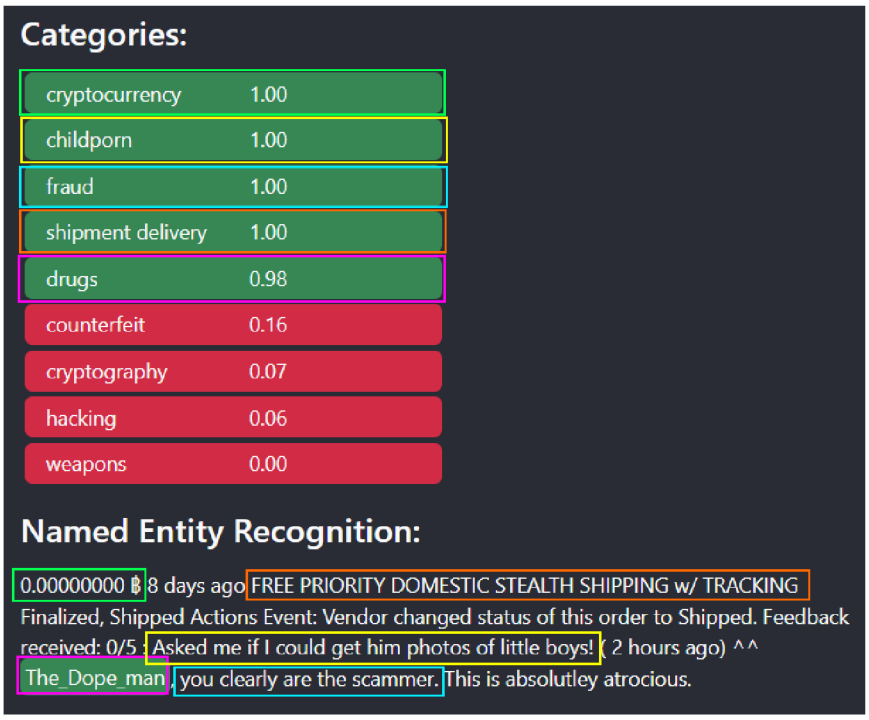


Figure 5.1: An example of a content (post in the forum thread) retrieved by the WebCat system. The example was put into an interactive parser tool provided by the web client, so we could analyse and identify the concrete mapping of categories and the relevant text areas. Boxes matching the same colour indicate the corresponding mapping. When any of the marked text is removed, the corresponding category is no longer being detected.

Another example presented in Figure 5.2 demonstrates the utility of the NER model to extract useful information about the named entities that occurred in the content. We have also identified the text areas that affect the classifier for *cryptocurrency* and *counterfeit* categories, to address the problem mentioned above with the hyperlinks in regards to *cryptocurrency* label, and to demonstrate the ability to classify challenging and niche categories, such as *counterfeit*.

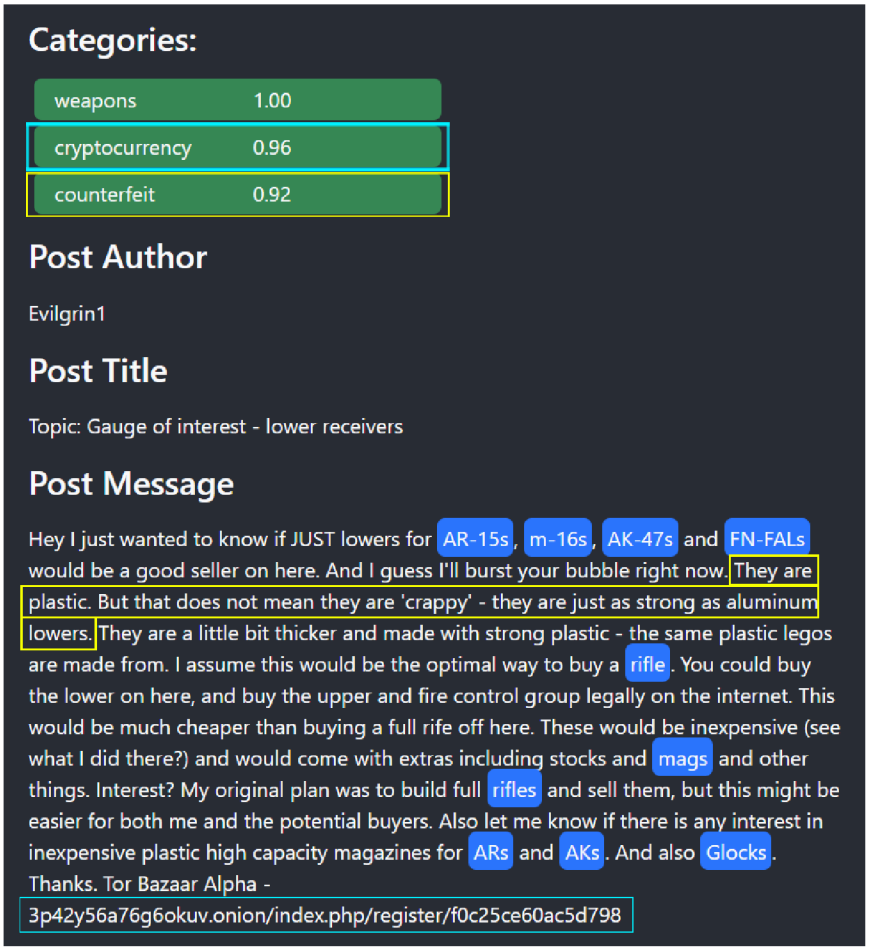


Figure 5.2: An example of a content (post in the forum thread) retrieved by the WebCat system. Words highlighted with the blue colour in the text were tagged by the NER model and classified as tokens with *product* label.

The last example shown in Figure 5.3 demonstrates the susceptibility of the classifier to draw a wrong conclusion from the ambiguous sequence of words. This can be applied to many different cases. For example, the reason behind the selection of the label *shipment delivery* rather than *delivery* alone, was the fact of frequent association of the reference to email communication with the *delivery* category, which was unintended and produced unnecessary noise. This demonstrates the importance of selecting unambiguous labels and testing these labels against representative samples to validate the intentions of the specific category.

Overall, the examples presented suggest that more work needs to be done on the formulation of labels and text pre-processing. Another space for improvement may be the exploration and further evaluation of different hypothesis templates, as was discussed in Section 5.1. In this regard, we will strive to create a curated data set for evaluation purposes with the use of tools that we have developed, to support further research in this area, particularly for the domain of the darknet environment.

Figure 5.3: An example of a content (post in the forum thread) retrieved by the WebCat system. This example represents the situation, where the classifier is confused by the ambiguous meaning of the sequence.

5.3 Future Expansions and Improvements

The topic of this thesis is extensive and could be developed further in many different directions. We have focused mainly on the design and implementation of the system for automatic webpage content categorisation and extraction, however, many techniques and features used in this work could be further expanded and improved. In the following sections, we will explore and propose subjects for further research and development.

Templating

There are multiple ways that templating in general could be improved to make this technique faster and more robust. For now, the templates are highly generic and are not bound to specific websites. If the complexity of the template object is too small, there is the possibility that the template will match a different website with a similar structure, resulting in incorrect content mapping. Adding meta-data about the page into the templates might solve this issue and further speed up the process of the correct template selection for the parsed webpage.

In connection with the previous paragraph, resolving the template affiliation to the specific web pages would also help to allow the system to be fully autonomous when parsing large-scale web archives. At the moment, due to the possibility of conflicting templates, the system is not reliable in processing many different websites at once. To implement this feature, the templates have to be more robust and other mechanisms, such as webpage candidate selection for the automatic template creation, must be further developed. Due to these missing parts, the system is not yet able to use the ChatGPT-powered templating engine to produce new templates for unknown web pages in a fully autonomous way.

Support for Different Input Formats

The database structure was designed to store data of various content from unstructured sources, primarily HTML webpages. This allows the possibility of expanding the pool of supported file types accepted by the system, opening the way for different use cases. One of the possible use cases might be the analysis of content from structured formats such as CSV or JSON. The system could also contain a feature for exporting results into different file types, or serve as an enrichment tool that would append analytical results to the files directly.

Smart Scheduling

As was outlined in Section 4.3 about the scheduler service, the scheduling algorithm is fairly simple and does not make full use of the system's capabilities. One of the possible improvements might include evaluating the complexity of the required processing pipeline configuration and assigning the task to CPU or GPU-based workers based on the estimated complexity.

Custom Deep Learning Models

Another important and specific modification for the domain of the darknet environment could involve fine-tuning categorisation and NER models. Furthermore, in theory, any demanded domain could be adapted by training our own models on data sets from different domains, adapting the analysis to the specific needs of the user. The web client application could also be modified to reflect the availability of different features tailored for specific domains and use cases.

In terms of the domain of interest, the key challenge is to find suitable data sets that could potentially improve the overall performance of the models. We have explored different ways for potential synthesis or auto-labelling of the darknet domain data, such as exploitation of the general language models (GLMs) such as `OpenAI GPT-4, GPT-3` or some of its open-source but much less performant variants, namely `gpt4all` [3] and `ChatGLM` [49]. We believe that the transfer of knowledge from these language models by creating synthesised data sets could potentially improve the quality of the analysis in many different domains. However, this topic requires its own research and therefore is beyond the scope of this thesis.

5.4 Summary

The subject of automatic webpage content categorisation and extraction is a vast and multifaceted area of research. Our work is focused on the highly specialised domain of the darknet environment, which presents unique challenges due to the scarcity of suitable data sets. To tackle this issue, we have approached the problem with a highly adaptable and universal methodology, intended to create a flexible system capable of handling any content type and scalable to meet future development requirements.

The system we have designed is intended for general use and we have made critical design decisions to create a system that is both agile and extensible. This includes careful consideration of the overall architecture, database structure, and configurable pipeline of the system.

As demonstrated in Section 5.2, the model trained on the natural language inference data set has the potential to categorise content effectively, provided that the appropriate hypothesis template is used. We have conducted an in-depth analysis of the performance of various hypothesis templates, as detailed in Section 5.1.

To facilitate the data extraction process, we have integrated a named entity recognition (NER) model into our processing pipeline. This model is capable of identifying named entities that are stored in the database for further analysis. In addition, we have employed a templating methodology to automate content extraction. This involves creating a template and using it to process similar web pages in a consistent manner. Our templating methods are described in Section 4.5.

Furthermore, we have proposed an automated approach to segmenting website content using the OpenAI's `ChatGPT-3.5` model. This technique provides annotations that can be used to generate templates and automatise the extraction process. The segmentation process is described in Section 4.6.

In order to enhance the accessibility and ease-of-use of our system, we have developed a web client application that offers a user-friendly graphical interface for interacting with the system. Detailed screenshots of this application can be found in the appendices A.

Bibliography

[1] Aggarwal, C. *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing, 2018. ISBN 9783319944630. Available at: https://books.google.cz/books?id=achqDwAAQBAJ.

[2] Alyafeai, Z., AlShaibani, M. S. and Ahmad, I. *A Survey on Transfer Learning in Natural Language Processing*. arXiv, 2020. DOI: 10.48550/ARXIV.2007.04239. Available at: https://arxiv.org/abs/2007.04239.

[3] Anand, Y., Nussbaum, Z., Duderstadt, B., Schmidt, B. and Mulyar, A. *GPT4All: Training an Assistant-style Chatbot with Large Scale Data Distillation from GPT-3.5-Turbo* [https://github.com/nomic-ai/gpt4all]. GitHub, 2023.

[4] Bhatt, S., Goyal, P., Dandapat, S., Choudhury, M. and Sitaram, S. *On the Universality of Deep Contextual Language Models*. arXiv, 2021. DOI: 10.48550/ARXIV.2109.07140. Available at: https://arxiv.org/abs/2109.07140.

[5] Bowman, S. R., Angeli, G., Potts, C. and Manning, C. D. A large annotated corpus for learning natural language inference. In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2015.

[6] Branwen, G., Christin, N., Décary Hétu, D., Andersen, R. M., StExo et al. *Dark Net Market archives, 2011-2015* [https://www.gwern.net/DNM-archives]. July 2015. Accessed: DATE. Available at: https://www.gwern.net/DNM-archives.

[7] Clark, K., Luong, M.-T., Le, Q. V. and Manning, C. D. *ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators*. 2020.

[8] Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K. and Harshman, R. *Indexing by latent semantic analysis*. Wiley Online Library, 1990.

[9] Devlin, J., Chang, M.-W., Lee, K. and Toutanova, K. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019.

[10] Ekbal, A. and Bandyopadhyay, S. Named entity recognition using support vector machine: A Language independent approach. january 2010, vol. 39.

[11] Goodfellow, I., Bengio, Y. and Courville, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[12] Haykin, S. *Neural Networks and Learning Machines*. Pearson, 2009. Pearson International Edition. ISBN 9780131293762. Available at: https://books.google.cz/books?id=KCwWOAAACAAJ.

[13] He, P., Gao, J. and Chen, W. *DeBERTaV3: Improving DeBERTa using ELECTRA-Style Pre-Training with Gradient-Disentangled Embedding Sharing*. 2021.

[14] Huang, Z., Xu, W. and Yu, K. *Bidirectional LSTM-CRF Models for Sequence Tagging*. 2015.

[15] IBM. *What is natural language processing (NLP)?* [online]. [visited 2022-09-11]. Available at: https://www.ibm.com/topics/natural-language-processing.

[16] Kowsari, K., Jafari Meimandi, K., Heidarysafa, M., Mendu, S., Barnes, L. et al. Text Classification Algorithms: A Survey. *Information (Switzerland)*. april 2019, vol. 10. DOI: 10.3390/info10040150.

[17] Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P. et al. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2020.

[18] Lang, K. NewsWeeder: Learning to Filter Netnews. In: Prieditis, A. and Russell, S., ed. *Machine Learning Proceedings 1995*. San Francisco (CA): Morgan Kaufmann, 1995, p. 331–339. DOI: https://doi.org/10.1016/B978-1-55860-377-6.50048-7. ISBN 978-1-55860-377-6. Available at: https://www.sciencedirect.com/science/article/pii/B9781558603776500487.

[19] Laurer, M. *Model Card for DeBERTa-v3-large-mnli-fever-anli-ling-wanli* [online]. [visited 2023-05-02]. Available at: https://huggingface.co/MoritzLaurer/DeBERTa-v3-large-mnli-fever-anli-ling-wanli.

[20] Laurer, M., Atteveldt, W. van, Salleras Casas, A. and Welbers, K. *Less Annotating, More Classifying – Addressing the Data Scarcity Issue of Supervised Machine Learning with Deep Transfer Learning and BERT - NLI*. June 2022. Available at: https://osf.io/74b8k.

[21] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A. et al. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *CoRR*. 2019, abs/1910.13461. Available at: http://arxiv.org/abs/1910.13461.

[22] Li, J., Luong, T. and Jurafsky, D. A Hierarchical Neural Autoencoder for Paragraphs and Documents. In: *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Beijing, China: Association for Computational Linguistics, July 2015, p. 1106–1115. DOI: 10.3115/v1/P15-1107. Available at: https://aclanthology.org/P15-1107.

[23] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M. et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019.

[24] Liu, Z., Lin, Y. and Sun, M. *Document Representation*. Singapore: Springer Singapore, 2020. 91–123 p. ISBN 978-981-15-5573-2. Available at: https://doi.org/10.1007/978-981-15-5573-2_5.

[25] MacDonald, M. *HTML5 : the missing manual*. 1st edth ed. O'Reilly Media, 2011. ISBN 9780131293762.

[26] Manning, C. D., Raghavan, P. and Schütze, H. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[27] Mikolov, T., Chen, K., Corrado, G. and Dean, J. *Efficient Estimation of Word Representations in Vector Space*. arXiv, 2013. DOI: 10.48550/ARXIV.1301.3781. Available at: https://arxiv.org/abs/1301.3781.

[28] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. and Dean, J. *Distributed Representations of Words and Phrases and their Compositionality*. arXiv, 2013. DOI: 10.48550/ARXIV.1310.4546. Available at: https://arxiv.org/abs/1310.4546.

[29] Mitra, B. and Craswell, N. An Introduction to Neural Information Retrieval. *Foundations and Trends® in Information Retrieval*. 2018, vol. 13, no. 1, p. 1–126. DOI: 10.1561/1500000061. ISSN 1554-0669. Available at: http://dx.doi.org/10.1561/1500000061.

[30] Nadkarni, P. M., Ohno Machado, L. and Chapman, W. W. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*. september 2011, vol. 18, no. 5, p. 544–551. DOI: 10.1136/amiajnl-2011-000464. ISSN 1067-5027. Available at: https://doi.org/10.1136/amiajnl-2011-000464.

[31] Network, M. D. *The HTML DOM (Document Object Model)* [online]. [visited 2022-11-12]. Available at: https://developer.mozilla.org/en-US/docs/Web/XPath.

[32] Nivre, J. On Statistical Methods in Natural Language Processing. *Promote IT Second Conference for the Promotion of Research in IT at New Universities and University Colleges in Sweden*. january 2002.

[33] Padmanabhan, A. *Text Corpus for NLP* [online]. [visited 2022-09-12]. Available at: https://devopedia.org/text-corpus-for-nlp.

[34] Pan, S. J. and Yang, Q. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*. 2010, vol. 22, no. 10, p. 1345–1359. DOI: 10.1109/TKDE.2009.191.

[35] Pennington, J., Socher, R. and Manning, C. D. Glove: Global vectors for word representation. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, p. 1532–1543.

[36] Richardson, C. *What are microservices?* [online]. [visited 2023-04-16]. Available at: https://microservices.io/.

[37] Robie, J. *What is the Document Object Model?* [online]. [visited 2022-01-16]. Available at: https://www.w3.org/TR/WD-DOM/introduction.html.

[38] Steven Bird, E. L. *Natural Language Processing with Python*. 1st ed. O'Reilly Media, Inc., 2009. ISBN 9780596516499.

[39] Thönes, J. Microservices. *IEEE Software*. 2015, vol. 32, no. 1, p. 116–116. DOI: 10.1109/MS.2015.11.

[40] Trabelsi, M., Chen, Z., Davison, B. D. and Heflin, J. Neural ranking models for document retrieval. *Information Retrieval Journal*. Springer Science and Business Media LLC. oct 2021, vol. 24, no. 6, p. 400–444. DOI: 10.1007/s10791-021-09398-0. Available at: https://doi.org/10.1007%2Fs10791-021-09398-0.

[41] Ushio, A. and Camacho Collados, J. T-NER: An All-Round Python Library for Transformer-based Named Entity Recognition. In: *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*. Online: Association for Computational Linguistics, Apr 2021, p. 53–62. DOI: 10.18653/v1/2021.eacl-demos.7. Available at: https://aclanthology.org/2021.eacl-demos.7.

[42] Ushio, A., Neves, L., Silva, V., Barbieri, F. and Camacho Collados, J. Named Entity Recognition in Twitter: A Dataset and Analysis on Short-Term Temporal Shifts. In: *The 2nd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 12th International Joint Conference on Natural Language Processing*. Online: Association for Computational Linguistics, November 2022.

[43] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al. Attention is All you Need. In: Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R. et al., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, vol. 30. Available at: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[44] W3Schools. *HTML Attributes* [online]. [visited 2022-01-16]. Available at: https://www.w3schools.com/html/html_attributes.asp.

[45] W3Schools. *The HTML DOM (Document Object Model)* [online]. [visited 2022-12-12]. Available at: https://www.w3schools.com/js/js_htmldom.asp.

[46] W3Schools. *HTML Elements* [online]. [visited 2022-01-16]. Available at: https://www.w3schools.com/html/html_elements.asp.

[47] W3Schools. *HTML Introduction* [online]. [visited 2022-01-16]. Available at: https://www.w3schools.com/html/html_intro.asp.

[48] Wang, W. and Gang, J. Application of Convolutional Neural Network in Natural Language Processing. In: *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE)*. 2018, p. 64–70. DOI: 10.1109/ICISCAE.2018.8666928.

[49] Zeng, A., Liu, X., Du, Z., Wang, Z., Lai, H. et al. GLM-130B: An Open Bilingual Pre-trained Model. In: *The Eleventh International Conference on Learning Representations (ICLR)*. 2023. Available at: https://openreview.net/forum?id=-Aw0rrrPUF.

Appendix A

User Interface of the Web Client

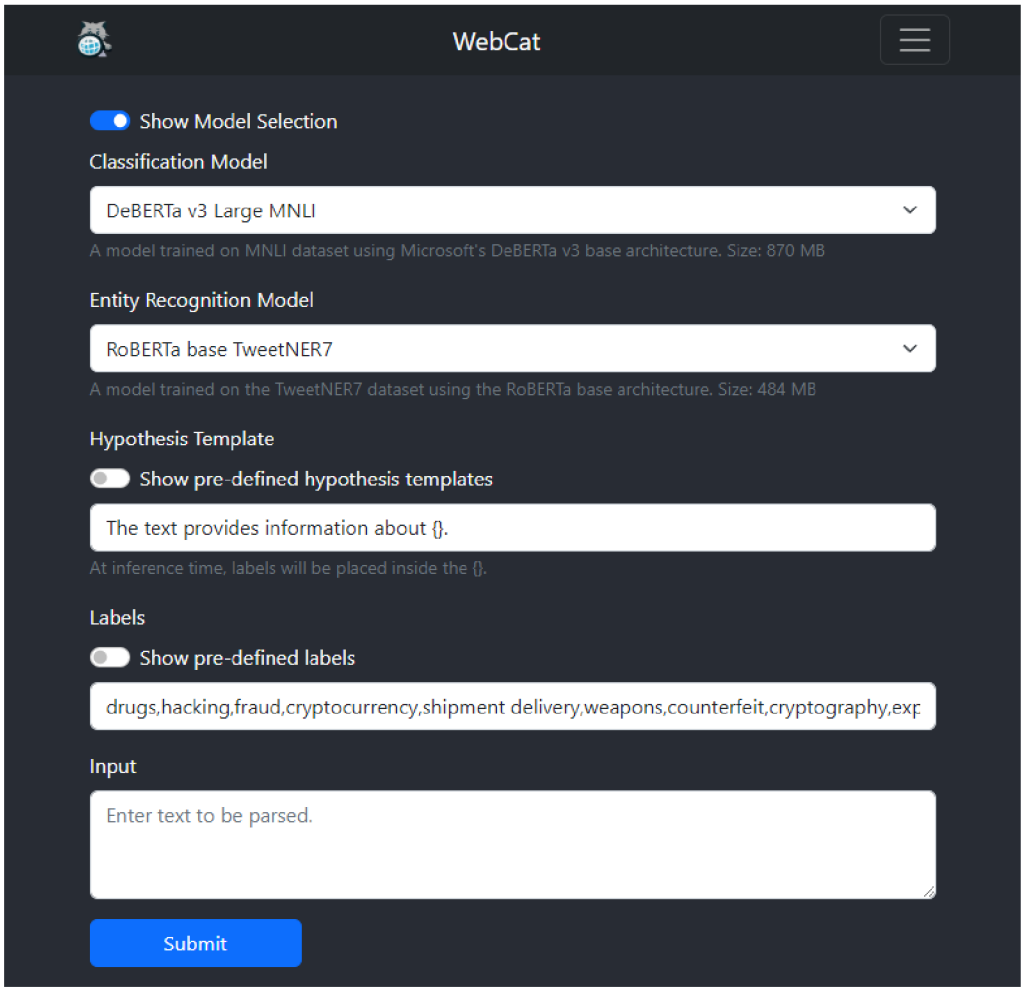


Figure A.1: Screenshot of the Interactive Parser tool, provided by the Web client. This tool provides an interface for interactive testing of different hypothesis templates, category labels and models by providing option to pass raw textual data for processing.

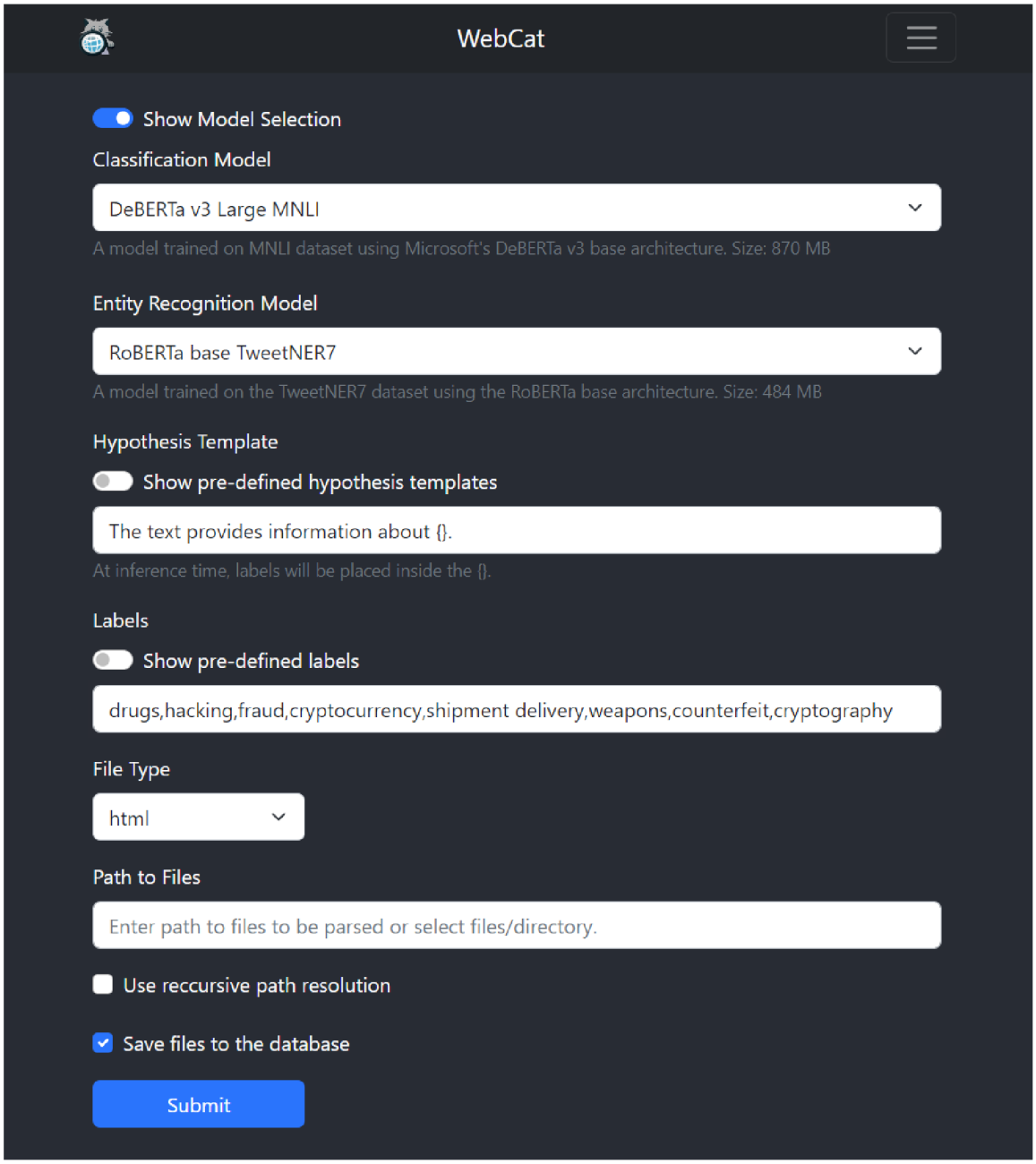


Figure A.2: Screenshot of the Files Parser tool, provided by the Web client. This tool provides the same options as the Interactive Parser, but accepts only files as the input for processing.

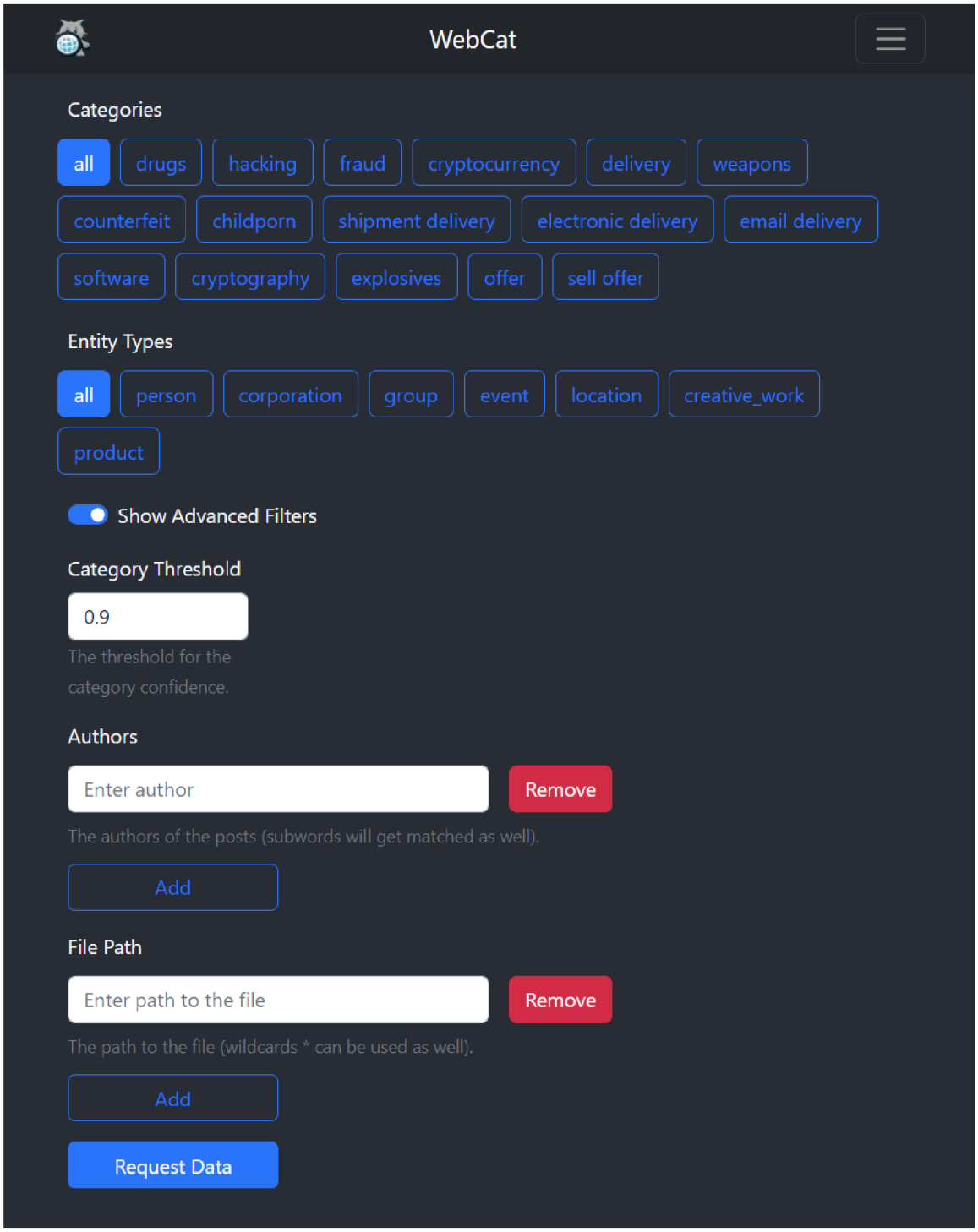


Figure A.3: Screenshot of the filtering options provided by the Data Viewer component. This feature requests the data that are stored in the database. The construction of the filtered query is handled by the API.

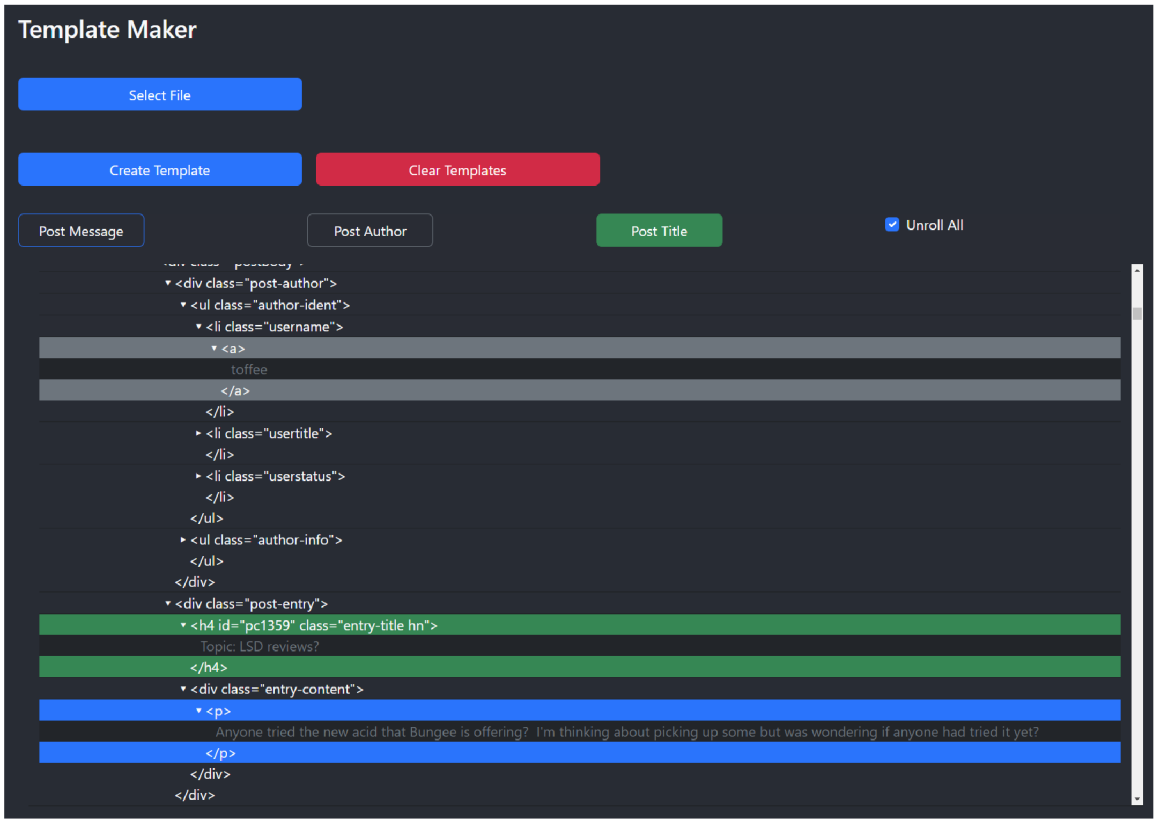


Figure A.4: Screenshot of the Template Maker tool in the WebCat Client application. This tool provides an easy way of creating templates directly from HTML files selected directly from the file system. Once the file is loaded, the virtual representation of the webpage is rendered and the user can manually annotate the content and create template.

Figure A.5: The WebCat application logo created with the aid of OpenAI's DALL-E model.

Appendix B

Attached Medium Contents

This thesis also includes a memory medium with the following contents:

- `src/` ... Source code for the WebCat system.
- `tex/` ... LaTeX source code for this thesis.
- `xreinm00.pdf` ... Copy of this thesis in PDF.

For the installation instructions, see the `README.md` file inside the `src/` directory. The attached contents were also uploaded to NextCloud[1] storage of the Faculty of Information Technology, Brno University of Technology.

[1] https://nextcloud.fit.vutbr.cz/s/5m87JSqXDgPJtmY