FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Department of Computer Science

Master-Thesis

Constructing a Knowledge Graph by extracting information from financial news articles

Rainer Gogel

Student-ID: 1272442

Advisor: Prof. Dr. Joerg Schaefer
Co-Advisor: Prof. Dr. Baris Sertkaya

November 12, 2024

Statutory Declaration

I herewith declare that I have completed this Master-Thesis independently, without making use of other than the specified literature and aids. All parts that were taken from published and non-published texts either verbally or in substance are clearly marked as such. This thesis has not been presented to any examination office before.

Frankfurt am Main, November 10, 2024

Rainer Gogel

Rainer Gogel

Signature

Acknowledgements

I would like to express my sincere gratitude to Professor Dr. Joerg Schaefer and Professor Dr. Baris Sertkaya for their willingness to supervise this Master-Thesis. I am particularly grateful to Professor Dr. Joerg Schaefer for stepping in as my primary supervisor at short notice, enabling me to escape an unpleasant situation.

I am also grateful to them for being a part of my academic journey and for their guidance along the way. Their expertise, passion for their field, and dedication to imparting knowledge through insightful and inspiring lectures and exercises have significantly enriched this journey. I am truly grateful for the wonderful learning experience they have provided.

Summary

Reading and understanding news articles takes time and effort and some readers are only interested in information that is relevant to them. This Master-Thesis project demonstrates how a user, who is only interested in news about certain companies and topics, can retrieve such information from financial news articles in an efficient manner.

The project's code extracts company names and their Coreference from the news text, classifies this text into different topic classes and stores this information in a Knowledge Graph. This way, the user can retrieve the desired information and discover complex relationships by querying the Knowledge Graph or by communicating with the Knowledge Graph's ChatBot (Graph Bot).

In the *Information Extraction Pipeline*, different approaches were studied, implemented in code and compared with each other. One of the key findings was that Generative LLMs can be used for a wide range of extraction tasks and that these models often outperform other, more traditional approaches.

Another key finding is that the retrieved information from the Graph Bot should also be more accurate than answers coming from a typical LLM ChatBot, even if that LLM ChatBot has a traditional RAG system attached to it. This is because the Graph Bot's response is more based and focused on the stored news articles in the Knowledge Graph and less on next-word probabilities and vector similarities.

Such a Knowledge Graph can be seen as an alternative RAG system that might replace the more commonly used vector databases there. This is supported by a recently published and influential research paper by Microsoft [33] that points in the same direction.

Glossary

Anaphora Expression, most often a pronoun or other pro-form, that depends upon an antecedent expression [92]. 50, 51, 55

ANN Artificial Neural Network. 20, 26, 27, 28, 35, 38, 55, 113

Antecedent Word that establishes the meaning of a pronoun or other pro-form [93]. 50, 51

BERT Bidirectional Encoder Representations from Transformers, a pretrained LLM [31]. 17, 29, 54, 74, 100

Cataphora Preceding expression, whose meaning is determined or specified by the later expression [94]. 51

CNN Convolutional Neural Network. 17

Content Words Content Words possess semantic content and contribute to the meaning of the sentence. Nouns, verbs, adjectives and adverbs are commonly considered Content Words [72]. 23

Coreference Two or more expressions refering to the same person or thing [73]. 4, 12, 50, 51, 52, 53, 54, 55, 56, 58, 62, 64, 65, 67, 68, 84, 106

Coreference Cluster Cluster of referencers in a semantic equivalence relation with the same referent [81]. 51, 55, 56, 57, 58, 67

Coreference Resolution Task of finding all expressions that refer to the same entity in a text [44]. 10, 12, 14, 15, 18, 34, 50, 52, 53, 54, 55, 68, 78, 106, 113

Corpus Collection or dataset of multiple unique documents. 20, 21, 22, 23

CPython Default implementation of the Python programming language. 18

CRF Conditional Random Fields, a type of Markov Network. 37, 38, 113

Cypher Query script language for Knowledge Graphs created by neo4j. 13, 88, 89, 92, 97, 98, 100, 101, 104, 105, 106, 114

Document Text that might contain multiple sentences such as a news article. 20, 21, 22, 23, 24

Document Term Matrix Tabular data structure where rows represent references to documents and columns represent the unique terms of a vocabulary. These unique terms can be words or tokens and are usually sorted alphabetically. The table entries are counts of these terms found in the respective document. 20, 21, 24, 71, 72

Function Words Function Words have little semantic meaning and mainly express grammatical relationships of words within a sentence [75]. 23

Generative LLM A generative LLM that generates text in response to a Prompt request sent to it. 4, 9, 10, 34, 35, 38, 39, 45, 46, 47, 48, 52, 56, 57, 60, 62, 63, 64, 67, 68, 73, 78, 79, 81, 83, 105, 106, 107

GPE Named entity of type Geo-Political Entity. 36

Graph Bot Chatbot that is attached to the Knowledge Graph. 4, 10, 13, 35, 101, 103, 104, 105, 106, 114

Hallucination Presentation of false or misleading information as facts by an LLM, see: [76]. 34, 35, 39

HMM Hidden Markov Model. 37, 113

IRI Internationalized Resource Identifier as defined by [90]. 86

ISIN International Securities Identification Number. 90, 94, 96

JSON JavaScript Object Notation. 64, 98

LDA Latent Dirichlet Allocation. 10, 72, 73

Lemmatization Process of converting inflected forms of a word to their lemma or semantic word root such as *is*, *was*, *were* to *be*, [77]. 17, 23

LLM Large Language Model. 4, 6, 7, 13, 23, 34, 35, 38, 39, 40, 48, 51, 56, 57, 60, 63, 64, 67, 72, 73, 79, 80, 81, 83, 101, 103, 104, 105, 107

LOC Named entity of type Location. 36, 37, 38, 39, 69

Mention Subjects or entities mentioned in a text to which coreferences exist. 50, 51, 52, 53, 54

N-Gram Sequence of N adjacent symbols such as letters in particular order [23]. 51

NER Named Entity Recognition. 9, 14, 15, 18, 34, 36, 37, 38, 39, 40, 41, 46, 47, 48, 49, 51, 56, 59, 62, 65, 67, 68, 84, 113

NLP Natural Language Processing. 16, 18, 20, 23, 34, 50, 69, 73

NMF Non-Negative Matrix Factorization, a Matrix decomposition method. 10, 71, 72, 73, 76, 114

ORG Named entity of type Organization. 17, 36, 38, 69

PER Named entity of type Person. 17, 36, 37, 38, 50, 69

POS Part of Speech or word class such as Noun, Verb, Adverb, etc. 17, 18, 51, 56, 57

Prompt Textual instruction sent to an LLM ChatBot in expectation to receive a desired response. 6, 34, 38, 39, 46, 47, 48, 56, 60, 62, 63, 68, 79, 80, 81, 82, 102, 103, 104, 105, 113

RAG Retrieval Augmented Generation. 4, 34, 35, 105, 106

REGEX Regular Expression. 16, 36, 39, 40, 41, 42, 43, 44, 45, 46, 48, 64, 65

ROBERTA Robustly Optimized BERT Pre-training Approach, a pretrained LLM [51]. 17

Span Chain of multiple tokens, often a spacy data object [34]. 18, 36, 38, 39, 45, 50, 51, 53, 54, 55, 58, 59, 65

SpanBERT Pretrained LLM, see: [42]. 54, 55

Stemming Process of converting inflected or derived words to their word stem by removing suffixes such as *fishing*, *fished*, *fisher* to *fish* [78]. The semantic meaning can change in the process such as *fisher* and *fish* are semantically different. 23

Stop Words Common words in a natural language that carry neither much semantic nor grammatical information. Stop Word types include articles, conjunctions, prepositions, pronouns and very common verbs such as *is* [79]. 22

SVD Singular Value Decomposition, a Matrix decomposition method. 10, 72, 114

TF-IDF Term-Frequency-times-Inverse-Document-Frequency. 23, 24, 25, 75, 76, 77

Token Word or Sub-Word such as a syllable or a letter, often a spacy data object [34]. 18, 20, 38, 39, 45, 50, 51, 53, 54, 55, 56, 59, 65

Transformer ANN architecture laid out in the seminal *Attention is all you need* paper [98]. 28, 29, 31, 34, 38, 56, 113

Vocabulary Set of unique words in a corpus or, more generally, in a natural language such as English. 20, 21, 22, 23, 25, 72, 75

Contents

[1]This section was mainly taken from [36]

[2]Part of the Python code for this section was adopted from [37]

1. Introduction

1.1. Overview

The goal of this Master-Thesis project is to extract structured information from unstructured text and store this information in a Knowledge Graph to facilitate information retrieval. The unstructured text are financial news articles and the target information to be extracted is information about certain corporate entities or companies.

To extract the information, an *Information Extraction Pipeline* is built that contains three main components:

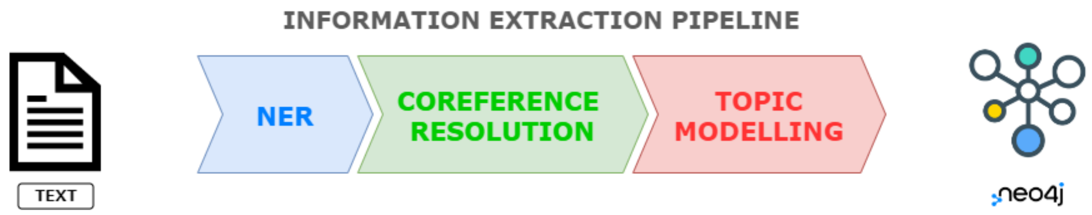


Figure 1.1.: Information Extraction Pipeline

- **NER**: A Named Entity Recognition component that identifies the desired company names in the text
- **Coreference Resolution**: A Coreference Resolution component that detects references to previously identified company names
- **Topic Modelling**: A Topic Modelling component that classifies those sentences in the text, that contain company names and their coreferences, into different topic classes

The *Information Extraction Pipeline* yields sentences that contain company names, their Coreferences and the respective topic for each of these sentences. This information is then stored in a neo4j [84] Graph database according to the following schema:

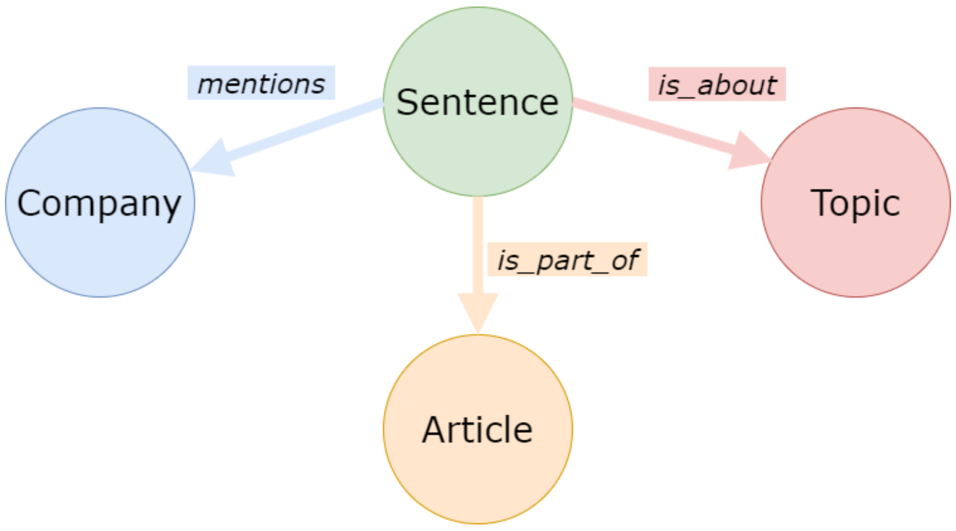


Figure 1.2.: Knowledge Graph Schema

There are three Node types in the Knowledge Graph:

- **Sentence**: the *Sentence* in which a *Company* is mentioned
- **Topic**: the *Topic* or overarching theme of this *Sentence*
- **Company**: the *Company* entity whose name is mentioned in the *Sentence*
- **Article**: the *Article* of which the *Sentence* is part of

A *Sentence* Node thus has three relationships to other Nodes in the Knowledge Graph:

- **is_about**: a *Sentence* is about a certain *Topic*
- **is_part_of**: a *Sentence* is part of an *Article*
- **mentions**: a *Sentence* mentions a *Company*

This structured representation of the unstructured text does not only allow for fast retrieval of concrete information, but can also reveal complex relationships between multiple companies, topics and articles.

The stored information can be retrieved via Cypher queries that can also be generated by an LLM-based Graph Bot that first converts human text to Cypher queries and then the results back to human-readable text.

1.2. Thesis and Code

This Master-Thesis consists of two separate parts, the written thesis and the accompanying Python code.

Python Code The Python code can be found in the following GitHub repository:

`https://github.com/rainergo/UASFRA-MS-Thesis`

The code could be used as a template for a production use case, but mainly serves to support this Master-Thesis and to present the process rather than the result.
All functionalities of the code are aggregated and condensed in the Jupyter Notebook

MAIN.ipynb

located in the root directory of the repository and explained in Appendix A. This Notebook runs the entire pipeline from text to Knowledge Graph.

Target Use Case The Python code is targeted at a user with a particular focus on financial news of publicly listed European companies in German language. This focus is driven by my professional background in finance and personal interest in this field.

1.3. Thesis Outline

This thesis will be divided into multiple chapters. In the chapter after this introduction, I describe the source and nature of the text data.
Thereafter, I will describe the spacy [34] Python library and how it was used, followed by an introduction to text representation.
In the fourth chapter, I start with a description of Named Entity Recognition (NER) and how that pipeline component was implemented in code.
The same approach applies to the Coreference Resolution and Topic Modelling chapters thereafter.
In the Knowledge Graph chapter, I show how the previously extracted data is fed into the neo4j database and how information retrieval algorithms can reveal some complex relations between news articles, companies and topics.
I conclude with what I learned from this project and where in hindsight I would approach tasks differently.

2. Project Setup

This chapter will describe the project's data sources and the Python library *spacy*, which was extensively used for the NER and Coreference Resolution components in the *Information Extraction Pipeline*.

2.1. Data

The data consists of *News Articles* and *Company Data*. The dataset requirements are derived from the project's domain focus as outlined in section 1.2. This requires that

- news articles must be in German language.
- news articles should mention companies, at least potentially.
- companies must be publicly listed on a stock exchange.
- companies must be based in Europe.
- such company and news articles data must be publicly available.

2.1.1. News Articles

Many news article datasets are available on platforms such as Kaggle [8], but it seems that only a few of them are in German [8]. Most of the German news datasets that I analyzed are either not rooted in the financial domain, outdated for copyright reasons or deal with macroeconomic policies or broader economic trends rather than company news.

As I could not find proper data that fits the outlined objectives, I decided to scrape the information from financial news providers.

Sources The data was scraped from the websites of *EQS-News* [19] and *dpa-AFX* [18], both business news hubs that aggregate multiple news sources. The text in the html-files were converted to text files, cleaned and read into pandas [9] DataFrames. The type of the news articles includes *ad-hoc news*, *press releases*, *corporate news* and *regulatory filings*. These could be published by *companies* themselves or *media outlets* such as newspapers, news agencies or other broadcasters.

Data Description Of the news articles, the article text, the article title and some other metadata such as publication dates, authors, etc. were scraped. The news articles contained many irregularities, errors, misspellings and phrases that were not part of the actual news.
Multiple regular expression (REGEX) patterns and techniques were applied to clean the text. Most of the errors and undesired phrases of the text could be removed, but not all. So the subsequent models must handle them.
The news article dates range from *May 1, 2023* through *May 31, 2024* and its frequency is daily. The daily data was aggregated on a monthly basis and the pandas DataFrames were stored as Apache parquet [10] files.

2.1.2. Company Data

OpenBB [15] is an open source framework that aspires to replicate the functionalities of the Bloomberg Terminal, a costly financial service product by the media company Bloomberg [7]. OpenBB offers a wide range of functionalities, connects data providers and is the data source for the company data used in the project. Data for companies, that are publicly listed on bourses in

- France
- Germany
- Great Britain
- Spain
- Italy
- The Netherlands

and that are members of the main stock indexes there, were downloaded and stored in an Apache parqet file. This file contains unique company names and their stock ticker symbols for slightly above 2500 companies.

2.2. Spacy

Spacy [34] is a modular, open-source Python library that is commonly used for a wide range of NLP-related tasks. For popular languages like English and German, spacy offers pre-trained pipelines (see Fig.2.1) in different sizes and versions.

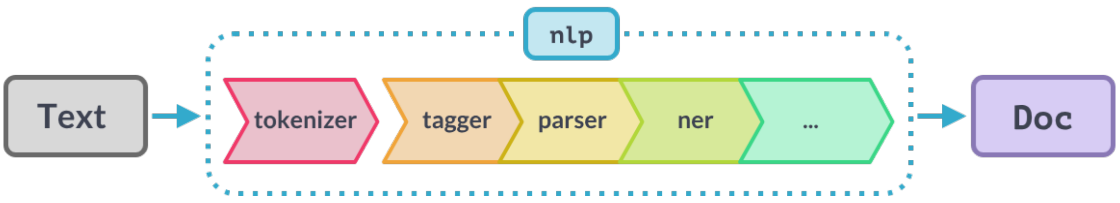


Figure 2.1.: *Spacy Pipeline*

The model sizes range from *Small* to *Medium* and *Large*, with bigger models usually performing better than smaller ones.
The spacy-trained Transformer model for English is based on ROBERTA [51], the one for German on BERT-German [31]. The TOK2VEC [34] embeddings for the non-Transformer models (suffixes: sm, md, lg) are memory-efficient Multi-Hash embeddings [57] with a CNN encoding layer [34].

Pipeline Components After downloading the models, by default, the modular pipeline consists of a:

- **Tokenizer**: Segments text into tokens
- **POS-Tagger**: Assigns POS tags such as *NOUN* or *VERB*
- **Dependency-Parser**: Assigns syntactic dependency labels between POS-tags and sets sentence boundaries
- **Lemmatizer**: Assigns base forms or lemmas (see: Lemmatization)
- **Entity-Recognizer**: Detects and labels named entities such as ORG or PER
- **Custom Components**: Proprietary components implemented by the user
- **Pipeline Plugins**: External applications that are developed by others as a spacy pipeline component

The *Dependency-Parser* and *Entity-Recognizer* modules in the pipeline were separately trained with task-specific datasets and plugged back into the pipeline [34] (see Fig.2.1).

Custom Component With a spacy *Custom Component*, users can implement their own custom algorithms and functions. Compared to a pure Python implementation, these *Custom Components* are usually faster as they are optimized with CPython in the background. This functionality was extensively used in the project as the NER and Coreference Resolution pipeline components (see Fig.1.1) were implemented with such *Custom Components*.

Pipeline Plugins Spacy *Pipeline Plugins* are applications and standalone packages that are developed by other software developers to be used as a spacy pipeline component. Such registered plugins enhance the spacy ecosystem and can easily be added by referencing it in the *add_pipe()*-function. One such plugin studied in the project is the package *Crosslingual-Coreference* (”xx_coref”) [21] that was loaded with the

nlp.add_pipe(”xx_coref”)

function.

Custom Extensions Users can also attach custom extensions to spacy’s Tokens and Spans in a *Doc*-Object (see Fig.2.1) which allows them to tag or label certain words or expressions in the text to further process them later. In the project, such custom extensions were also used extensively. For instance: For a company identified in the Span or Token of a document (*Doc*-object), the company name and company symbol (stock tocker symbol) were attached to the custom extensions *comp_name* and *comp_symbol* as in the following example:

doc._.comp_name = ”Adidas AG”

doc._.comp_symbol = ”ADS.DE”

Later on, all companies in a *Doc*-object can be collected and further processed.

Modularity and Custom Pipeline The *Tokenizer*-component is always required in the pipeline, but all other components (see Fig.2.1) can easily be removed from it if they are not needed for a specific NLP-task. For instance, if only a POS-tagger is needed to identify *Nouns*, it can be enabled with the *select_pipes()*-method which also disables all other components except for the *Tokenizer*:

nlp.select_pipes(enable=”tagger”)

2.2.1. Usage

The spacy *Custom Pipeline* was build with Python modules that can be found in the following directory:

src/B_spacy_pipeline

In the *spacy_pipe_build* module and *SpacyPipeBuild* class, the method *build_pipe()* (Python-Code 1) controls the pipeline's components:

```python
    def build_pipe(self):
        if self.spacy_task != SpacyTask.BASIC:
            self.func_init_extensions()
        match self.spacy_task:
            case SpacyTask.ALL:
                self.nlp.select_pipes(enable=[self.vectorizer.factory_name])
                self.api_parser()
                match self.ner_method:
                    case ExtractionType.TRADITIONAL:
                        self.func_own_regex_search()
                        # self.api_entity_ruler()
                        self.func_comp_name_token_regex_match()
                    case ExtractionType.PRETRAINED:
                        self.api_ner()
                match self.coref_method:
                    case ExtractionType.PRETRAINED:
                        self.func_coref_resolve_pretrained()
                    case ExtractionType.GENERATIVE_LLM:
                        self.func_coref_resolve_generative()
                    case _:
                        raise ValueError('Cases other than ExtractionType.PRETRAINED and
                        ↪  ExtractionType.GENERATIVE_LLM are not supported yet.')
                self.func_sentencizer()  # Must be in for correct sentence splitting
```

Python-Code 1: Function: build_pipe()

A pipeline component can be replaced by substituting it with the desired module function depending on the specific task. All module functions are defined in the *SpacyPipeBuild* class.

Other Python Libraries Beside spacy, other important Python libraries were used, but these will be described within their particular scope in the subsequent chapters.

3. Text Representation

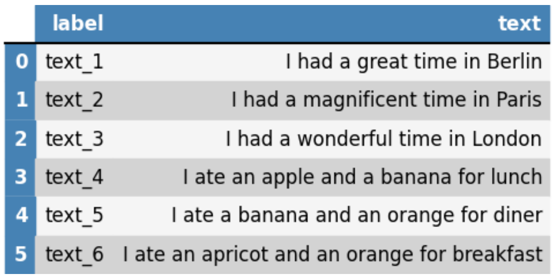
3.1. Overview

The key question in text processing and NLP is how to encode text into numbers so that computers can understand them. The process is also known as *Vectorization* [55] and there are two main approaches to vectorize text.
The first and more traditional approach is to encode word or Token occurrence counts. A more modern approach is to train Artificial Neural Networks (ANN) to get text embeddings that also incorporate syntax and semantics. In the next subsections, I will explain these encoding techniques as they form the foundation for most of the subsequent chapters.

3.2. Definitions

In NLP, the terms Corpus, Vocabulary and Document are often interpreted differently which requires clarification. In this thesis, a Document is defined as text that might contain multiple sentences such as a news article in the data described above. All news articles in the entire data set constitute the Corpus which refers to a collection of documents. All unique words in a Corpus represent the Vocabulary. A Document Term Matrix is a tabular data structure where rows represent references to documents and columns represent the unique terms of a vocabulary. Such terms or Tokens can either be words or sub-words and in a Document Term Matrix represent its features. The data entries in the Document Term Matrix are the feature values and show if or how often the term appears in the respective document.

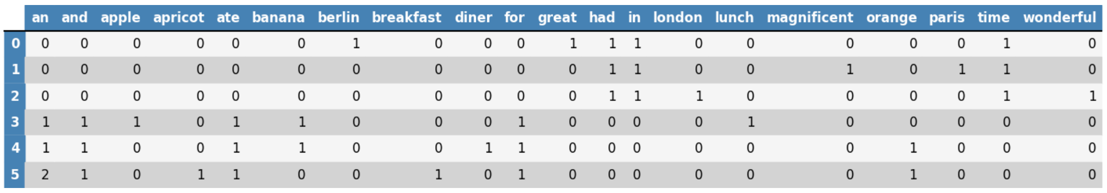
A simplified example can be seen in Fig.3.1 and in Fig.3.2:



	label	text
0	text_1	I had a great time in Berlin
1	text_2	I had a magnificent time in Paris
2	text_3	I had a wonderful time in London
3	text_4	I ate an apple and a banana for lunch
4	text_5	I ate a banana and an orange for diner
5	text_6	I ate an apricot and an orange for breakfast

Figure 3.1.: Sample Documents

The text in each row is a Document though in this example only contains one sentence each and the Corpus here constitutes all six Documents.



	an	and	apple	apricot	ate	banana	berlin	breakfast	diner	for	great	had	in	london	lunch	magnificent	orange	paris	time	wonderful
0	0	0	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0	1	1	0
2	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	1	1
3	1	1	1	0	1	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0
4	1	1	0	0	1	1	0	0	1	1	0	0	0	0	0	0	1	0	0	0
5	2	1	0	1	1	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0

Figure 3.2.: Document Term Matrix

The Document Term Matrix is shown in Fig.3.2 where the header row represents the Vocabulary. The data entries in the table are the counts for each Vocabulary term in each Document.

3.3. Traditional Methods

3.3.1. One-Hot-Encoding

Each Document in the Fig.3.1 can be represented by the respective row in Fig.3.2 when interpreted as a vector.

$$\vec{document}_0 = [\,0\;0\;0\;0\;0\;0\;1\;0\;0\;0\;1\;1\;1\;0\;0\;0\;0\;0\;1\;0\,]$$

If such vectors only contain binary ones and zeros, i.e. if they do not count word occurrences but only indicate if a word is present or not, such vectors are called One-Hot encoded vectors.

3.3.2. Bag-of-Words

The Document Term Matrix shown in Fig.3.2 also contains word counts as can be seen from the Document in the last row:

$$\vec{document}_5 = [\,2\;1\;0\;1\;1\;0\;0\;1\;0\;1\;0\;0\;0\;0\;0\;0\;1\;0\;0\;0\,]$$

Such vectors are known as Bag-of-Word vectors as they account for the frequency of a word in a Document.

3.3.3. Document Similarities

After vectorization, it is possible to calculate similarities between sentences mathematically. One common way to do that is to calculate the cosine similarity of the desired vectors which is the cosine of the angles between the two vectors [68]:

$$\textit{cosine similarity}_{ab} = \frac{\vec{a} \cdot \vec{b}^T}{|\vec{a}| \cdot |\vec{b}|} \tag{3.1}$$

For instance: The cosine similarity between document 0 and document 1 of the Sample Documents (Fig.3.1) is calculated by:

$$\textit{cosine similarity}_{01} = \frac{\overrightarrow{document0} \cdot \overrightarrow{document1}^T}{|\overrightarrow{document0}| \cdot |\overrightarrow{document1}|} = 0.60$$

The high cosine similarity of 0.60 is also attributed to the fact that both documents contain the same words

$$[\ I,\ had,\ a,\ time,\ in\ ]$$

some of which are considered Stop Words that do not carry much semantic information.

3.3.4. Feature Dimension Reduction

Usually a Document contains far fewer terms than a Vocabulary which results in sparse vectors where many features have zero values. Sparse vectors not only increase the computational complexity and the risk to overfit a Machine Learning model, but also are a sign that some important information in the data is missing [67]. For this reason, it is common practice to reduce the Vocabulary and so the vector and feature dimension.

Remove Stop Words

Stop Words include articles, conjunctions, prepositions, pronouns and very common verbs such as *is*. They appear in most Documents and thus do not contribute to distinguish them. Beside commonly known Stop Words in a Corpus, there might be additional domain-specific Stop Words such as the word for the currency unit *Euro* in finance-related Documents.

Remove Function Words

NLP differentiates between Content Words and Function Words. Function Words help to construct the grammatical relationships in a sentence whereas Content Words serve to carry their semantic meaning. In Topic Modelling, the target is to extract semantics. Thus, Function Words are less relevant and can be removed.

Reduce words to their Lemmas and Stems

Words appear in many different inflectional forms such as verb tenses (past, present and future) and pluralizations. Lemmatization is the process to convert all such forms to the lemma or root of the word [77]. This process removes some contextual information but usually well preserves the semantic meaning of the text. Stemming reduces a word to its word stem by clipping off some suffix chars. Contrary to Lemmatization, this process can change the semantic meaning of the word (for instance: *fisher*→*fish*) and thus must be applied carefully. Nevertheless, both methods drastically reduce the size of the Vocabulary.

Others

Other methods to reduce the Vocabulary include the removal of non-words such as punctuation chars and numbers from the text. If nouns in a specific domain carry proportionally more information than other words, it might be beneficial to remove all non-noun words.

3.3.5. TF-IDF

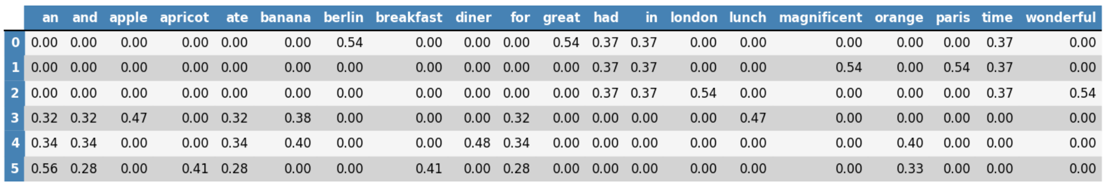
It is known from Information Theory that low probability terms carry much more information than high probability terms [74]. Low probability terms are those words that appear very seldom in a Corpus or in the Documents that were used to train commonly used LLMs. If such low probability words appear much more frequently in a certain Document than in the related Corpus, this might be an indication that this word is important to understand the context of the particular Document. This is the concept of TF-IDF:

$$\mathit{TF\text{-}IDF} = TF * IDF$$

$$TF = \frac{\text{number of times the } \textbf{term} \text{ appears in the document}}{\text{total number of terms in the document}}$$

$$IDF = log\ (\frac{\text{number of documents in the corpus}}{\text{number of documents in the corpus containing the } \textbf{term}}) \tag{3.2}$$

The Term Frequency (TF) is the frequency a target word appears in a given Document. In a Bag-of-Word approach (Sec.3.3.2), this number would go into the Document Term Matrix, but in TF-IDF, it is first multiplied by the Inverse Document Frequency (IDF). The IDF is the logarithm of the number of all Documents divided by the number of those Documents that contain the target word. If the frequency of the target word in all Document is low, then the IDF will be high and this will increase the weight of the target word in the Document Term Matrix. This can be seen from the Document Term Matrix in Fig.3.3 when TF-IDF vectorization was used:



	an	and	apple	apricot	ate	banana	berlin	breakfast	diner	for	great	had	in	london	lunch	magnificent	orange	paris	time	wonderful
0	0.00	0.00	0.00	0.00	0.00	0.00	0.54	0.00	0.00	0.00	0.54	0.37	0.37	0.00	0.00	0.00	0.00	0.00	0.37	0.00
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.37	0.37	0.00	0.00	0.54	0.00	0.54	0.37	0.00
2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.37	0.37	0.54	0.00	0.00	0.00	0.00	0.37	0.54
3	0.32	0.32	0.47	0.00	0.32	0.38	0.00	0.00	0.00	0.32	0.00	0.00	0.00	0.00	0.47	0.00	0.00	0.00	0.00	0.00
4	0.34	0.34	0.00	0.00	0.34	0.40	0.00	0.00	0.48	0.34	0.00	0.00	0.00	0.00	0.00	0.00	0.40	0.00	0.00	0.00
5	0.56	0.28	0.00	0.41	0.28	0.00	0.00	0.41	0.00	0.28	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.00

Figure 3.3.: Document Term Matrix with TF-IDF

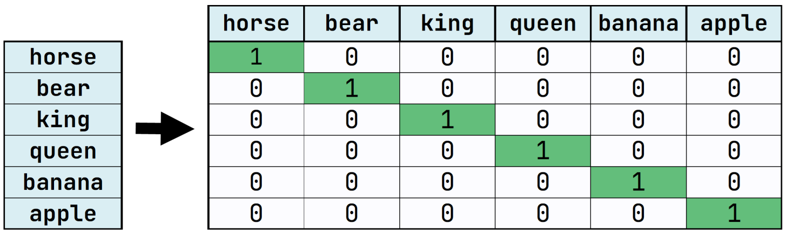
Words that appear only once in all Documents (for instance the word *apple*) have a higher weight in the TF-IDF Document Term Matrix than words that appear very often (for instance the word *ate*).

3.4. Word Embeddings[1]

3.4.1. Static Word Vectors

As explained in section 3.3.1, words can be represented as one-hot encodings as also shown in the example in Fig.3.4.

[1]This section was mainly taken from [36]



horse
bear
king
queen
banana
apple

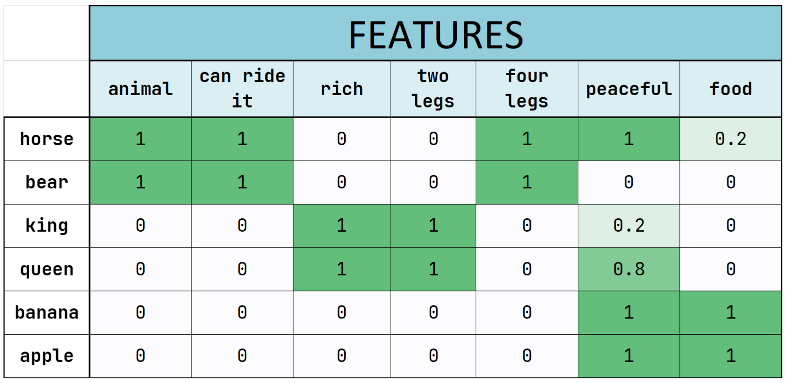
horse	bear	king	queen	banana	apple
1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1

Figure 3.4.: One-Hot example

The first disadvantage of this approach is the computational inefficiency as each word in our simple example in Fig.3.4 would require a sparse six-column vector with five 0's and only one 1. The second disadvantage is that this approach does not encode syntactical or semantic word similarities as similar words would have different and unrelated vector representations.

Manually crafted features

To encode syntactical or semantic word commonalities, one could think about word characteristics or attributes and manually encode the magnitude of those attributes for each word as shown in Fig.3.5. A *horse* is an *animal*, one *can ride it*, it has *four legs* and is usually *peaceful*.
This is very similar to the Vocabulary in the TF-IDF and *Bag-of-Word* approach outlined in Section 3.3.5, but differes from it in that the features here do not represent actual words but contextual concepts. The feature *animal* only has values in its respective vector position, not when the *word animal* itself appears, but only when the *contextual concept* of an *animal* appears.



	FEATURES						
	animal	can ride it	rich	two legs	four legs	peaceful	food
horse	1	1	0	0	1	1	0.2
bear	1	1	0	0	1	0	0
king	0	0	1	1	0	0.2	0
queen	0	0	1	1	0	0.8	0
banana	0	0	0	0	0	1	1
apple	0	0	0	0	0	1	1

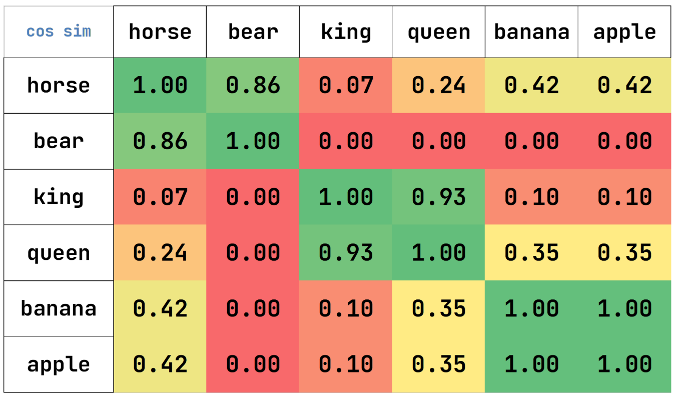
Figure 3.5.: Manually Crafted Features

The word *horse* would then be represented as a vector of values for each of these handcrafted features (Fig.3.6):

```
vec_horse = [1, 1, 0, 0, 1, 1, 0.2]
```

Figure 3.6.: Word vector for the word *horse*

This approach ensures that related words are represented similarly as their values in the respective vector position are close to each other. Such similarities can also be calculated numerically by applying the cosine similarity method, see Equation 3.1. The cosine similarities between each pair of words in Figure 3.5 are depicted in Fig.3.7.



cos sim	horse	bear	king	queen	banana	apple
horse	1.00	0.86	0.07	0.24	0.42	0.42
bear	0.86	1.00	0.00	0.00	0.00	0.00
king	0.07	0.00	1.00	0.93	0.10	0.10
queen	0.24	0.00	0.93	1.00	0.35	0.35
banana	0.42	0.00	0.10	0.35	1.00	1.00
apple	0.42	0.00	0.10	0.35	1.00	1.00

Figure 3.7.: Exemplary Cosine Similarity Matrix based on Fig.3.5.

The cosine similarity for the word pair *horse* and *apple*, for instance, yields a value of 0.42. For the word pair *apple* and *banana* however, this value is 1.0, owing to the fact that both are fruits and in our simple example have the same feature values at each index position of their word vector.

Learned features

Manually crafting feature values for every word in a vocabulary is cumbersome at best. Better than crafting features by hand is to have a machine learning model to learn these features and feature values. This is the *word2vec* approach Mikolov et al. proposed in their seminal papers in 2013 [56, 55]. A shallow, two-layer ANN (Fig.3.8) is trained with sentences that contain a masked word to be predicted (target or dependent variable) based on its non-masked adjacent words (input or independent variables) in that sentence.

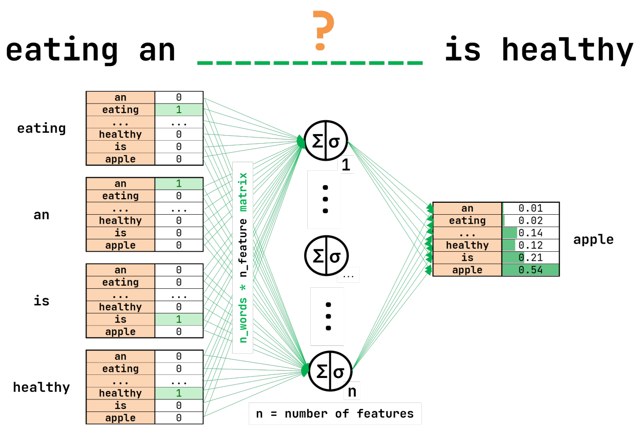
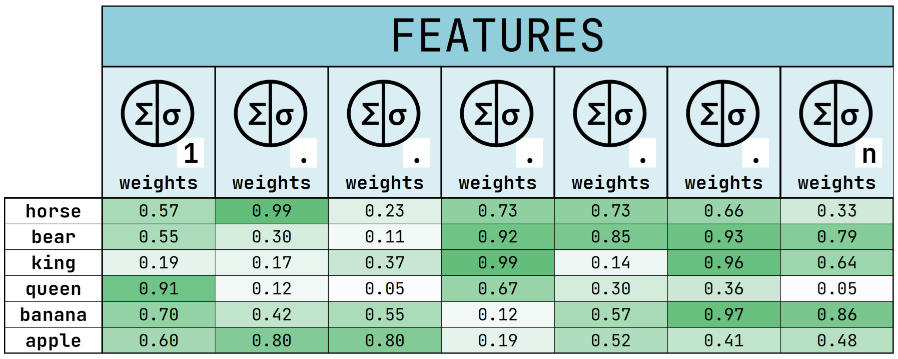


Figure 3.8.: word2vec: Masked words learned by a shallow, two-layer ANN.

The number of learned features depends on the number of neurons in the middle layer and the learned weights represent the respective feature values. The learned features do not have names as before (like *animal*, *rich*, *food*, etc.) and thus cannot be interpreted semantically by humans (easily) (Fig.3.9).



	FEATURES						
	1 weights	. weights	. weights	. weights	. weights	. weights	n weights
horse	0.57	0.99	0.23	0.73	0.73	0.66	0.33
bear	0.55	0.30	0.11	0.92	0.85	0.93	0.79
king	0.19	0.17	0.37	0.99	0.14	0.96	0.64
queen	0.91	0.12	0.05	0.67	0.30	0.36	0.05
banana	0.70	0.42	0.55	0.12	0.57	0.97	0.86
apple	0.60	0.80	0.80	0.19	0.52	0.41	0.48

```
 vec_horse =  [0.57, 0.99, 0.23, 0.73,   ... , 0.73, 0.66, 0.33]
  vec_bear =  [0.55, 0.3, 0.11, 0.92,  ... , 0.85, 0.93, 0.79]
  vec_king =  [0.19, 0.17, 0.37, 0.99,   ... , 0.14, 0.96, 0.64]
 vec_queen =  [0.91, 0.12, 0.05, 0.67,   ... , 0.3, 0.36, 0.05]
vec_banana =  [0.7, 0.42, 0.55, 0.12,  ... , 0.57, 0.97, 0.86]
 vec_apple =  [0.6, 0.8, 0.8, 0.19,   ... , 0.52, 0.41, 0.48]
```

Figure 3.9.: The learned parameters/weights represent the feature values.

The problem with static word embeddings

The learned word vectors are also known as word embeddings and work well for tasks such as measuring similarities between individual words. But they often fail when the scope goes beyond just words towards the semantic meaning of whole sentences. A sentence is simply not just a chain of individual and independent words, but a construct containing interdependencies.

One of these word-wise interdependencies are homophones, i.e. words that are spelt the same but have different meanings. The meaning of the word *bank* in the sentence *he withdraws money from his bank* is strongly affected by the surrounding words *withdraws* and *money* as they clearly suggest that *bank* in that sentence refers to a financial institution. In contrast, the meaning of the same word *bank* in the sentence *he was fishing in the river from a sand bank* is strongly affected by the words *river* and *sand* (Fig.3.10).

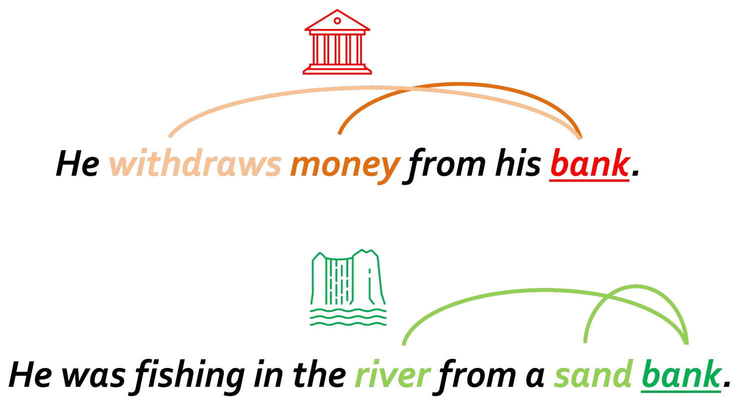

He withdraws money from his bank.
He was fishing in the river from a sand bank.

Figure 3.10.: Homophone *bank* dependent on its context

When humans read these sentences, they derive the meaning of the word *bank* by paying *attention* to these adjacent words [54].

There are more interdependencies in sentences than homophones, but in general it can be said that the meaning of a word depends on its respective context. Word embeddings such as word2vec [56, 55] are static in the sense that they treat equal words alike without taking their context into account. This is one of the reasons why static word embeddings such as word2vec often fail to understand the meaning of sentences.

3.4.2. Contextual Word Embeddings

It can surely be said, that in 2017 a single research paper changed the world: *Attention is all you need* by Vasvani et al. [98] laid out a new ANN-architecture dubbed Transformer that is able to embedd text contextually. Transformers embrace the human *attention* principle discussed above, i.e. the ability to infer the meaning of a word by paying *attention* to its adjacent words in that sentence.

Transformer Architecture

The Transformer architecture consists of *Encoder* (left blocks in Fig.3.11) and *Decoder* (right blocks in Fig.3.11) stacks. The Encoder and Decoder stacks each have N layers depending on the model version and size. In the basic pre-trained BERT [31] model (*bert-base-uncased*), which was one of the first models to implement the Transformer architecture, N is equal to 12, i.e. has 12 layers.

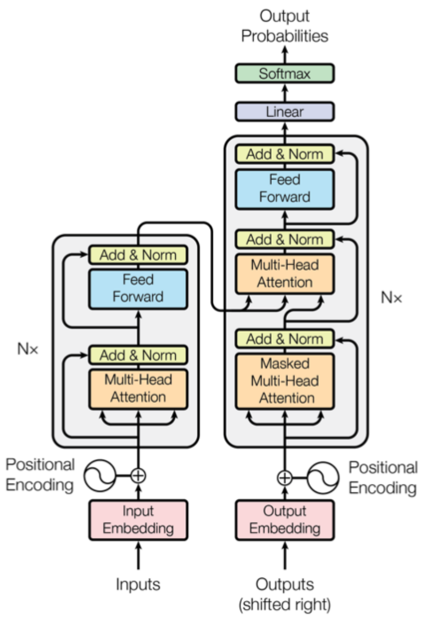


Figure 3.11.: Transformer Architecture: Encoder and Decoder

For encoding text, only the Encoder part of the Transformer is needed. For generating text, the Decoder part is also needed (Fig.3.11).

Self-Attention

The central element in the architecture of Transformers is the *self-attention mechanism* in the Multi-Head-Attention module depicted in Fig.3.12.

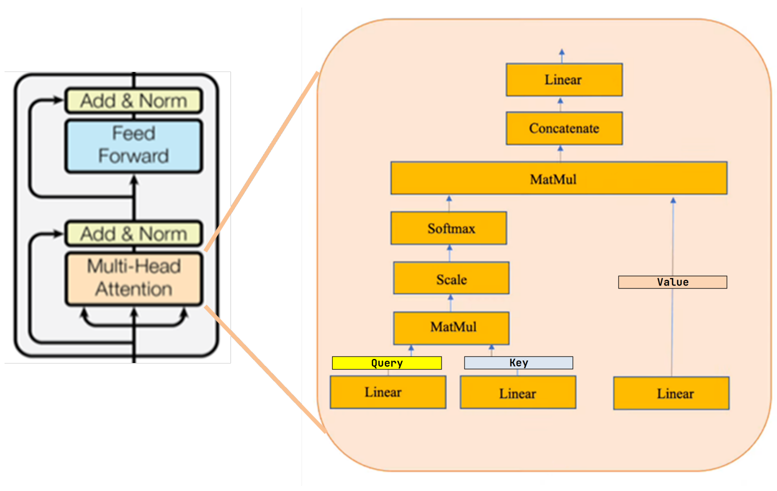
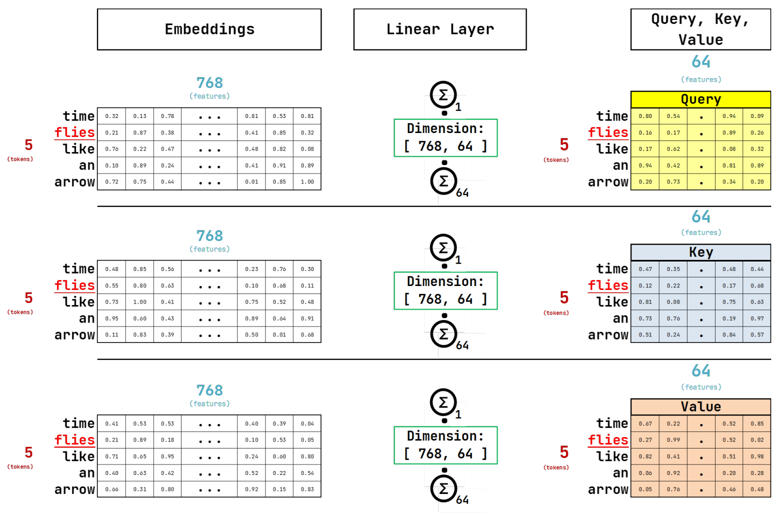


Figure 3.12.: Multi-Head Attention. Image from: [31] and [59]

In the Multi-Head-Attention module, three different matrices named *Query*, *Key* and *Value* are learned in the training phase based on one and the same static embeddings of the input words (Fig.3.13).



time	0.32	0.13	0.78	. . .	0.81	0.53	0.81
flies	0.21	0.87	0.38	. . .	0.41	0.85	0.32
like	0.76	0.22	0.47	. . .	0.48	0.82	0.08
an	0.10	0.89	0.24	. . .	0.41	0.91	0.89
arrow	0.72	0.75	0.44	. . .	0.01	0.85	1.00

	Query				
time	0.80	0.54	.	0.94	0.09
flies	0.16	0.17	.	0.89	0.26
like	0.17	0.62	.	0.08	0.32
an	0.94	0.42	.	0.81	0.89
arrow	0.20	0.73	.	0.34	0.20

time	0.48	0.85	0.56	. . .	0.23	0.76	0.30
flies	0.55	0.80	0.63	. . .	0.10	0.68	0.11
like	0.73	1.00	0.41	. . .	0.75	0.52	0.48
an	0.95	0.60	0.43	. . .	0.89	0.64	0.91
arrow	0.11	0.83	0.39	. . .	0.50	0.01	0.68

	Key				
time	0.47	0.35	.	0.48	0.44
flies	0.12	0.22	.	0.17	0.68
like	0.81	0.08	.	0.75	0.63
an	0.73	0.76	.	0.19	0.97
arrow	0.51	0.24	.	0.84	0.57

time	0.41	0.53	0.53	. . .	0.40	0.39	0.04
flies	0.21	0.89	0.18	. . .	0.10	0.53	0.05
like	0.71	0.65	0.95	. . .	0.24	0.60	0.80
an	0.40	0.63	0.42	. . .	0.52	0.22	0.54
arrow	0.66	0.31	0.80	. . .	0.92	0.15	0.83

	Value				
time	0.67	0.22	.	0.52	0.85
flies	0.27	0.99	.	0.52	0.02
like	0.82	0.41	.	0.51	0.98
an	0.06	0.92	.	0.20	0.28
arrow	0.05	0.76	.	0.46	0.48

Figure 3.13.: $InputEmbeddings \cdot LinearLayer = Query, Key, Value$

The next step in the *Multi-Head Attention* module during both, training and inference (in a downstream task), is the matrix multiplication depicted in Fig.3.14 to produce the *Attention Filter*. It has the same dimension ([5, 5] for the given sentence) as the length of the input text.

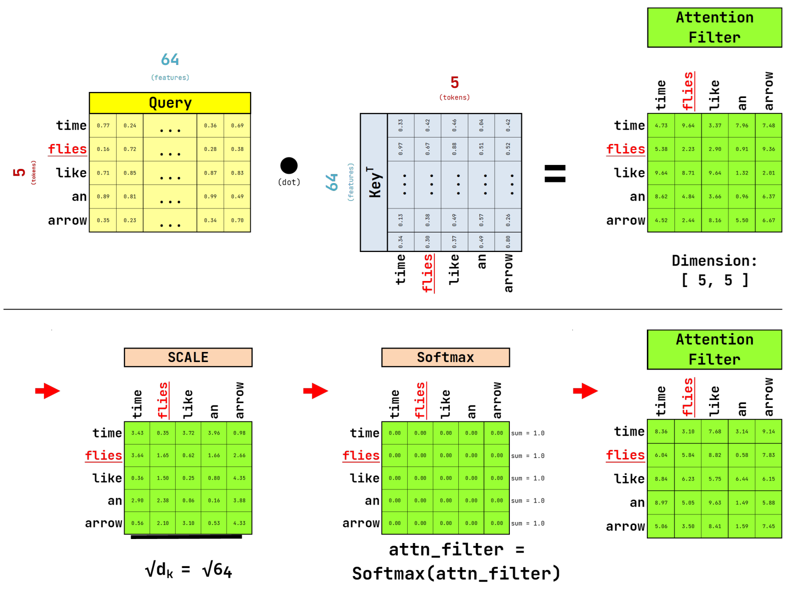
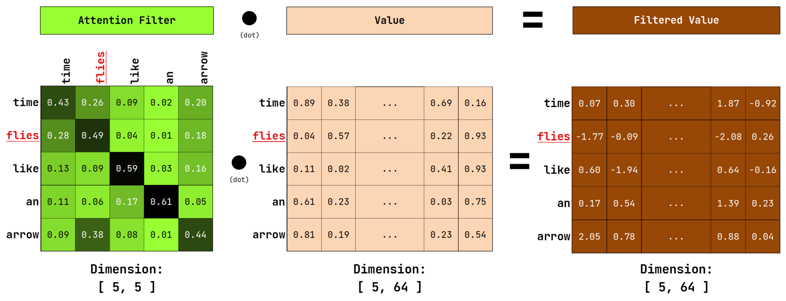


Figure 3.14.: *Query* · *Key* = *Attention Filter*

The task of the *Attention Filter* is to identify those adjacent words in a sentence that help determine the context for and the meaning of a given word.

In the sample sentence *time flies like an arrow*, a human reader would probably pay attention to the words *arrow* and *time* to determine the context and the meaning of the word *flies*. A well-trained *Attention Filter* would thus probably have higher values at the coordinate crossing for the word *flies* (2nd row), *time* (1st column) and *arrow* (5th column).

The next step in the calculation process during training and inference is the most important and the main reason why Transformers can embedd context: The multiplication of the *Attention Filter* with the *Value* matrix.



Attention Filter	time	flies	like	an	arrow
time	0.43	0.26	0.09	0.02	0.20
flies	0.28	0.49	0.04	0.01	0.18
like	0.13	0.09	0.59	0.03	0.16
an	0.11	0.06	0.17	0.61	0.05
arrow	0.09	0.38	0.08	0.01	0.44

Value					
time	0.89	0.38	...	0.69	0.16
flies	0.04	0.57	...	0.22	0.93
like	0.11	0.02	...	0.41	0.93
an	0.61	0.23	...	0.03	0.75
arrow	0.81	0.19	...	0.23	0.54

Filtered Value					
time	0.07	0.30	...	1.87	-0.92
flies	-1.77	-0.09	...	-2.08	0.26
like	0.60	-1.94	...	0.64	-0.16
an	0.17	0.54	...	1.39	0.23
arrow	2.05	0.78	...	0.88	0.04

Figure 3.15.: *Attention Filter* · *Value* = *Filtered Value*

The *Value* matrix is assumed to contain encoded information about the syntax and semantics of the input text though still static at this stage. The *Attention Filter* multiplies the *Value* matrix carrying over those values to the *Filtered Value* matrix that are important to understand the context of the words. As the *Attention Filter* is different for every sentence depending on its context, the resulting *Filtered Value* matrix contains the context-dependent embeddings.

To stress this important point, an example will be given that focuses on just one coordinate of the *Filtered Value* matrix: the 2nd row and the 1st column as shown in Fig.3.16.

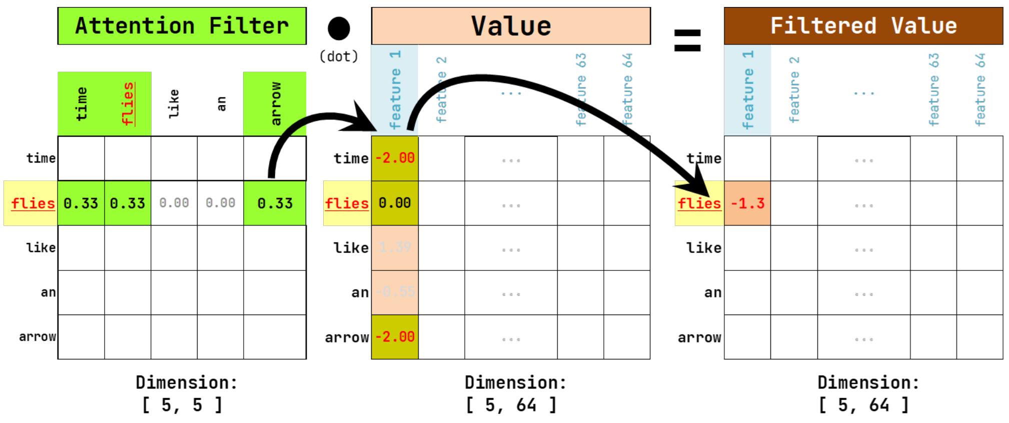


Figure 3.16.: *Filtered Value*: *time flies like an arrow*

time flies like an arrow It is assumed that the *Attention Filter* has identified the words *time* and *arrow* to be important context words for the word *flies* in the sentence *time flies like an arrow.* The *Attention Filter* thus shows exemplary values of **0.33** for each of these context words and the word *flies* itself, and a value of **0.00** for the remaining words *like* and *an.*

It is further assumed that the first column of the *Value* matrix represents a semantically interpretable feature that humans would describe as *food-like* (like the column *food* in the handcrafted feature table in Fig.3.5). If a given word in a wider sense has something to do with *food*, it is assumed that the values in the first column of the *Value* matrix are positive, else zero or negative. In Fig.3.16, the exemplary *food* feature values for the word *time* and *arrow* are **-2.00** as both words are assumed to have nothing to do with *food*. The exemplary *food* feature value for the word *flies* itself is assumed to be **0.00** (as some *flies* might be edible insects and to make the point clearer). The dot product of the 2nd row of the *Attention Filter* with the 1st column of the *Value* matrix so yields a value of **-1.3** at the 2nd row and 1st column of the *Filtered Value* matrix (see Fig.3.16).

The 2nd row of the *Filtered Value* matrix already represents the contextualized or dynamic word embedding vector of the word *flies* in the sentence *time flies like an arrow*. Whereas the *food* feature value in the *Value* matrix for the word *flies* was **0.00**, this value at the same coordinate in the *Filtered Value* matrix has changed to **-1.3**. This change in the vector representation of the word *flies* can be entirely attributed to the negative *food* feature value contributions coming from the words *time* and *arrow* in the *Value* matrix.

fruit flies like a banana The same analysis is now done for the sentence *fruit flies like a banana* shown in Fig.3.17.

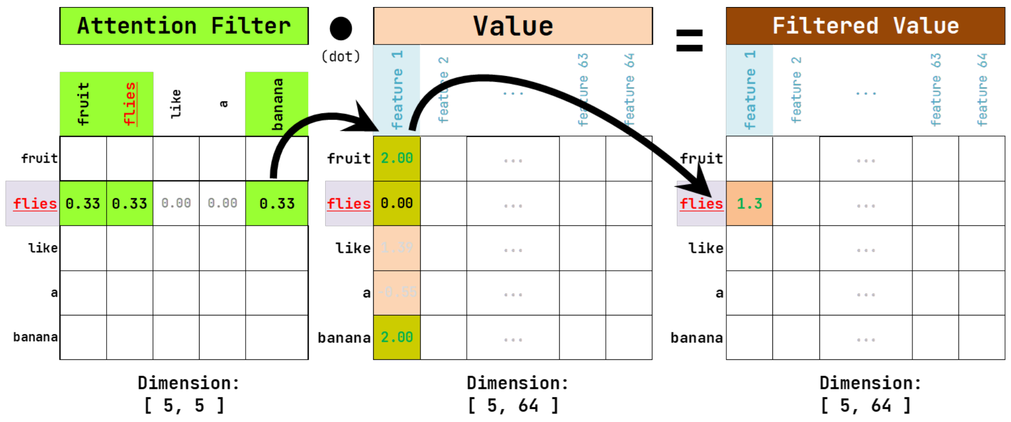


Figure 3.17.: *Filtered Value*: *fruit flies like a banana*

Here, the assumption is that the *Attention Filter* has identified the words *fruit* and *banana* to be important context words for the word *flies*, assigning the same exemplary values of **0.33** to them, like before. The exemplary *food* feature values for the word *fruit* and *banana* are now assumed to be **+2.00** as both words are strongly related to the idea of *food*. The exemplary *food* feature value for the word *flies* itself has not changed as it is indirectly coming from the static *Input Embeddings*. Every unique word (representation) in the *Value* matrix is the same for all sentences. The dot product of the 2nd row of the *Attention Filter* with the 1st column of the *Value* matrix **now** yields a positive value of **+1.3** (see Fig.3.17), again coming from a value of **0.00** in the *Value* matrix. This change in the vector representation of the word *flies* in the *Filtered Value* matrix can be entirely attributed to the positive *food* feature value contributions coming from the words *fruit* and *banana* in the *Value* matrix.

Different embeddings for same word depending on context The application of this principle to not only one column (here the *food* feature column), but to

all 64 columns of the *Value* and *Filtered Value* matrices, ensures that different semantic and syntactic aspects of word inter-dependencies are accounted for. The *Filtered Value* matrix is where all the attention magic plays out. Whereas the *Value* matrix can be considered a **static** word embedding, the *Filtered Value* matrix truly is a **dynamic** representation of words as it depends on the context words that are identified by the *Attention Filter* and that are different for every sentence. The embedding vector of the word *flies* in the *Filtered Value* matrix is different for the two sentences because the adjacent words to *flies* are different and so is its context.
This means that Transformer-based models can encode human language semantics and syntax much better than previous models.

3.5. Is a Generative LLM all you need?

Generative LLMs A recent trend in NLP seems to be the increasing use of Generative LLMs, like OpenAI's GPT models, for all kinds of NLP-related tasks such as NER or Coreference Resolution. Instead of developing, pretraining or fine-tuning custom models using traditional programming languages, algorithms and datasets, researchers and practitioners seem to increasingly focus on engineering Prompts to let Generative LLMs perform the task.

RAG Systems RAG systems are sometimes added to such Generative LLMs in order to improve their accuracy. RAG stands for *Retrieval Augmented Generation* and is a means to enrich a Prompt with relevant information. Proprietary documents are typically chunked and loaded into a vector database to add context and information to the Prompt. RAG systems typically rely on vector similarity measures such as cosine similarity (see Section 3.3.3). The text of a user question is converted to an embedding vector for which a similar embedding vector is searched in the vector database. The text document with the highest cosine similarity is processed and formulated as an answer to the user's question.

The Hallucination problem Although LLMs, with contextual word embeddings at its core, have revolutionized the NLP-world, Hallucination, or the presentation of false or misleading information as facts [76], still remains a problem and for some researchers is even inevitable [100].

Hallucinations are often caused by the conflict between the user's intent and the characteristics of LLMs:

- Good LLMs, and good Machine Learning models in general, are supposed to *generalize* well but are not supposed to *memorize* their training data

- Generative LLMs generate their text based on the highest probability for the next word
- These probabilities hinge on the data fed to the LLM during training or retrieved from RAG-systems

Generalization means that the LLM outputs text that fits learned patterns, but does not output the training (or RAG) text itself. Even if the training data was all true and factual (which is unlikely), the generated output text would most likely be different. Even if a certain, factually correct text was part of the training data, an LLM ChatBot will most likely not return the same text. Next word generation based on probabilities means, that of all candidate words the one word with the highest probability is taken, even if that probability itself is very low.

For a creative goal such as writing novels or poetry, this might not be a problem and often is a desired characteristic. For the goal of retrieving or generating text that contains correct and accurate facts, this might be damaging as demonstrated by the *ChatGPT-Lawyer*-case [58], where a lawyer cited fake legal cases that were generated by ChatGPT.

Domain-specific training data and RAG-systems can decrease Hallucination rates, but not entirely. A recent study [53] by Yale and Stanford researchers revealed that even domain-specifically trained, Generative LLMs with extensive RAG-systems attached to them, showed Hallucination-rates of between 17 and 33 percent.

Questions in regard to LLMs Two of the questions for this project are driven by this problem. The first question is:

Given the problem of hallucinations, how do Generative LLMs perform versus tradional methods on the task of information extraction?

In the *Information Extraction Pipeline*, different approaches will be studied, implemented in code and compared with each other. This covers traditional rule-based and pre-trained ANN models, but also models based on Generative LLMs.

In the Knowledge Graph part, a ChatBot (Graph Bot) will be attached to the Knowledge Graph. This is to elaborate on the question

Can Knowledge Graphs be used as an alternative to traditional RAG-systems?

A recent research by Microsoft [33] does point in this direction, but does this also apply to this project and its data?

In the next chapter, the project and *Information Extraction Pipeline* will start with extracting named entities.

4. NER - Named Entity Recognition

A named entity is an object that can be referred to and tagged with a predefined entity class label to which the object belongs. The most common entity classes and tags are PER for persons, ORG for organizations, LOC for geographical locations and GPE for geopolitical entities. The task of Named Entity Recognition (NER) is to find Spans of text that constitute such entity classes or tags [45].

4.1. Background

This section will give an overview over the models that have been used previously and the current state-of-the-art models.

4.1.1. Rule-Based Models

Rule-based models were the earliest attempts to recognize named entities in text. They often used lexical patterns, syntactic features, and domain-specific knowledge such as the following:

Rule*: If a token is capitalized and the next token is a noun, then the first token is likely a person name.*

While Machine Learning (see 4.1.2) or Deep Learning approaches (4.1.3) dominate the academic debate and research, production use cases today still use handcrafted rules and query techniques [45] to find named entities. One of the reasons for that is the ambiguity and similarity of entity names and types that confuses even the best NER models available. For instance, *A.G. Edwards* could be both, a PER or an ORG and the company *Blue Shark* could be falsely identified as an animal. Another reason is that in some use cases only certain entity types or already known entity names shall be found. In such cases, it often makes sense to comb through a text with REGEX patterns or apply other handcrafted query strategies.

4.1.2. Machine Learning Models

There are different types of Machine Learning models for the NER task.

Hidden Markov Models Hidden Markov Models or HMMs [45] are applied to sequences of observations and their respective hidden states. In the field of NER, the observations are the words in a text sequence and their hidden states are the NER-tags such as PER or LOC to be predicted.

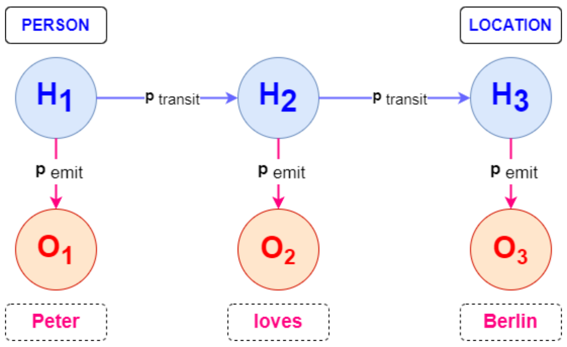


Figure 4.1.: HMM: transition/emission probabilities: $\mathbf{p}_{\text{trans}}$, $\mathbf{p}_{\text{emit}}$

Conditional Random Fields CRFs [45] belong to the family of HMMs and try to address the shortcomings of HMMs when it comes to text sequences. The first problem with a standard HMM is that transition and emission probabilities (Fig.4.1) are static whereas in text, they are dynamic. The emission probability that the name of a PER is *Peter* depends on the context and therefore is dynamic as is the probability that a LOC comes two words after a PER. The second problem is that dependencies are limited in a standard HMM meaning that previous hidden states have no direct impact on observations farther away in the sequence. In a text sequence though, the NER tag of the first word in a sentence might have an impact on a word that sits at the end of a sentence.

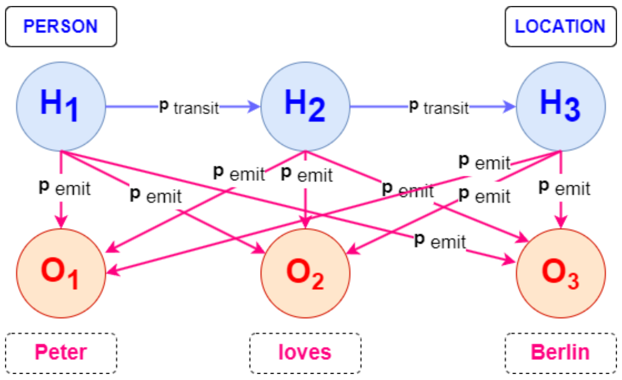


Figure 4.2.: Linear Chain CRF

A linear chain CRF (Fig.4.2) overcomes this limitation by allowing emission probabilities from a hidden state (i.e. a NER tag) to previous and subsequent observations (i.e. words) but, to avoid computational complexity, prohibits transition probabilities to previous and subsequent hidden states. The network of emission probabilities and the mapping from the words to the respective NER tag is modeled via $\boldsymbol{k}$ different feature functions $\mathbf{f}_\mathrm{j}$:

$$f(H_i) : \sum_{j=1}^{k} \vec{\mathbf{w}}_\mathbf{j} \cdot f_j(O_i, H_{i-1}, H_i, i) \tag{4.1}$$

Each feature function has different hidden-state-to-observation *connections* and the model is trained with all vectors $\boldsymbol{w_j}$ as a trainable matrix. The conditional probability of a certain state H_i (or NER-tag) given the observation O_i (or word) and a normalization factor Z is calculated by:

$$p(H_i \mid O_i) = \frac{1}{Z} \cdot \exp \sum_{l=1}^{n_{words}} \sum_{j=1}^{k} \vec{\mathbf{w}}_\mathbf{j} \cdot f_j(O_i, H_{i-1}, H_i, i) \tag{4.2}$$

Words as input to a CRF model can come in the form of an encoded index from a lookup table or as embedded word vectors [45].

4.1.3. Deep Learning Models

The advent of LLMs also changed the field of NER fundamentally. Today all leading NER-models are based on Transformer architectures [22]. Transformers, or in more general terms, ANN-based models can be distinguished between pre-trained models and Generative LLMs. Whereas pre-trained models are supervised general purpose ANNs that are fine-tuned for a specific NER task, Generative LLMs use the decoder part of a Transformer architecture to generate text in response to a NER-specific Prompt request.

Pre-Trained Models There are multiple pre-trained models available such as bert-base-NER [49] or NER-BERT [52], but the one model that stands out in terms of performance and multilingual support is GliNER [101]. GliNER is a light-weight Transformer model based on the deBERTa-v3 [39] architecture with additional layers for Span and entity representations. It was trained with texts from numerous domains and thousands of entity types [101] which allows custom entity labels beyond the ones typically found in NER models (i.e. ORG, PER, LOC, etc.). GliNER maximizes a combined Token-Span-Entity embedding matching score in a computationally efficient way to achieve an O(n log n) complexity. Because it is available as a spacy wrapper module, it can easily be added to a spacy pipeline.

spacy itself until recently also had a well-performing NER component in their pre-trained pipelines (i.e. *en_core_web_sm*, etc.) but performance-wise had to surrender to more recent and specialized models such as GliNER.

GliNER's multilingual spacy wrapper was used in the project and later compared against a traditional REGEX model.

Generative LLMs Besides using LLMs such as the ones from OpenAI directly by sending Prompt requests, there are also dedicated architectures and indirect ways to extract the named entities from text.

Wang et al [99] in 2023 introduced GPT-NER that transforms the Span labeling task into a text generation task. For instance, the task of finding LOC entities in the input text *Columbus is a city*, is transformed to generate the text sequence *@@Columbus## is a city*, where the special Tokens @@ and ## surround the entity to be extracted [99]. Before returning the result to the user, the Prompt also asks the Generative LLM to self-verify its findings thus decreasing the problem of Hallucinations. The text is then searched for the special Tokens and the model returns the entity in between them.

In the same year, Ashok and Lipton [12] introduced PromptNER, which focuses on a specific set of entity types. The Prompt provides the model with annotated examples and forces it to justify its findings with the entity type definitions provided.

Most other approaches that I came across, use Generative LLMs for NER in a similar way or concentrate on specific industries or domains.

4.2. Code-Implementation

4.2.1. Implementation of Pre-Trained Model

As GliNER [101] is available as a registered spacy plugin (see Section 2.2), the implementation of it in a spacy pipeline is straightforward. The name of the plugin is just referenced in the spacy function *enable_pipe("gliner_spacy")* within the *api_gliner()* function (see Python-Code 2)

```python
    def api_gliner(self, spacy_comp: SpacyComp = SpacyComp.GLINER):
        config = {"labels": ["organization"], "gliner_model": "urchade/gliner_multi"}
        if spacy_comp.factory_name in self.nlp.disabled:
            self.nlp.enable_pipe(name=spacy_comp.factory_name)
        else:
            if self.nlp.has_pipe(spacy_comp.custom_name):
                print('api gliner already has pipe gliner')
                self.nlp.remove_pipe(spacy_comp.custom_name)
            self.nlp.add_pipe(spacy_comp.custom_name, config=config)
        print(f'"GLINER" api initialized')
```

Python-Code 2: api_gliner()

and this function is then added to the *build_pipe()* method (see line 110 of Python-Code 3) in the *SpacyPipeBuild* class, as outlined in Section 2.2.1.

```python
 99             case SpacyTask.NER:
100                 match self.ner_method:
101                     case ExtractionType.TRADITIONAL:
102                         self.nlp.select_pipes(enable=[self.vectorizer.factory_name,
                        ↪  SpacyComp.MORPHOLOGIZER])
103                         # self.api_parser()
104                         self.func_own_regex_search()
105                         # self.api_entity_ruler()
106                         self.func_comp_name_token_regex_match()
107                         # self.func_check_spacy_ent_with_fuzzy_match()
108                     case ExtractionType.PRETRAINED:
109                         self.nlp.select_pipes(enable=[self.vectorizer.factory_name])
110                         self.api_gliner()
111                     case ExtractionType.GENERATIVE_LLM:
112                         pass
113                 self.func_sentencizer()  # Must be in for correct sentence splitting
```

Python-Code 3: build_pipe()

4.2.2. Implementation of Rule-Base Model

REGEX As stated previously, in some use cases it might make sense to use rule-based models (see 4.1.1) for NER. Pre-trained LLMs, that incorporate syntactical and semantic information in their embeddings, are required if entity names are unknown and must be determined in a probabilistic way. As the company names are provided in advance in this project, rule-based methods such as REGEXs can be tried for NER.

For REGEX to find desired company names in a text, REGEX patterns need to be created first.

REGEX patterns can consist of multiple components and these components can either be mandatory or optional. One such example is the following:

(Hello)\s?(World)\*

The first capturing group contains the word *Hello* and the second the word *World*. The second group and the space in between the words are optional as these terms are followed by an asterix or a question mark which are *quantifiers*. The first group is mandatory as an *optional quantifier*, such as ?, *, {0, n}, is missing. In this example, the expression *Hello* or *Hello World* would match the pattern, but the expressions *World* or *Hi World* would not.

Company names can consist of one or multiple words. To distinguish company names from each other and from non-company names, the company name must be recognizable and distinct. If a company name consists of more than one word, the central question for REGEX patterns is which words in the company name must be mandatory and which can be optional. Let's consider the following company name example:

Deutsche Telekom AG

A REGEX pattern must include at least the first two words combined as the words *Deutsche* and *Telekom* alone are not distinct and are commonly used words. Let's consider another example:

Apple Inc.

Here, the same logic applies: The pattern must include the legal term *Inc.* as otherwise the pattern could find a fruit instead of the company *Apple* whereas in the next example the legal term could be optional:

2CRSI S.A.

The term *2CRSI* itself is so significant and distinct, that the probability of the same term referring to something else than the given company, is very low.

The optionality of a word in a company name is important for two reasons:

- Company names are typically fully mentioned at the beginning of a text but in later sentences might be referred to with just one part of their name.

- Company mentions could also be part of a noun term.

Let's consider the following sentences:

ACCENTRO Real Estate AG today published their better than expected earnings. Accentro had a good first quarter, the Accentro-stock climbed 3% on the stock exchange.

A REGEX pattern that has no optionality for the words *Real Estate AG* would only match the first company mention, but would not match the second and third. So a good REGEX pattern might look like this:

(?:\bACCENTRO|Accentro\b)\s*(?i:\bReal\b)?\s*(?i:\bEstate\b)?\s*(?i:\bAG\b)?

The central word *Accentro* could be upper or title cased, the words *Real*, *Estate*, *AG* are optional and their case could be any. The $\backslash b$ means beginning and end of a word and the spaces in between the words are optional. With such a pattern, all three mentions of the company name in the above example will be matched.

Implementation Details As the number of companies to be searched for exceeds 2500, manually creating such regular expression patterns would be very time-consuming. The function

create_and_save_entity_patterns()

in the module *spacy_input* in the *B_spacy_pipeline* folder is particularly dedicated to algorithmically transform company names to REGEX patterns.
They work as follows:

1. The function splits the company name into a list of words.
2. Each word in this list is classified into one of the following classes:

   Binding: A conjunction term that links two other words such as *and*, *+*, -, *&* like in *Smith & Wesson*, *Busch+Lombard AG* or *Basic-Fit N.V.*, etc.

   Person Name Initials such *A.G.* in *A.G. Edwards.*

   Person Names: For that, a list of 5000 German and English common first and last names are searched for.

   Legal Terms such as *AG*, *NV*, *GmbH*, etc for which a legal term list was compiled.

   Industry Hints: Words that give a hint to the industry the company operates in such as **buildings**, **capital**, **carbon**, **care**, **casino**, **catering**, etc.

Number Terms such as 11880 in *11880 Solutions.*

Significantly Cased Words such as *SUESS* or *MicroTec*

Number and Letter Words such as *4imprint* in *4imprint Group plc.*.

Articles such as *The* in *The New York Times.*

Common Words: If the word is commonly used according to a list of *5000 most common words* in English and German or if the word is in a list of common company name prefixes and suffixes such as *Global* or *Group.*

Unknown Words: Words that do not belong to one of the other classes above.

3. Once all words of a company name are classified, combination patterns of up to five words are created.

The classification task often is carried out with the help of regular expression patterns itself and Python string functions.

As a three-word example, let's assume the company name is:

4imprint Group plc.

In the first step, the company name gets split into: [*4imprint*, *Group*, *plc*]. In the second step, each of the three words is classified: *4imprint* is classified as a *Number and Letter word*, *Group* is classified as a *Common Word* and *plc* is classified as *Legal Term.*
Number and Letter Words are considered unique and distinct so all other words in that company name can be optional. The resulting regular expression pattern is:

```
(?:\b4imprint|4Imprint\b)\s*(?i:\bGroup\b)?\s*(?i:\bplc\b)?
```

Let's consider a four-word example:

Advanced Bitcoin Technologies AG

In the first step, the company name again gets split into: [*Advanced*, *Bitcoin*, *Technologies*, *AG*] In the second step, *Advanced* is classified as a *Common Word* as is the word *Technologies. Bitcoin* is classified as an *Industry Hint* and *AG* as a *Legal Term.* No word from these classes is considered unique and distinct, but the legal classifier is not needed as there is more than one word in the company name. So the algorithm determines that the first three words are mandatory in the REGEX pattern and the legal term is optional:

```
(?:\bAdvanced\b)\s*(?:\bBitcoin\b)\s*(?:\bTechnologies\b)\s*(?i:\bAG\b)?
```

Pattern Naming and Multithreading The patterns are saved as *JSONL*-files and are later loaded by a spacy pipeline component that tries to find matches for the patterns in the text. As the number of companies and thus the number of patterns exceeds 2500, two questions arise:

1. How can matches of patterns be mapped to the individual company identifiers and names?
2. How can over 2500 patterns be efficiently run against each text?

Each REGEX pattern gets a name that is derived from the company's unique identifier which is the company stock ticker symbol.

The function *symbol_to_groupname_convert()*

```python
    @staticmethod
    def symbol_to_groupname_convert(symbol: str, named_group_prefix: str = 'SYMB_',
                                    do_reverse: bool = False) -> str:
        CONVERTER_MAP = {".": "_DOT_", "-": "_DASH_"}
        REVERSE_CONVERTER_MAP = {v: k for k, v in CONVERTER_MAP.items()}
        if do_reverse:
            if not symbol.startswith(named_group_prefix):
                raise ValueError(f"Named group to be converted must start "
                                 f"with symbol: {named_group_prefix}")
            symbol = symbol[len(named_group_prefix):]
            conversion_dict = REVERSE_CONVERTER_MAP
        else:
            if symbol.startswith(named_group_prefix):
                raise ValueError(f"Named group already converted as "
                                 f"it starts with symbol: {named_group_prefix}")
            symbol = named_group_prefix + symbol
```

Python-Code 4: symbol_to_groupname_convert()

first converts the company symbol to a regular expression name which is then prefixed to the pattern that was created above.

For instance: The stock ticker symbol for the company *Advanced Bitcoin Technologies AG* is *ABT.DU*. The function converts this to a pattern-eligible name which is *SYMB_ABT_DOT_DU* and with it prefixes the regular expression pattern that was created above:

(?P<SYMB_ABT_DOT_DU>(?:
(?:\bAdvanced\b)\s*(?:\bBitcoin\b)\s*(?:\bTechnologies\b)\s*(?i:\bAG\b)?))

A match against this pattern returns a Python *re match object* that, among other information, contains the pattern's name. The pattern's name then can be re-converted to the company symbol which is attached as extension to the matching Span or Token in the Doc-object of the spacy pipeline.

To make this process computationally efficient, the matching algorithm is run concurrently. The function *run_re_finditer_concurrently()* runs the Python re function *finditer* in parallel and the spacy pipeline component *OwnRegexSearch* applies this concurrent function to each article text:

```python
def run_re_finditer_concurrently(pattern_list: list, text: str) -> Generator:
    """ The function here must have exactly one parameter. """
    with (concurrent.futures.ThreadPoolExecutor(max_workers=len(pattern_list))
          as executor):
        future_to_result = [executor.submit(re.finditer, pattern, text)
            for pattern in pattern_list]
        futures_done = concurrent.futures.as_completed(fs=future_to_result,
            timeout=None)
        for future in futures_done:
            try:
                data = list(future.result())
            except (Exception, TimeoutError):
                print(f'Fetching concurrent.future failed for future: {future}')
```

Python-Code 5: run_re_finditer_concurrently()

This match algorithm, that runs over 2500 patterns on each article text, on my standard machine only takes around 500 milliseconds on average per text to execute.

Comparing Rule-Based vs. Pre-Trained vs. Generative LLMs With 500 milliseconds per text, the rule-based method with REGEX is approximately as fast as the respective GliNER component in spacy's pre-trained pipeline. The REGEX algorithm does not find all companies in the text and is particularly sensitive to misspelled company names, too little optionality in the REGEX pattern or if company names only consist of one, very common word.

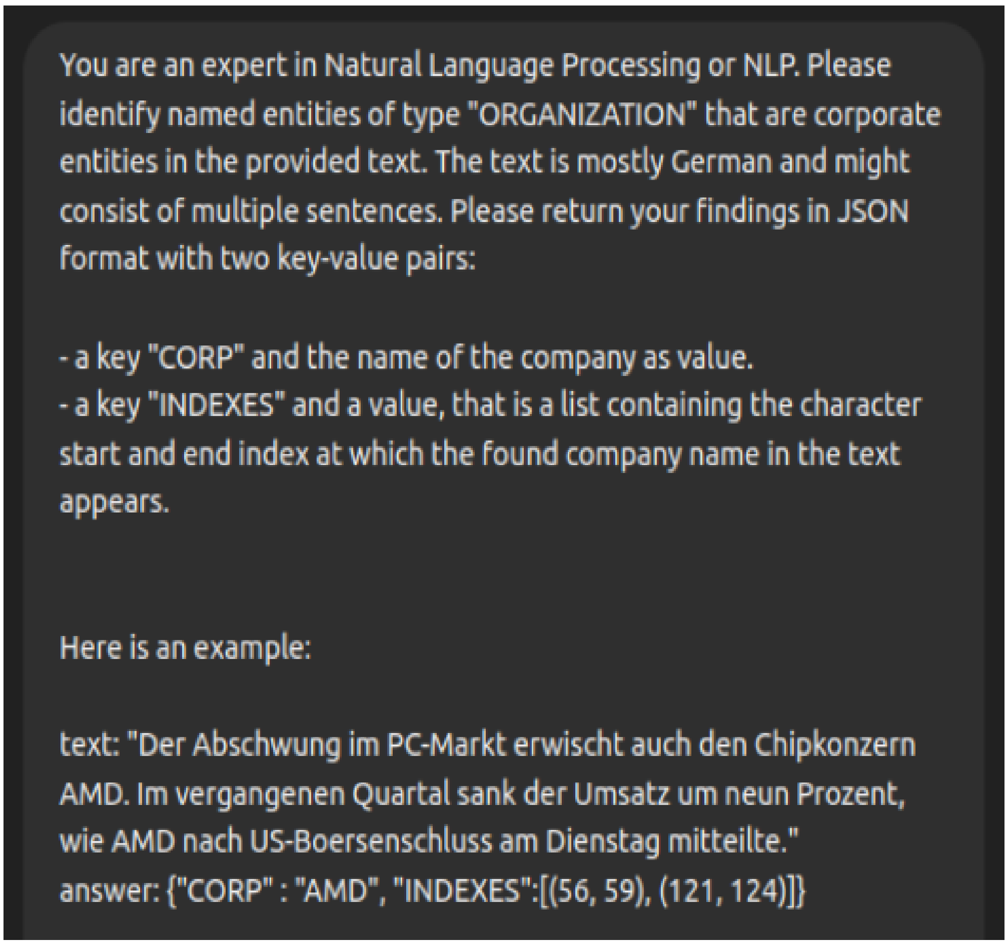
But in most cases, it is as accurate as spacy's pre-trained GliNER pipeline component and in some cases, particularly if the company name is peculiar and can be confused with subjects from other domains, it is more accurate. Some shortcomings of the regex approach can be cured if non-working patterns are manually adjusted. For instance: The company name

Schott Pharma AG & Co. KgaA

contains the person name *Schott* which is uncommon and thus not found in the existing list of person names for the classification task in the *create_and_save_entity_patterns()* function. The name *Schott* also starts with a capital letter and is thus considered unique and distinct for which the algorithm allows the other words in the company name to be optional. The REGEX pattern could be adjusted by manually adding the name *Schott* to the list of common person names or by requiring the word *Pharma* to be mandatory.

4.2.3. Implementation of Generative LLM model

For the sake of comparing different approaches, I also tried a Generative LLM for the NER task. Asked to find named entities in a text, ChatGPT using the OpenAI 4o-mini model with a simple Prompt, already found most of the entities in a sample text:



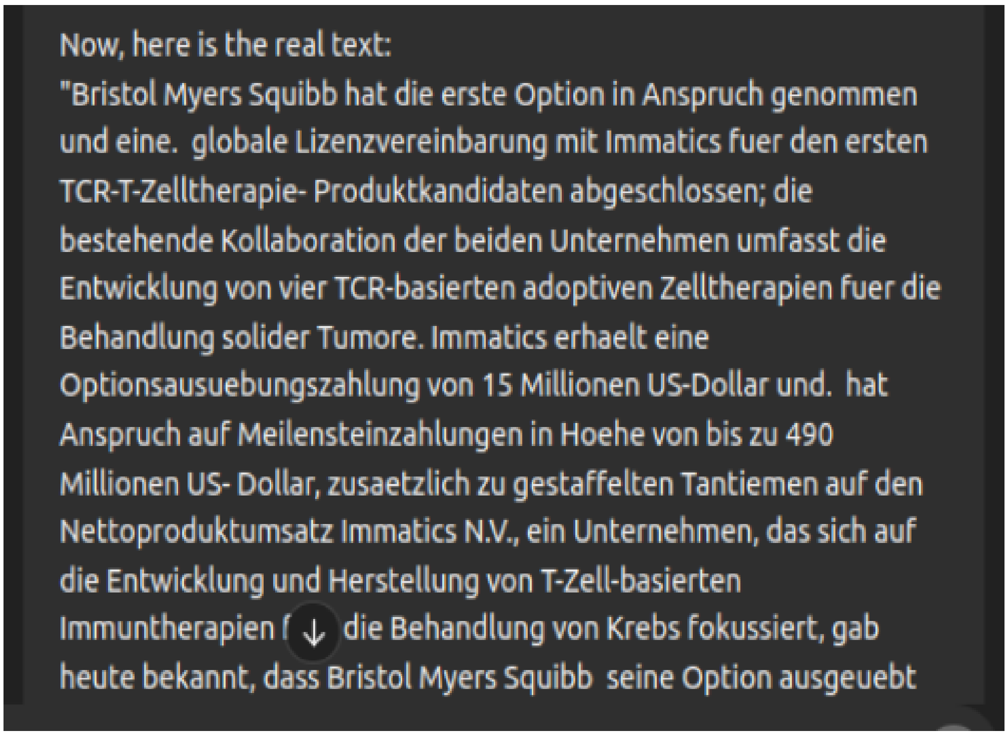
You are an expert in Natural Language Processing or NLP. Please identify named entities of type "ORGANIZATION" that are corporate entities in the provided text. The text is mostly German and might consist of multiple sentences. Please return your findings in JSON format with two key-value pairs:

- a key "CORP" and the name of the company as value.
- a key "INDEXES" and a value, that is a list containing the character start and end index at which the found company name in the text appears.

Here is an example:

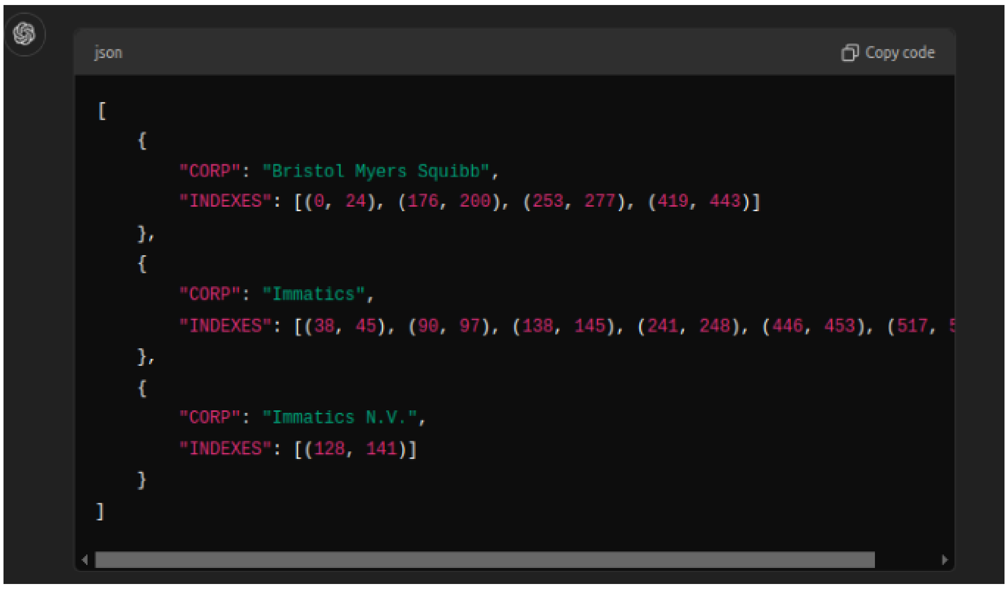
text: "Der Abschwung im PC-Markt erwischt auch den Chipkonzern AMD. Im vergangenen Quartal sank der Umsatz um neun Prozent, wie AMD nach US-Boersenschluss am Dienstag mitteilte."
answer: {"CORP" : "AMD", "INDEXES":[(56, 59), (121, 124)]}

Figure 4.3.: ChatGPT NER Prompt - Part 1



Now, here is the real text:
"Bristol Myers Squibb hat die erste Option in Anspruch genommen und eine. globale Lizenzvereinbarung mit Immatics fuer den ersten TCR-T-Zelltherapie- Produktkandidaten abgeschlossen; die bestehende Kollaboration der beiden Unternehmen umfasst die Entwicklung von vier TCR-basierten adoptiven Zelltherapien fuer die Behandlung solider Tumore. Immatics erhaelt eine Optionsausuebungszahlung von 15 Millionen US-Dollar und. hat Anspruch auf Meilensteinzahlungen in Hoehe von bis zu 490 Millionen US- Dollar, zusaetzlich zu gestaffelten Tantiemen auf den Nettoproduktumsatz Immatics N.V., ein Unternehmen, das sich auf die Entwicklung und Herstellung von T-Zell-basierten Immuntherapien f die Behandlung von Krebs fokussiert, gab heute bekannt, dass Bristol Myers Squibb seine Option ausgeuebt

Figure 4.4.: ChatGPT NER Prompt - Part 2



```json
[
    {
        "CORP": "Bristol Myers Squibb",
        "INDEXES": [(0, 24), (176, 200), (253, 277), (419, 443)]
    },
    {
        "CORP": "Immatics",
        "INDEXES": [(38, 45), (90, 97), (138, 145), (241, 248), (446, 453), (517, 5
    },
    {
        "CORP": "Immatics N.V.",
        "INDEXES": [(128, 141)]
    }
]
```

Figure 4.5.: ChatGPT JSON Response

But this Generative LLM-approach for NER was inferior to the rule-based

REGEX approach because:

- It found less entities.
- It found entities that were not searched for.
- The latency was higher as the request had to be sent to the OpenAI server first.
- The position indexes for the found entities were all wrong so locating them would have required an extra step or the usage of OpenAI function tools which would have increased the latency even further.
- It was more costly as OpenAI charges fees for using their LLMs.
- The results were very volatile and unstable as they differed from request to request.

Some of these shortcomings can probably be cured by:

- fine-tuning the Prompt.
- providing more few-shot examples.
- using function tools that are available from most LLM-providers such as OpenAI.
- providing a list of entities that shall be searched for.
- using local, open-source LLMs and frameworks such as Llama3 [4] running on Ollama [6] installed on a local machine.

Nevertheless, the rule-based approach with REGEX already delivered very good results and in my assessment is best suited if the company names are given. This would change if they were not given as then the creation of REGEX patterns was not possible and a choice would need to be made between a pre-trained and fine-tuned model or a Generative LLM-approach.

4.2.4. Information Extraction Pipeline

After the NER component has run, the spacy pipeline has attached the found company information to the respective custom extensions:

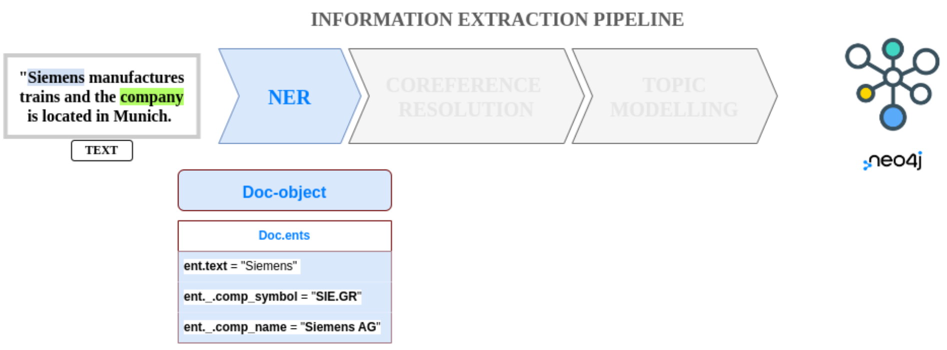

INFORMATION EXTRACTION PIPELINE
"Siemens manufactures trains and the company is located in Munich.
TEXT
NER
COREFERENCE RESOLUTION
TOPIC MODELLING
neo4j
Doc-object
Doc.ents
ent.text = "Siemens"
ent._.comp_symbol = "SIE.GR"
ent._.comp_name = "Siemens AG"

Figure 4.6.: spacy pipeline after NER component

5. Coreference Resolution

5.1. Background

In linguistics, Coreference occurs when two or more expressions refer to the same person or thing [73]. In the sample sentence of Fig.5.1, the expression *he* refers to a PER with the name *Philip* and the word *it* refers to the musical instrument *bass*, a thing.

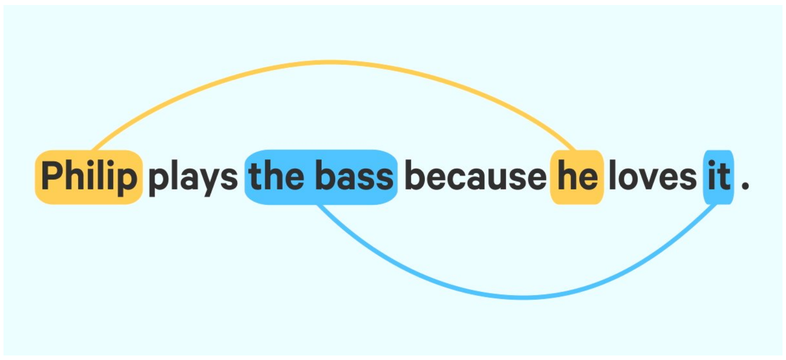


Figure 5.1.: Coreferences. Image from: [34]

Coreference Resolution is considered one of the most challenging tasks in NLP and until recently was deemed an unsolved problem [62]. The fundamental problem is the complex and ambiguous nature of natural language text, the requirement for a deep language understanding and the use of background knowledge in detecting Coreferences [62].

5.1.1. Definitions

In the sample sentence of Fig.5.1, there are four Mentions of which three are Tokens [*Philip, he, it*] and one is a Span [*the bass*] consisting of more than one Token. Mentions often consist of long Spans such as *The New York Times* or *Bayerische Motoren Werke AG*. Finding Coreferences in a text thus often is considered a Span-based task. The Mention *Philip* is the Antecedent for the Mention *it*. As the name *Philip* comes before *it*, *it* is the Anaphora for *Philip*. If *Philip* came after *it*, one would usually refer to *it* as Cataphora. The sample sentence has two Coreference Clusters or Coreference chains: [*Philip, he*], [*the bass, it*].

5.1.2. Methods

Mention Detection The first step to detect Coreferences is Mention detection [44]. Earlier methods used grammatical parsers and named entity taggers on the text and, based on that, extracted Spans that meet certain criteria as Mention-candidates. More recent Mention detection systems go even further and extract literally all N-Gram Spans of Tokens or words up to N=10, regardless of their grammatical attributes [44]. Due to the computational complexity of this approach of

$$O(number\_of\_tokens^4) \tag{5.1}$$

for N=2 (Bi-Grams) [47], there is a need to filter out unlikely Spans.

Rule Based Methods Earlier systems used rules to filter out non-coreferential pronouns like *it* in sentences such as *It is likely that ...*. These rule-based systems relied on regular expressions, dictionaries of key-verbs/-adjectives, POS and NER tags and other grammatical or syntactical rules. Rule-based systems, however, generally underperform more modern systems that incorporate a learning process [44].

Feature Based Methods The next step thus were standard Machine Learning models that used decision trees, support vector machines and binary classifiers [50]. Theoretically, such models are capable to not only identify potential Mentions but also whether such Mentions are indeed Coreferences. One common approach of such Classifiers is to implement a *Mention-Pair-Architecture* that predicts if a given Span pair of an Anaphora and an Antecedent are Coreferences or not [44]. Another Classifier architecture type is the *Mention-Rank-Architecture* that chooses the highest-scoring Antecedent for each Anaphora. *Entity-based architectures* enhanced their feature set by adding Mention-distances, syntactic, symantic, rule-based and lexical attributes to predict if a Token belongs to a certain Coreference Cluster [14].

Nevertheless, correctly detecting co-referential Mentions remained difficult, mainly due to the model's lack for a deep language understanding [44].

Neural Network Based Methods A major breakthrough came with the advent of LLMs as contextual embeddings allowed to semantically compare Mentions. Consider the following phrases:

1: **Make a payment! You can make [it] in advance.** [anaphoric]

2: **Go west! You can make [it] in Hollywood.** [non-anaphoric]

Idea based on: [44]

Whereas the *it* in the second phrase is non-anaphoric and part of the idiom *make it*, the *it* in the first phrase is anaphoric and refers to a *payment*. With contextual word-embeddings, the vectors for *payment* and *it* ideally should be similar in the first sentence whereas the vector for *it* in the second sentence should significantly differ to any other word vector in that phrase. The capability to detect the Mentions *payment* and *it* as Coreferences should thus increase with contextual word embeddings.

5.2. Models

Coreference models can also be distinguished into three classes: Rule-Based, Pre-Trained and Generative LLM models. As rule-based models factually are non-existent nowadays, the focus in the following chapter will be on Pre-Trained and Generative LLM models.

5.2.1. Pre-Trained Models

Research All descriptions of the currently most performant pre-trained Coreference models are laid out in publicly available research papers. Unfortunately, not all models are implemented in code and easily available as Python package. This includes the two currently most performing [64] pre-trained Coreference models by Google Research [13] and Vladimir Dobrovolskii [32]. As the model training of unimplemented Coreference research papers would go beyond the scope of this thesis, I will shortly describe only those research papers for which a pre-trained Python package exists.

e2e: End-to-end Neural Coreference Resolution In 2017, Lee et al.[47] proposed an LSTM [40] model with a *Mention-Rank-Architecture* where parsers and taggers are not required. The model consists of two parts as shown in Fig.5.2 and 5.3:

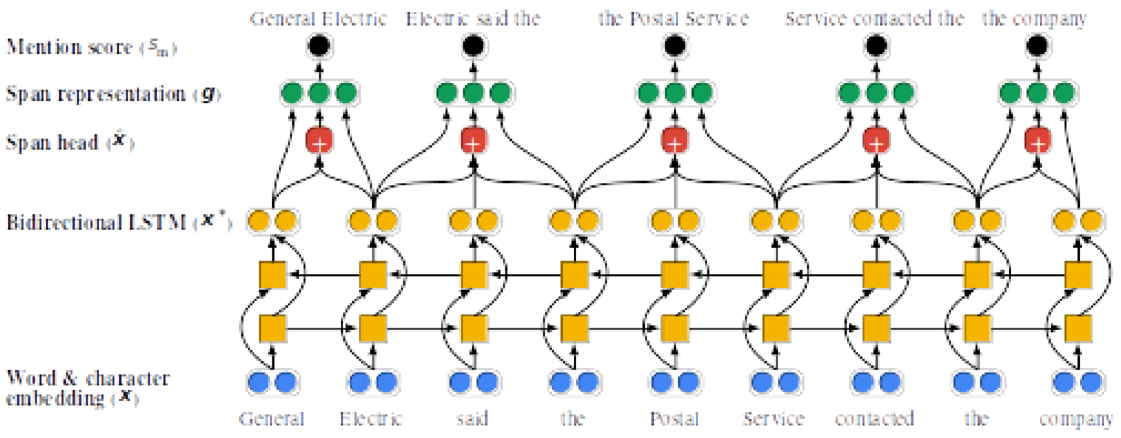


Figure 5.2.: End-to-end Neural Coreference Resolution: Part 1

In the first part (Fig.5.2), the LSTM calculates Span-embeddings and assigns corresponding Mention scores. e2e is a supervised model trained with hand-labeled data. Only the top M-ranked Spans with the highest Mention scores are considered potential Mentions and are further processed. The concatenation of the Token embedding vectors creates the contextual Span-representation vectors.

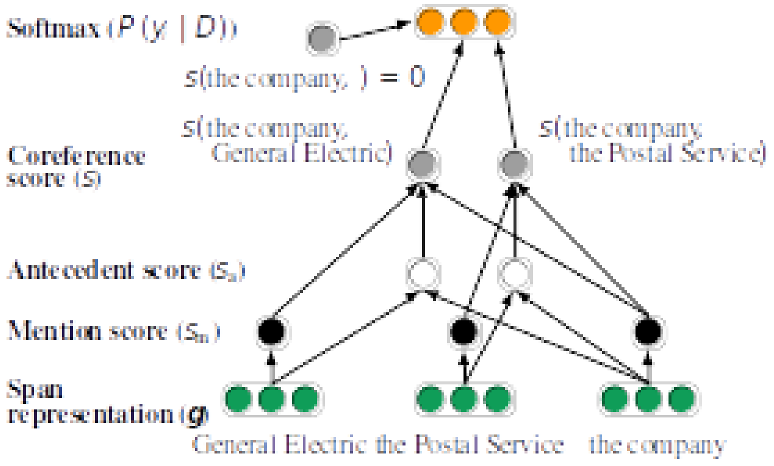


Figure 5.3.: End-to-end Neural Coreference Resolution: Part 2

In the second part of the model in Fig.5.3, *Antecedent scores* are computed from pairs of Span-representations. The final Coreference *score* is computed by summing the two Mention *scores* of the Spans and their pairwise *Antecedent score*. This calculation method avoids the computational inefficient evaluation of all (non-filtered-out) Span pairs and ensures, that the Span-pairs with the highest similarity yield the highest Coreference *score*.

c2f: Higher-order Coreference Resolution with Coarse-to-fine Inference In 2018, Lee et al. further improved their own model by changing the inference procedure of the *Mention-Rank-Architecture*[48] to refine Span representations. This improved both, the computational complexity and the performance of the model.

BERT for Coreference Resolution In 2019, Joshi et al. [43] reused the *Mention-Rank-Architecture* of Lee et al.[47] but substituted the LSTM-embeddings with BERT-embeddings [31] and later with SpanBERT-embeddings [42]. SpanBERT was designed to better represent Spans of text and is a pre-training method that extends the BERT-model by masking contiguous random Spans, rather than random Tokens [42]. SpanBERT is applicable to a wide range of tasks such as question answering, relation extraction and Coreference Resolution. For their Coreference-model, they applied the SpanBERT-method on a Coreference task and a Coreference training set [42]. The model achieves an F1-score of 77.7 for the SpanBERT-base and 79.6 for the SpanBERT-large [43], currently ranking among the five best Coreference Resolution models [64].

s2e: Coreference Resolution without Span Representations In 2021, Kirstain et al.[46] came up with a different architecture that focused on Tokens instead of Spans.

$$f_m(q) = \mathbf{v}_s \cdot \mathbf{m}^s_{q_s} + \mathbf{v}_e \cdot \mathbf{m}^e_{q_e} + \mathbf{m}^s_{q_s} \cdot \mathbf{B}_m \cdot \mathbf{m}^e_{q_e}$$

Figure 5.4.: Bilinear function: start/end Mention embeddings: $\mathbf{m}^{\mathrm{s}}$, $\mathbf{m}^{\mathrm{e}}$

The model computes Mention scores as bi-affine products over the start and end Token representations ($\mathbf{m}^{\mathrm{s}}$, $\mathbf{m}^{\mathrm{e}}$) with $\mathbf{v}_{\mathrm{s}}$, $\mathbf{v}_{\mathrm{e}}$ and $\mathbf{B}_{\mathrm{m}}$ as trainable matrices (Fig.5.4).

$$\begin{aligned} f_a(c,q) = \mathbf{a}^s_{c_s} \cdot \mathbf{B}^{ss}_a \cdot \mathbf{a}^s_{q_s} + \mathbf{a}^s_{c_s} \cdot \mathbf{B}^{se}_a \cdot \mathbf{a}^e_{q_e} \\ + \mathbf{a}^e_{c_e} \cdot \mathbf{B}^{es}_a \cdot \mathbf{a}^s_{q_s} + \mathbf{a}^e_{c_e} \cdot \mathbf{B}^{ee}_a \cdot \mathbf{a}^e_{q_e} \end{aligned}$$

Figure 5.5.: Bilinear function: start/end Antecedent embeddings: $\mathbf{a}^{\mathrm{s}}$, $\mathbf{a}^{\mathrm{e}}$

Similarly, it also extracts start and end Token representations ($\mathbf{a}^{\mathrm{s}}$, $\mathbf{a}^{\mathrm{e}}$) for the antecedent scoring function with $\mathbf{B}_{\mathrm{m}}$ as another trainable matrix (Fig.5.5). This calculation is equivalent to computing a bilinear transformation between the concatenation of each Span's boundary Tokens' representations, but bypasses the need to create n^2 explicit Span representations (for Bi-Gram Span) und thus reduces complexity [46]. The model is the fastest of all known Coreference models and, with an F1-score of 80.4, currently ranks fourth [64] in regard to prediction performance.

Available Models Based on these research papers, the following Python modules are available:

AllenNLP AllenNLP [35] is an AI platform supported by the Allen Institute for Artificial Intelligence [1]. AllenNLP offers multiple pre-trained models and a pip-installable Python module [35]. Among the models offered is a pre-trained Coreference Resolution model that is based on the c2f-model by Lee et al.[48]. The GloVe embeddings in the original paper [48] have been substituted with SpanBERT embeddings [2]. Unfortunately, since 2022 the AllenNLP ecosystem is in maintenance mode. Dependencies to more recent Python versions are not upgraded and the most recent training for the *coref-spanbert-large*-model goes back to March 2021. Additionally, AllenNLP only offers pre-trained models for English but not for German.

F-COREF F-COREF is a Python implementation and an adopted version of the s2e-model by Kirstain et al. [46]. F-COREF predicts Coreference Clusters 29 times faster than the AllenNLP model and requires only 15% of its GPU memory use, with only a small drop in performance (78.5 vs 79.6 average F1) [60]. Unfortunately, F-COREF currently only has a pre-trained model for English but not for German.

Coreferee Coreferee [41] is a pip-installable Python package that uses a mixture of ANNs and programmed language-specific rules. It focuses heavily on detecting Anaphoras and noun phrases and thus depends on the grammatical and syntactical capabilities of spacy [34]. The likelihood scores for anaphoric pairs are calculated by utilizing the contextualized vectors of the underlying and pre-trained spacy models. Specific language packages other than English exist for German, Polish and French and must be downloaded individually.

Crosslingual-Coreference The Crosslingual-Coreference model [21] builds on the AllenNLP-Coreference model [2] but modifies its *coref_resolved*-method. It investigates all the Coreference Clusters found by the AllenNLP-Coreference model, but only considers Coreference Clusters that contain a noun phrase [5]. To determine if a Coreference Cluster contains a noun phrase, it uses the POS-tagger of pre-trained models from spacy [34] that are also available in languages other than English. If the model parameter is set to *minilm*, the LLM used in resolving Coreference Clusters is multilingual and pre-trained by Microsoft [3]. As such, this model can be applied multilingual, i.e. to English and German.

Evaluated Pre-Trained Models As only the Coreferee [41] and Crosslingual-Coreference model [21] are available for the German language, in the project only those two pre-trained models were evaluated against each other. I tried both extensively with multiple texts, and the Crosslingual-Coreference model [21] clearly outperformed the Coreferee [41] model. It found more Coreference mentions and had a lower entity confusion rate in texts that contained multiple entities. The Crosslingual-Coreference model [21] was later compared against a Generative LLM approach.

5.2.2. Generative LLM Models

Pre-Trained AND Generative Besides using Generative LLMs and Prompt requests directly, there are also dedicated architectures and indirect ways to extract Coreferences from text. In the previous section, the Coreference model by Google Research [13] was classified as a pre-trained model, but it could also be classified as a Generative LLM model as the boundaries between these two types are increasingly blurry.

The pre-trained Google model [13] is a sequence-to-sequence model that uses both stacks of the Transformer architecture, i.e. the encoder *and* the decoder. It so generates new text based on an input sequence where the input sequence is an untagged string and the output sequence the input string plus tags attached to it. The tags contain the desired Coreference annotations. The model is trained with the aim to annotate the input string with the respective Coreference tags.

This approach is very similar to the one Wang et al [99] use in their model for the NER task (see 4.1.3).

Whereas the Google [13] model introduces some extra layers to an encoder-decoder stack, Zhang et al [102] use a plain pre-trained encoder-decoder (text-to-text) T5 model [65] and finetune it in a supervised way with annotated examples. The source sequence contains the input string and the target sequence the *Coreference-annotated* input string. During finetuning, the classification task uses the usual source-target sequence pair and tries to minimize the cross-entropy loss given the per-Token labels [102].

Models used As stated previously, neither the Google [13] model nor the Zhang et al [102] model are (easily) available as a Python module. I therefore tested the Generative LLM approach directly as described in the next section.

5.3. Code Implementation

5.3.1. Implementation of Pre-Trained Model

As stated previously (see Section 5.2.1), the Crosslingual-Coreference model [21] is based on the deprecated AllenNLP-Coreference model [2], depends on the POS-tagger from a certain spacy version and uses a fine-tuned and pre-trained Microsoft LLM that was last updated in 2021. It has some issues for Python versions greater than 3.10 and dependency conflicts to other Python libraries in the project that could not be resolved.

I thus decided to run the Crosslingual-Coreference model [21] in an isolated environment using Docker. The files for building the Docker container can be found in the */src/D_coref/img_xx_coref_files* directory that also contains a Dockerfile and a requirements.txt file. The Docker container is accessible by using http requests.

The Crosslingual-Coreference model [21] in the Docker container is run as a spacy pipeline-component that attaches found Coreference Clusters to spacy's Doc-object custom extensions. The Coreference Clusters are nested Python lists in the format

[[[cl1_start1, cl1_end1], [cl1_start2, cl1_end2]], [[cl2_start1, cl2_end1], [cl2_start2, cl2_end2]]]

where character start and end index positions for each word in a Coreference Cluster are shown.

The function *spread_comp_ext_to_coref_cluster_spans()* in Python-Code 6

```python
    def spread_comp_ext_to_coref_cluster_spans(self, coref_clusters: list[list[list[int]]], doc: Doc):
        search_matches: list[SearchMatch] = []
        for ent in doc.ents:
            if (comp_name := getattr(ent._, SpacyExt.COMP_NAME.value)) != self.init_mark:
                comp_symbol = getattr(ent._, SpacyExt.COMP_SYMBOL.value)
                comp_start_char: int = ent.start_char
                comp_end_char: int = ent.end_char
                for cluster in coref_clusters:
                    cluster_has_overlap = self._is_valid_coreference(comp_start_char=comp_start_char, comp_end_char=comp_end_char, cluster=cluster, doc=doc)
                    if cluster_has_overlap:
                        for cluster_item in cluster:
                            cl_item_start_char = cluster_item[0]
                            cl_item_end_char = cluster_item[1]
                            cl_item_text = doc.text[cl_item_start_char:cl_item_end_char]
                            # Note: exclude conditions here
                            cl_item_is_excluded: list = [term for term in self.excluded_coreferences if term.lower() in cl_item_text.lower().split()]
                            if not cl_item_is_excluded:
                                search_match: SearchMatch = SearchMatch(comp_name=comp_name, comp_symbol=comp_symbol, text=cl_item_text, label=self.comp_label, start_idx=cl_item_start_char, end_idx=cl_item_end_char, idx_refer_to=IDXReferTo.CHARS)
                                search_matches.append(search_match)
            else:
                continue
        return search_matches
```

Python-Code 6: spread_comp_ext_to_coref_cluster_spans()

first checks if any Coreference Cluster word overlaps with a spacy custom extension Span that has a company name and company symbol attached to it. If this is the case, it creates a new instance of the SearchMatch class with the same company name and company symbol for every other member of the Coreference Cluster.

Before setting the found Coreferences to the spacy custom extensions, the function *resolve_span_conflicts_and_set_new_ents()* in Python-Code 7

```python
    def resolve_span_conflicts_and_set_new_ents(self, doc: Doc, matches: list[SearchMatch], set_in: SpacyComp, overwrite_own_ext: bool = True) -> Doc:
        if not matches:
            return doc
        ents = sorted(list(doc.ents), key=lambda span: span.start)
        matches = sorted(matches)
        for m in matches:
            # Note: Do start_idx and end_idx refer to word- or char-indexes ?:
            if m.idx_refer_to == IDXReferTo.WORDS:
                new_ent = Span(doc, m.start_idx, m.end_idx, label=m.label)
            elif m.idx_refer_to == IDXReferTo.CHARS:
                new_ent = doc.char_span(m.start_idx, m.end_idx, label=m.label, alignment_mode='expand')
            else:
                raise ValueError('idx_refer_to must be set to clarify whether start/end refers to word or char indexes.')
            # Note: Remove old ents that overlap with new ents:
            old_ents_shall_be_substituted: bool = True
            old_ents_to_be_removed: list[Span] = []
            if ents:
                for old_ent in ents:
                    if old_ent.start > new_ent.end or old_ent.end <= new_ent.start:
                        continue
                    else:
                        if getattr(old_ent._, SpacyExt.SET_IN.value) == self.init_mark:
                            # Note: Case1: old_ent is set by spacy's SpacyComp.NER.factory_name
                            old_ents_to_be_removed.append(old_ent)
                        elif getattr(old_ent._, SpacyExt.SET_IN.value) != self.init_mark and (new_ent.start <= old_ent.start and new_ent.end >= old_ent.end):
                            # Note: Case2: old_ent is set by OWN FUNCTION but IS FULLY WITHIN BORDERS of new_ent
                            if overwrite_own_ext:
                                old_ents_to_be_removed.append(old_ent)
                            else:
                                old_ents_shall_be_substituted = False
                                break
                        else:
                            # Note: Case3: All other cases such as: old_ent is set by own function but IS NOT within borders of new_ent, etc.
                            old_ents_shall_be_substituted = False
                            break
            # Note: Set new_ent ONLY IF ALL old_ents WERE NOT SET PREVIOUSLY BY OWN FUNCTION OR IF old_ent is fully WITHIN BORDERS of new_ent:
```

Python-Code 7: resolve_span_conflicts_and_set_new_ents()

checks if the respective spacy Token or Span already has these extensions set by the previous NER component of the pipeline. Only if this is not the case, the custom extension is set with the information from the SearchMatch instances.

5.3.2. Implementation of Generative LLM Model

To implement the Generative LLM model directly as reasoned in Section 5.2.2, the Python LangChain [71] library was used. LangChain is a modular LLM framework where components such as Prompts, LLMs, Agents and Tools can be chained together to build an LLM pipeline.

The latest LangChain version 0.3 uses Pydantic version 2 [70], a mandatory type checking library, that had unresolvable dependency conflicts with existing libraries in the project. For this reason, I also isolated the Generative LLM approach by using Docker. The files for building the Docker container can be found in the */src/D_coref/img_llm_extract_coref_files* directory that, among other files, contains a Dockerfile and a requirements.txt file.

Data Model In the mentioned directory, there is also a Python file named *data_models.py* (see Python-Code 8) that lays out the desired format of the LLM response and the format of messages in the Prompt. This is important because Generative LLMs usually suffer to deliver the generated text response in an appropriate format such as JSON so that it can be further processed by an algorithm.

```python
from pydantic import BaseModel, Field


class Coreference(BaseModel):
    """ Coreferences occur when one or more expressions or mentions in a text refer to a company name at another position in that text.
    For example: In the text 'Steve Jobs founded Apple. The company was very successful. Today it is a media company.' the mention 'company' in the second sentence and the mention 'it' in the third sentence are coreferences to the company name 'Apple'.
    The 'coref_text'-attribute is the substring of the found coreference within the text string.
    The 'coref_with_surrounding'-attribute is the coreference substring plus its surrounding characters to the left and right that can include up to two words on each side. """
    coref_text: Optional[str] = Field(default=None, description='The coreference substring in the text string')
    coref_with_surroundings: Optional[str] = Field(default=None, description='The coreference substring plus its characters to the left and right in the text string up to two words on each side.')


class ClusterHead(BaseModel):
    """ The cluster head is the anchor text of a coreference cluster to which coreferences refer.
    The cluster head always is the name of a company which is provided in the user message.
    The 'head_index_start'- and the 'head_index_end'-attributes are integer values that mark the start and end position of the cluster head substring within the text."""
    head_text: Optional[str] = Field(default=None, description='The string characters of the cluster head which is a company name')
    head_index_start: Optional[int] = Field(default=None, description='The position index of the first character of the cluster head substring')
    head_index_end: Optional[int] = Field(default=None, description='The position index of the last character of the cluster head substring plus one')


class Cluster(BaseModel):
    """ A coreference cluster consists of one cluster head in a text and one or more coreferences at another position in that text that co-refer to this cluster head.
        For example: In the text 'Steve Jobs founded Apple. The company was very successful. Today it is a media company.' the coreference 'company' in the second sentence and the coreference 'it' in the third sentence co-refer to the cluster head which is 'Apple'.
    """
    cluster_id: Optional[int] = Field(default=None, description='The identification number of the cluster provided by the user. Always return the same number that was provided by the user.')
    text: Optional[str] = Field(default=None, description='The text to search in')
    cluster_head: Optional[ClusterHead] = Field(default=None, description='The cluster object which is is provided in the user message')
    coreferences: Optional[list[Coreference]] = None


class DataContainer(BaseModel):
    data_list: list[Cluster] = []
```

Python-Code 8: Data Model Coref LLM Extract

Python-Code 8 shows the data model which consists of four Python classes that all inherit from Pydantic's *BaseModel*. This way, Pydantic in the background checks instances of these classes for their type (such as Integer, Float, String, etc.) and raises an error if the value of the respective instance variable does not comply with it.

Prompt The Prompt for the Generative LLM model contains examples (few-shots) in the form of instances of these Pydantic classes. Instances of the class Cluster contain the article text, an instance of the class ClusterHead and potentially multiple instances of the class Coreference.

Instances of the ClusterHead class are created with the information that was previously extracted by the pipeline's NER component: the company name (*head_text*) and the start (*head_index_start*) and end (*head_index_end*) position of this company name within the article text.

Instances of the class Coreference contain the Coreference string (*coref_text*) and the Coreference string with two words on the right and left (*coref_with_surroundings*).

As an example, let's say that the article text is

Siemens manufactures trains. It is located in Munich.

then the according few-shot example in the Prompt would be

Cluster(cluster_id=1, text="Siemens manufactures trains. It is located in Munich.", cluster_head=ClusterHead(head_text="Siemens", head_index_start=0, head_index_end=7), coreferences=[Coreference(coref_text="It", coref_with_surroundings="manufactures trains. It is located")])

The question is why the Coreference class does not include position indexes of the found Coreference word within the text similar to the *head_index_start* and *head_index_end* variables for the company name. Then the subsequent algorithm could easily locate the Coreference words in the text and attach them to the respective custom extensions of the Doc-object in the spacy pipeline.

I tried to force the Generative LLM to return those position integer values for Coreferences it found but the model, even equipped with the respective tools in the LangChain pipeline, did not succeed. I assume that this could also be the reason why Wang et al [99] had their NER model generate string annotations instead of returning text position indexes such as

@@Columbus## is a city

for the input sequence *Columbus is a city* (see Section 4.1.3).

I also could have asked the Generative LLM to annotate the Coreference it found with the same symbols such as:

Siemens manufactures trains. @@It## is located in Munich.

But the problem was that the article text, even after cleaning, already contained such symbols (i.e. @ and #) so that a subsequent algorithm could potentially have confused them.

The few-shot examples provided to the Generative LLM can be found in: *src/D_coref/img_llm_extract_coref_files/examples.py*

Response Format The Pydantic classes are not only used for few-shot examples in the Prompt, but also for defining the response format. The LangChain pipeline or chain is defined in
src/D_coref/img_llm_extract_coref_files/coref_langchain.py:

```
19  class CorefLangchain:
20      def __init__(self, prompt_template: str, model_name: str = "gpt-4o"):
21          nest_asyncio.apply()
22          self.prompt = PromptTemplate(template=prompt_template,
23                                       input_variables=["text", "cluster_id", "cluster_head"])
24          self.llm = ChatOpenAI(temperature=0, model=model_name,
            ↪  openai_api_key=os.getenv('OPENAI_API_KEY'))
25          self.llm = self.llm.with_structured_output(schema=Cluster)
26          self.chain = self.prompt | self.llm
27          self.examples: list[BaseMessage] = convert_examples_to_messages()
28
29      async def _run_chain(self, text: str, cluster_id: int, cluster_head: dict):
30          return await self.chain.ainvoke({"text": text, "cluster_id": cluster_id, "cluster_head":
            ↪  cluster_head, "examples": self.examples})
31
32      # Define a function to run multiple chains concurrently
33      async def _run_multiple_chains(self, container: DataContainer) -> list[Cluster]:
34          data_list: list[Cluster] = container.data_list
35          tasks = [self._run_chain(text=cluster.text,
            ↪  cluster_head=cluster.cluster_head.model_dump(), cluster_id=cluster.cluster_id) for
            ↪  cluster in data_list]
36          results = await asyncio.gather(*tasks)
37          return results
38
39      def get_coreferences(self, container: DataContainer) -> list[Cluster]:
40          return asyncio.run(self._run_multiple_chains(container=container))
```

Python-Code 9: LangChain pipeline for COREF

The LangChain chain is built in line 26 of the above code and connects a filled PromptTemplate with an LLM module, in this case OpenAI's ChatOpenAI module running their **gpt-4o** model.

In line 25, the ChatOpenAI module via the function

with_structured_output(schema=Cluster)

is bound to Pydantic's Cluster class, as discussed previously. This forces the LLM to return its response in a format that is compliant with that class. As the Generative LLM approach is run via a Docker container and http requests, the Pydantic class instances are converted to serializable Python dictionaries and JSON and back to Pydantic class instances in the process.

Processing the LLM Response Ideally, the response from the LLM is in the desired format and, among other information, contains the values for the variables *coref_text* and *coref_with_surroundings*. They will now be converted to values that can be attached as custom extensions in the Doc-object of the spacy pipeline by the following function:

```python
    def convert_llm_response_to_matches(self, llm_response: list[dict], unique_ents_with_cust_exts: list[EntsWithCustExts]):
        """ Unfortunately, Generative LLMs do not extract substring indices for their extractions well. So this must be done here. """
        matches: list[SearchMatch] = []
        for cluster, unique_ent in zip(llm_response, unique_ents_with_cust_exts):
            try:
                text: str = cluster['text']
                coreferences: list[dict] = cluster['coreferences']
                for coref in coreferences:
                    coref_with_surroundings: str = coref['coref_with_surroundings']
                    coref_text: str = coref['coref_text']
                    pattern_outer: str = rf"(?:{re.escape(coref_with_surroundings)})"
                    pattern_inner: str = rf"(?:{re.escape(coref_text)})"
                    for m_outer in list(re.finditer(pattern_outer, text)):
                        if m_outer:
                            text_outer: str = m_outer.group(0)
                            start_outer: int = m_outer.start()
                            m_inner: list[re.Match] = list(re.finditer(pattern=pattern_inner, string=text_outer))
                            if m_inner:
                                start_inner = m_inner[0].start()
                                end_inner = m_inner[0].end()
                                start = start_outer + start_inner
                                end = start_outer + end_inner
                                search_match: SearchMatch = SearchMatch(comp_name=unique_ent.comp_name, comp_symbol=unique_ent.comp_symbol, text=coref_text, label=self.comp_label, start_idx=start, end_idx=end, idx_refer_to=IDXReferTo.CHARS)
                                matches.append(search_match)
            except:
                # ToDo: Logger
                print(exc_info_formatter(msg='convert_llm_response_to_matches failed.'))
        return matches
```

Python-Code 10: Convert LLM response to Matches

The Coreference position index within the text are determined by using REGEX functions and patterns.

First, the position indexes of the *coref_with_surroundings* text within the article text, and afterward the *coref_text* within the *coref_with_surroundings* text are determined. A REGEX search for the *coref_text* in the article text <u>alone</u> would not be sufficient, because the *coref_text* could be a common word such as *it*, which might occur multiple times in the text without being a Coreference. The two required words to the left and right of the actual Coreference word ensure a text sequence of at least five Tokens which is unlikely to occur twice in the given article text.

The process for each Coreference yields a Python *re Match object* that contains the start and end index of the *coref_text* within the article text.

Before the information is attached to the Token or Span custom extensions of the Doc-object, the function *resolve_span_conflicts_and_set_new_ents()* again checks if there is an overlap with custom extension values that were already set by the previous NER pipeline component (see: Python-Code 7).

Collecting Entities Once the spacy pipeline has run, the company information stored in the custom extensions, is collected.

It could be that a news article contains multiple sentences each mentioning multiple companies. The functions

get_ents_with_custom_extension() and *get_sentences_with_custom_extensions()*

collect the company information fom the custom extensions in the spacy Doc-object and store them in a Python dictionary:

```
    @staticmethod
    def get_ents_with_custom_extension(ents: Doc.ents) -> list[EntsWithCustExts]:
        ents_with_custom_extension: list[EntsWithCustExts] = [EntsWithCustExts(start_char=ent.start_char, end_char=ent.end_char,
       ↪  ent_text=ent.text, comp_name=getattr(ent._, SpacyExt.COMP_NAME.value), comp_symbol=getattr(ent._,
       ↪  SpacyExt.COMP_SYMBOL.value), set_in=getattr(ent._, SpacyExt.SET_IN.value)) for ent in ents if getattr(ent._,
       ↪  SpacyExt.COMP_NAME.value) != ConfigBasic.spacy_init_mark]
        return ents_with_custom_extension if ents_with_custom_extension else None

    @staticmethod
    def get_sentences_with_custom_extensions(processed_doc: Doc) -> list[dict]:
        sents_with_cust_ext_ents: list[dict] = []
        for doc_sent in processed_doc.sents:
            if doc_sent.ents:
                ents_with_cust_ext: list[EntsWithCustExts] = PipeFunc.get_ents_with_custom_extension(ents=doc_sent.ents)
                if ents_with_cust_ext:
                    sent_with_cust_ext_ents = {'sentence': doc_sent.text, 'entities': [asdict(ent) for ent in ents_with_cust_ext]}
                    sents_with_cust_ext_ents.append(sent_with_cust_ext_ents)
        return sents_with_cust_ext_ents
```

Python-Code 11: Collection of custom extension information

They make sure that multiple company mentions in a sentence are accounted for. The resulting nested Python list of dictionaries is compiled for every article containing company information and structurally looks like this sample:

```
[
  {
    "entities": [
      {
        "comp_name": "Hypoport SE",
        "comp_symbol": "HYQ.DE",
        "df_index": 12,
        "start_char": 504,
        "end_char": 515,
        "ent_text": "Hypoport",
        "set_in": "own_regex_search"
      },
      {
        "comp_name": "JDC Group AG",
        "comp_symbol": "JDC.DE",
        "df_index": 12,
        "start_char": 583,
        "end_char": 595,
        "ent_text": "JDC Group AG",
        "set_in": "own_regex_search"
      }
    ],
    "sentence": "Digitalplattform-Experte Marcus Rex, der zuvor bei Hypoport SE die
    ↪  Versicherungsaktivitaeten leitete, verstaerkt den Vorstand der JDC Group AG als neuer CSO
    ↪  CMO."
  },
  {
    "entities": [
      {
        "comp_name": "JDC Group AG",
        "comp_symbol": "JDC.DE",
        "df_index": 12,
        "start_char": 1627,
        "end_char": 1650,
        "ent_text": "Technologieunternehmens",
        "set_in": "llm_coref_resolve"
      }
    ],
    "sentence": "Der Fokus des Technologieunternehmens liege zunaechst auf dem Ausbau des
    ↪  Vertriebs in Asien."
  }
]
```

Python-Code 12: Company information for article text

Sentences that contain more than one company mention within each dictionary store the information in another list of Python dictionaries. For sentences that contain only one company mention, this list only contains one item.

Topic Sentences The news articles are stored in a parquet file to be converted to a pandas DataFrame as described in Section 2. Each row in the DataFrame contains one article and each list of Python dictionaries, as shown in the Python-Code 12, is stored in one cell of the *ner_coref* column as object.

To do Topic Modelling on sentences as described in the next chapter, the nested list of Python dictionaries (Python-Code 12) must be flattened first to have one sentence in each row of the DataFrame. This is what the

convert_nested_ner_coref_dict()

function in the Python file *main_process.py* does:

```
78      @staticmethod
79      def convert_nested_ner_coref_dict(df: pd.DataFrame) -> pd.DataFrame:
80          df['art_id'] = df.index
81          df = df.explode('ner_coref').reset_index(drop=True)
82          df['top_sent'] = df['ner_coref'].str['sentence'].astype(object).replace(np.nan, pd.NA)
83          df['top_sent_masked'] = df['ner_coref'].apply(SpacyProcess.mask_sent)
84          return df
```

Python-Code 13: Un-Nesting NER and COREF Information

Before inserting new rows into the DataFrame, the function in line 80 inserts a new *art_id* column that is just the row index of the DataFrame. As each row contains just one article, the new column preserves the article identification of the soon-to-be expanded DataFrame. The pandas DataFrame function *explode* in line 51 then creates new rows for each dictionary (shown in Python-Code 12) and pandas *string accessor* method in line 52 finally inserts the sentences into the new column *top_sent*.

5.3.3. Comparing Pre-Trained vs. Generative LLM approach

The Generative LLM approach outperforms the pre-trained Crosslingual-Coreference model [21] on the sample text I tested them with. The latter model performs well if the article text mentions only one company, but links Coreferences to the wrong company if multiple companies are mentioned. Here, the pre-trained Crosslingual-Coreference model [21] might show its lack for contextual understanding as the underlying LLM is only used for finding and resolving Coreference Clusters (see:

Section 5.2.1). The Generative LLM is not perfect either as it does not find all Coreferences, but those that are found are mostly correct. In addition, the Generative LLM was only provided with four few-shot examples in the Prompt and adding more examples would probably improve the model further. I thus have decided to use the Generative LLM approach for the task of Coreference resolution.

5.3.4. Information Extraction Pipeline

After the Coreference Resolution component has run, the spacy pipeline has attached the found company information to the respective custom extensions that in a subsequent step will be stored as a list of dictionaries as shown in Python-Code 12:

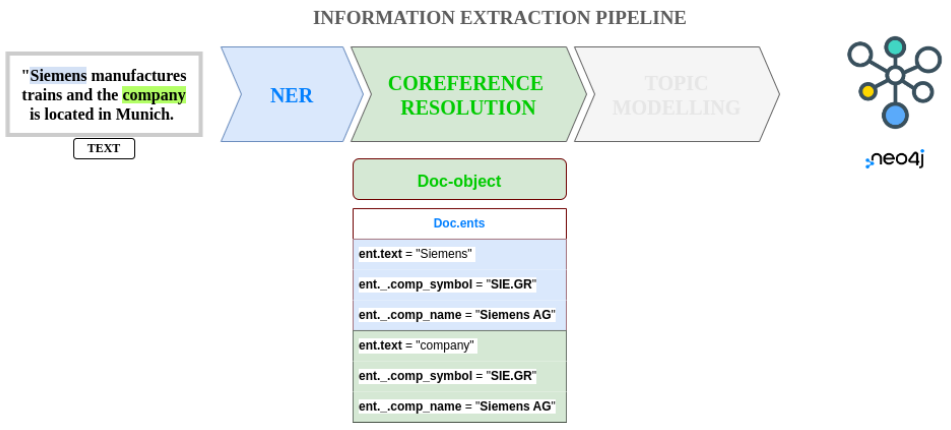

INFORMATION EXTRACTION PIPELINE
"Siemens manufactures trains and the company is located in Munich.
TEXT
NER
COREFERENCE RESOLUTION
TOPIC MODELLING
neo4j
Doc-object
Doc.ents
ent.text = "Siemens"
ent._.comp_symbol = "SIE.GR"
ent._.comp_name = "Siemens AG"
ent.text = "company"
ent._.comp_symbol = "SIE.GR"
ent._.comp_name = "Siemens AG"

Figure 5.6.: spacy pipeline after COREF component

In the example, it contains the company name *Siemens* coming from the NER component and the Coreference *company* coming from the Coreference Resolution component.

6. Topic Modelling

6.1. Information Extraction Types

Information Extraction in NLP can be subdivided into multiple fields [61], among them:

- **Relation Extraction**: Extraction and classification of relations between named entities
- **Event Extraction**: Extraction of a temporal Event
- **Topic Modelling**: Assigment of a topic to a short text or document

The initial expectation for this project was to extract information from the article sentences in the form of triples:

Triple: Subject – Relation – Object

where either the Subject or the Object would be a corporate entity (ORG).

If the Subject were a company, the Object could either be another company or any other common named entity type from the set of ORG, PER or LOC, and vice versa.

The assumption was that these Subjects and Objects would appear in the same sentence as in the following example:

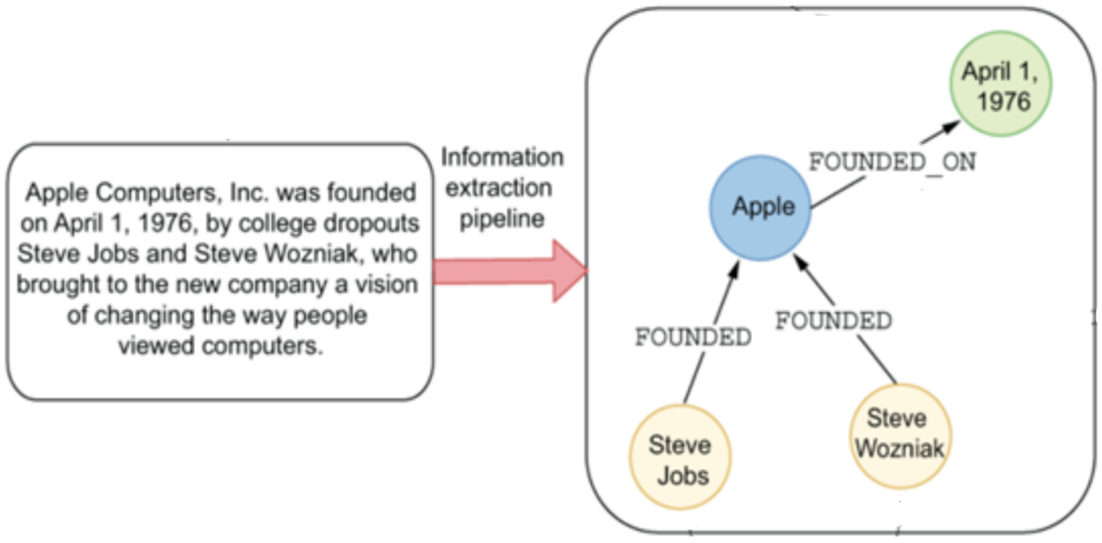

Apple Computers, Inc. was founded on April 1, 1976, by college dropouts Steve Jobs and Steve Wozniak, who brought to the new company a vision of changing the way people viewed computers.
Information extraction pipeline
April 1, 1976
FOUNDED_ON
Apple
FOUNDED
FOUNDED
Steve Jobs
Steve Wozniak

Figure 6.1.: Triple: Subject: Person - Relation: Founded - Object: Apple

After investigation of the news articles, it became clear that the limitation to only extract Relation triples was too restrictive. Many article sentences only contained one common named entity type or described an Event rather than a relationship between entities.

In some other sentences, Subjects or Objects could have been assigned to a wider and less common set of entity types such as *Product*, *Money* or *Law*, but this would have required to enlarge the set with many new custom entity types to cover the most fundamental themes in the finance domain.

It would also have meant to increase the complexity of the Knowledge Graph and deviate from the goal of the project, which was to extract information from news articles with a focus on particular companies. To answer the central question

What is the company news all about?

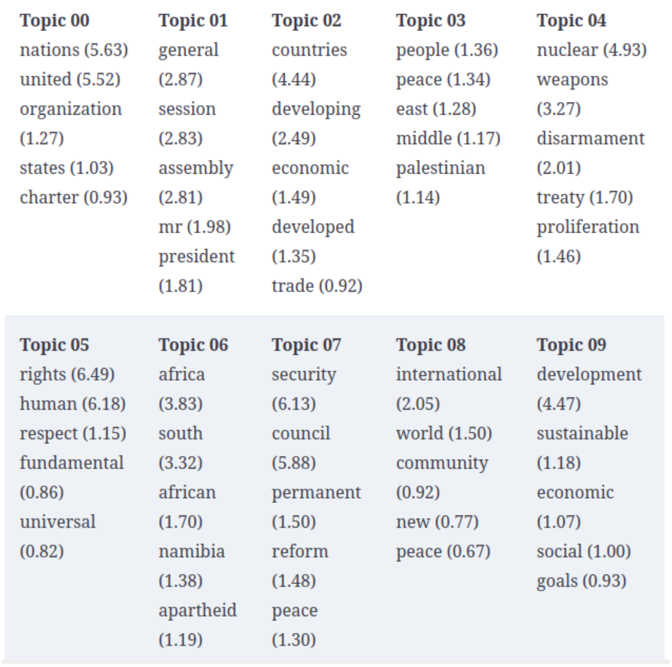
a more general and more comprehensive extraction type was needed.

Topic Modelling can be done on the sentence level and does not require entity pairs. Topics can cover Events as well as Relations between entities and a wide array of finance themes. Extracting information via Topic Modelling thus seems more appropriate for the data and task given than Relation Extraction.

6.2. Traditional Topic Modelling

Most of the traditional Topic Modelling methods are based on absolute (Bag-of-Word: Section 3.3.2) or relative (TF-IDF: Section 3.3.5) word counts as discussed in Section 3.

They also return topics in the form of most frequent words where the user must first read the words per topic to get a sense of what the topic is all about, see Figure 6.2.



Topic 00	Topic 01	Topic 02	Topic 03	Topic 04
nations (5.63)	general (2.87)	countries (4.44)	people (1.36)	nuclear (4.93)
united (5.52)	session (2.83)	developing (2.49)	peace (1.34)	weapons (3.27)
organization (1.27)	assembly (2.81)	economic (1.49)	east (1.28)	disarmament (2.01)
states (1.03)	mr (1.98)	developed (1.35)	middle (1.17)	treaty (1.70)
charter (0.93)	president (1.81)	trade (0.92)	palestinian (1.14)	proliferation (1.46)
Topic 05	**Topic 06**	**Topic 07**	**Topic 08**	**Topic 09**
rights (6.49)	africa (3.83)	security (6.13)	international (2.05)	development (4.47)
human (6.18)	south (3.32)	council (5.88)	world (1.50)	sustainable (1.18)
respect (1.15)	african (1.70)	permanent (1.50)	community (0.92)	economic (1.07)
fundamental (0.86)	namibia (1.38)	reform (1.48)	new (0.77)	social (1.00)
universal (0.82)	apartheid (1.19)	peace (1.30)	peace (0.67)	goals (0.93)

Figure 6.2.: 10 Topics and their most frequent words. Source: [11]

6.2.1. NMF

In the Non-Negative Matrix Factorization method (NMF), the most frequent words are retrieved by doing a matrix decomposition of the Document Term Matrix (see Section 3.2) which in the following Figure is dubbed *Documents-Words Matrix*:

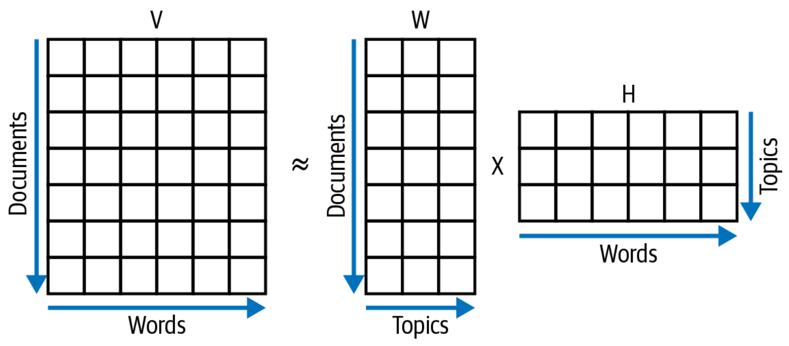


Figure 6.3.: NMF: Source: [11]

The Documents-Topics matrix in the middle of Figure 6.3 maps Documents to Topics whereas the Words-Topics matrix on the right maps topics to their features, i.e. the words in the Vocabulary. In most implementations, a rank parameter (or *n_components* parameter in Scikit-Learn's implementation [69]) can be provided so that the word dimension of the Document Term Matrix first gets reduced before it is decomposed. Because of this, NMF can also be used as a dimension reduction method [69]. To get the most frequent words, each row in the Words-Topics matrix is sorted by its highest values with their respective column indexes being looked up in the Vocabulary. The number of topics can be chosen arbitrarily but a smaller number will ”throw” more words into one topic which makes the decomposition less exact [11].

6.2.2. SVD

A very similar method is the Singular Value Decomposition (SVD) that decomposes the Document Term Matrix into three matrices:

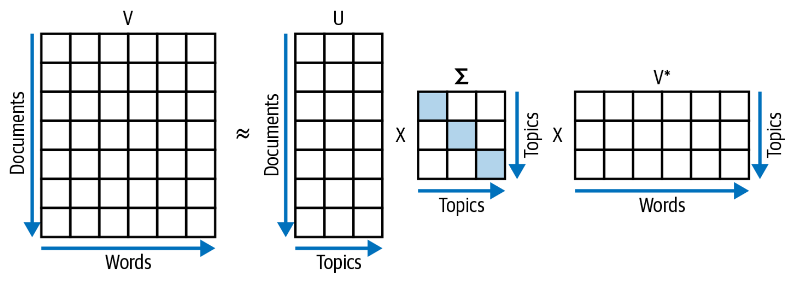


Figure 6.4.: SVD: Source: [11]

The nice thing about the SVD is that the quadratic Topics-Topics matrix (third matrix from left in Figure 6.4) on its diagonal contains the singular values which express the importance of each topic in a document. A higher singular value indicates a topic that captures more of the variance (or information) in the corpus [11]. From the most right Words-Topic matrix, the most important words per topic again can be extracted as described above (Section 6.2.1).

6.2.3. LDA

Latent Dirichlet Allocation (LDA), before the arrival of LLMs, had been the most prominent method for Topic Modelling [11]. LDA, contrary to the previous methods above, is a probabilistic method that is based on the assumption that each Document contains a mix of a few different topics. The method starts by randomly allocating words to each topic and also allocating topics to each document according to a Dirichlet distribution (*Dirichlet prior*) [11]. It then tries to re-create the words from the original document with stochastic sampling. At the end of the optimization process, LDA also returns topics as a list of most frequent words as the NMF and LDA methods do, but with a different distribution.

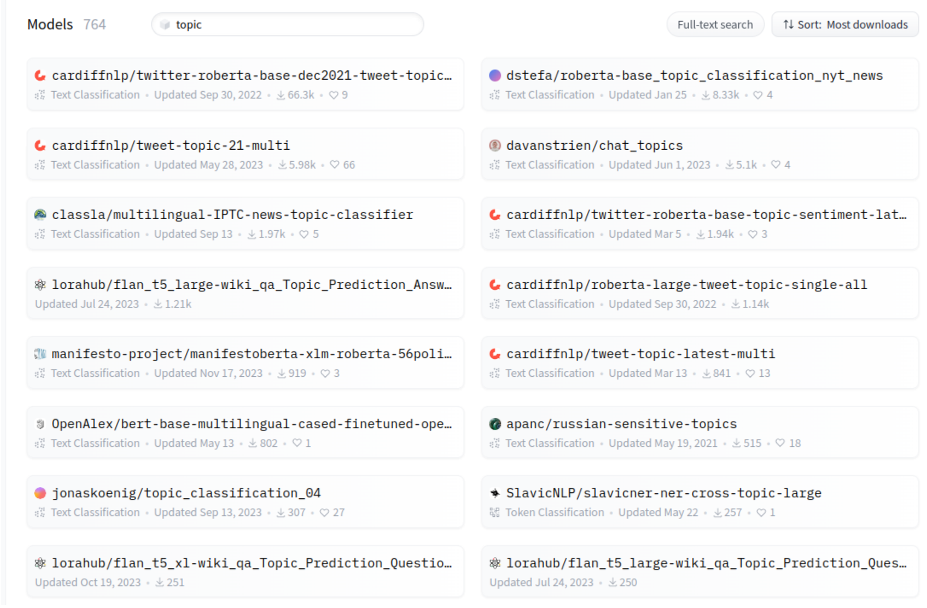
6.3. Embedding-based Topic Modelling

As with most other fields in NLP, the advent of LLMs also changed the way Topic Modelling can be approached today. As the input data or features of traditional Topic Models are static words, they do to not take into account the word's changing contextual meaning in different sentences, as already discussed in Section 3.4. Thus, it is no wonder that today's best performing Topic Models are all embedding-based [63].

LLMs can be used for Topic Modelling either with Pre-Trained or with Generative LLMs.

6.3.1. Pre-Trained Topic Models

Topic Modelling can be considered a classical text classification task and there are many pre-trained, publicly available models on HuggingFace [20] (see Figure 6.5) or other platforms, even for multilingual text.



cardiffnlp/twitter-roberta-base-dec2021-tweet-topic…
Text Classification • Updated Sep 30, 2022 • 66.3k • 9

dstefa/roberta-base_topic_classification_nyt_news
Text Classification • Updated Jan 25 • 8.33k • 4

cardiffnlp/tweet-topic-21-multi
Text Classification • Updated May 28, 2023 • 5.98k • 66

davanstrien/chat_topics
Text Classification • Updated Jun 1, 2023 • 5.1k • 4

classla/multilingual-IPTC-news-topic-classifier
Text Classification • Updated Sep 13 • 1.97k • 5

cardiffnlp/twitter-roberta-base-topic-sentiment-lat…
Text Classification • Updated Mar 5 • 1.94k • 3

lorahub/flan_t5_large-wiki_qa_Topic_Prediction_Answ…
Updated Jul 24, 2023 • 1.21k

cardiffnlp/roberta-large-tweet-topic-single-all
Text Classification • Updated Sep 30, 2022 • 1.14k

manifesto-project/manifestoberta-xlm-roberta-56poli…
Text Classification • Updated Nov 17, 2023 • 919 • 3

cardiffnlp/tweet-topic-latest-multi
Text Classification • Updated Mar 13 • 841 • 13

OpenAlex/bert-base-multilingual-cased-finetuned-ope…
Text Classification • Updated May 13 • 802 • 1

apanc/russian-sensitive-topics
Text Classification • Updated May 19, 2021 • 515 • 18

jonaskoenig/topic_classification_04
Text Classification • Updated Sep 13, 2023 • 307 • 27

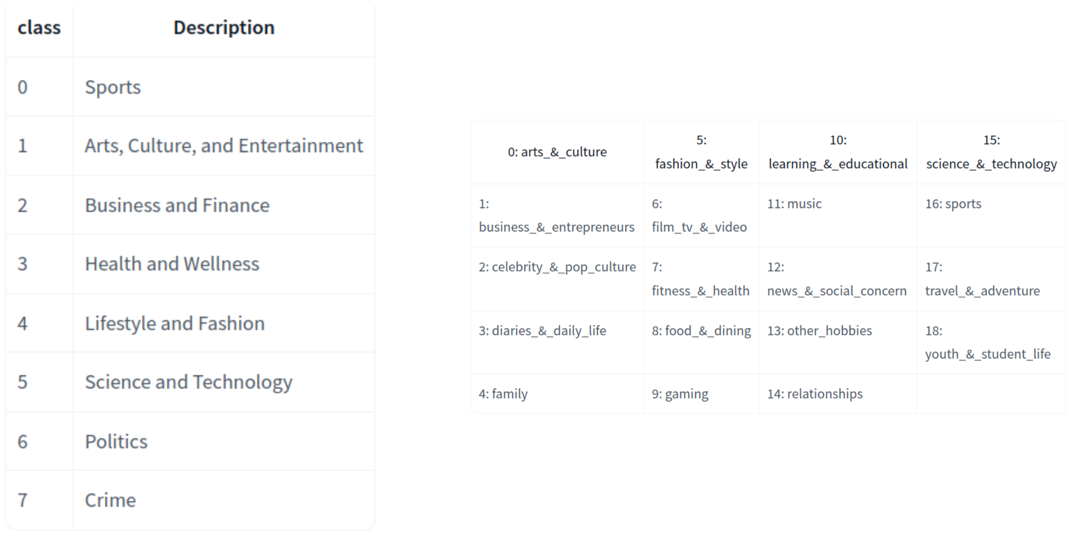
SlavicNLP/slavicner-ner-cross-topic-large
Token Classification • Updated May 22 • 257 • 1

lorahub/flan_t5_xl-wiki_qa_Topic_Prediction_Questio…
Updated Oct 19, 2023 • 251

lorahub/flan_t5_large-wiki_qa_Topic_Prediction_Ques…
Updated Jul 24, 2023 • 250

Figure 6.5.: Hugging Face Topic Models. Source: [20]

Unfortunately, the pre-trained Topic Models on Hugging Face [20] are either trained on labels that are very broadly categorized (see Figure 6.6) or very tightly confined to a narrow domain or language.



class	Description
0	Sports
1	Arts, Culture, and Entertainment
2	Business and Finance
3	Health and Wellness
4	Lifestyle and Fashion
5	Science and Technology
6	Politics
7	Crime

0: arts_&_culture	5: fashion_&_style	10: learning_&_educational	15: science_&_technology
1: business_&_entrepreneurs	6: film_tv_&_video	11: music	16: sports
2: celebrity_&_pop_culture	7: fitness_&_health	12: news_&_social_concern	17: travel_&_adventure
3: diaries_&_daily_life	8: food_&_dining	13: other_hobbies	18: youth_&_student_life
4: family	9: gaming	14: relationships	

Figure 6.6.: Hugging Face Topic Models - Typical labels. Sources: [16][17]

After extensively researching the domains of these models, I conclude that none of them fits the corporate financial news domain and the given data very well.

It could be possible to fine-tune a model and I would expect the performance of such models to be quite good. But fine-tuning would require to label and compile a lot of training samples which would go beyond the scope of this thesis.

6.3.2. BERTopic

Another kind of pre-trained model is BERTopic [38]. BERTopic is similar to traditional models in that it returns a most-common-word list per topic for which the user must define a topic label manually. But it differs from traditional models in that it does not use word vectors but Sentence Transformer (or SBERT) [66] embeddings. BERTopic further uses dimension reduction and clustering techniques. SBERT is a modification of the pre-trained BERT model that uses siamese and triplet network structures to derive semantically meaningful sentence embeddings that can be compared using cosine-similarity. This reduces the effort for finding the most similar pair of sentences significantly, while maintaining the accuracy from BERT [66].

BERTopic [38] is a modular system in the sense that individual components can be substituted depending on the dataset and use case. The modules and main steps to compose BERTopic [38] are:

1. **Embeddings**: Choose and apply an embedding component, typically SBERT [66]
2. **Dimension Reduction**: Choose and apply a dimension reduction method on the embeddings
3. **Clustering**: Choose and apply a clustering algorithm on the dimension-reduced embeddings
4. **Aggregate Text**: Aggregate the text of all documents within each cluster
5. **Apply TF-IDF Vectorization**: Apply TF-IDF vectorization to each of the per-Cluster-aggregated texts [1]
6. **Most Frequent Words**: Get the most frequent words for each cluster according to TF-IDF

This approach ensures that the distinction of Topic Clusters is made on the basis of contextual embeddings (Step 1). But it also ensures, that the Topic that each Cluster represents, can also be presented as a collection of its most frequent words (Step 4 and 5).

BERTopic [38] can be downloaded and is pip-installable. Most of the components in BERTopic are built with existing libraries such as Scikit-Learn. I also wanted to compare it with traditional Topic modelling approaches, so I decided to implement it myself. In this proprietary implementation not only word embeddings can be used, but also traditional word vectors such as those used in a classic Bag-of-Word (Sec.3.3.2) or TF-IDF (Sec.3.3.5) approach.

The modules for the self-implemented models can be found in the

src/E_topic_model/traditional

directory. The model follows the modularity and architecture of BERTopic in that the steps to build the model are similar:

1. **topic_prepare**: If desired, preprocesses the text to reduce the dimension of the Vocabulary as described in Section 3.3.4.

[1]In BERTopic, this is called class-based TF-IDF or *c-TF-IDF*

2. **topic_vectorize**: Choose between TF-IDF (Sec.3.3.5), Bag-of-Word (Sec.3.3.2), One-Hot (Sec.3.3.1) vectorization or Sentence Transformers embeddings
3. **topic_dim_reduce**: Choose between multiple dimension reduction methods such as PCA, NMF, UMAP, etc.
4. **topic_cluster**: Choose between multiple cluster methods such as HDB-SCAN, KMEANS, MEANSHIFT, etc.
5. **topic_model**: Run all of the above components and calculate Topics by presenting the most frequent words of each Cluster
6. **topic_visualize**: Reduce the dimension of the word vectors or embeddings to three and display the data and Clusters in a Plotly Scatter-3d-Graph.

In the same directory, there are two Jupyter notebooks that can be used for model training and prediction: *topic_model_train.ipynb* and *topic_model_test.ipynb*.
The Jupyter Notebook *topic_model_train.ipynb* was used to extensively test different combinations of text processing approaches, vectorization methods and cluster algorithms, but the results were disappointing.

Sentence Transformer Embeddings When using the Sentence Transformer embedding method as BERTopic does by default, the Cluster algorithms all had difficulties in separating the data points, not only visually (see Figure 6.7), but also by separating semantically coherent words and sentences into different Clusters.

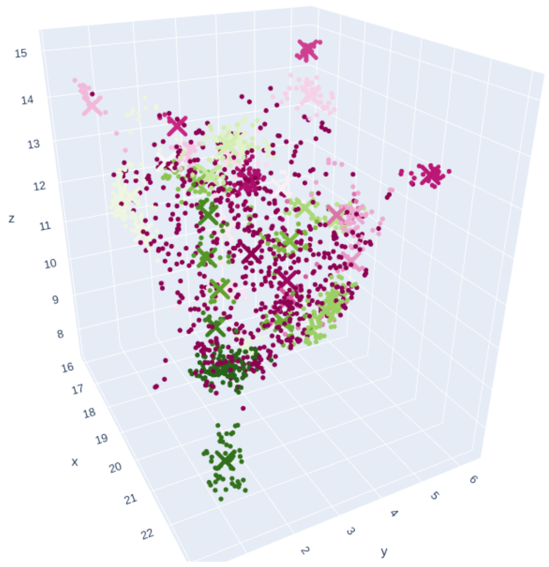


Figure 6.7.: SBERT/Sentence Transformer embedding Cluster

In Figure 6.7, UMAP was used to reduce the dimension to 20 and HDBSCAN was used to find clusters. Although there are some clusters that are clearly separate from others, most of the clusters are within a close distance and the separation of datapoints changes quickly with slightly different model parameters. The same words appear in many clusters and manually looking at some individual sentences leads to the conclusion that sentences within clusters are not coherent. The results were similar for other combinations of dimension reduction and clustering algorithms.

Word Vectors Looking at models were word vectors instead of embeddings were used, the performance of these models is not better:

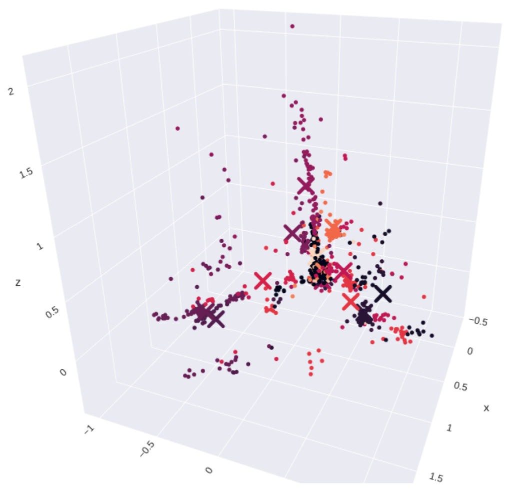


Figure 6.8.: TF-IDF Cluster

In Figure 6.8, TF-IDF was used for vectorization, PCA for dimension reduction and KMEANS for clustering. The models suffer from the same shortcomings as the embedding based model above: same words appear in many clusters and sentences within clusters are not coherent.

Performance Although the financial news domain spreads across a wide range of topics, these topics are very nuanced and all have to do with company news. I attribute the performance problems of the above methods to the fact that the differences between the individual sentences expressed as distances in vector space are probably too small to extract meaningful clusters.

6.4. Topic Modelling with Generative LLMs

Similar to creating a Generative LLM model for Coreference Resolution, the Generative LLM for Topic Modelling was also created with LangChain [71]. As the problem of unresolvable dependency conflicts with existing Python libraries prevailed, the approach was again isolated with Docker. The files for building the Docker container can be found in the

/src/E_topic_model/img_llm_extract_topics

directory.

Data Model The *data_models.py* file in this directory contains the Python Enum *TopicExplain* with the desired topic classes and a short description of each topic:

```python
class TopicExplain(str, Enum):
    """ The Topic of the sentence. Topics can only be one of the following: """
    topic1 = ("Sätze mit konkreten Zahlenangaben aus Quartals- oder Jahresberichten. Die genannten Zahlen beziehen sich auf die
    ↪  Bilanz, den Umsatz- oder die Gewinn- und Verlustrechnung (GuV). "
              "Beispiele dafür sind EBIT, EBITDA, Gewinn oder Verlust vor Steuern, Gewinn- oder Verlustmargen, der Umsatz,
              ↪  Veränderungen der Größen über einen Zeitraum, etc.")
    topic2 = "Sätze mit allgemeinen Aussagen und Einschätzungen zu Unternehmensergebnissen, die Bilanzerung und den Umsatz. Dies
    ↪  sind Wertungen, oft von Verantwortlichen im Unternehmen, die keine konkreten Zahlen beinhalten. "
    topic3 = ("Sätze, die sich auf eine bevorstehende oder vergangene Hauptversammlung oder die Veröffentlichung von
    ↪  Unternehmensergebnissen beziehen, ohne dass dabei konkrete Zahlen genannt werden. "
              "Beispiele dafür sind die Ankündigung einer Veröffentlichung von Quartals- oder Jahresberichten oder Informationen zu
              ↪  bzw. über eine Hauptversammlung.")
    topic4 = "Zukunftsgerichteter Ausblick, Prognosen, Ziele, Strategie und Pläne der Unternehmensleitung."
    topic5 = "Sätze, die Kennzahlen zu Unternehmensergebnissen beinhalten, ohne dass dabei ganze Sätze gebildet werden oder die
    ↪  Zahlen beschrieben und erläutert werden. Beispiele dafür sind tabellenartige Angaben von Kennzahlenvariablen und deren
    ↪  Werte wie: 'EBITDA EUR 23 Mio.'"
    topic6 = "Sätze, in denen die Aktivitäten und das Profil des Unternehmens dargestellt wird. Oft dienen die Sätze der positiven
    ↪  Selbstdarstellung seitens des Managements, dem Brand-Marketing oder einer allgemeinen Unternehmensbeschreibung."
    topic7 = "Stimmrechte, Kapitalveränderungen, Dividenden, Finanzierung, Listing an Börsen, Marktkapitalisierung."
    topic8 = "Sätze, in denen das vom Unternehmen angebotene Produkt, eine Produktentwicklung oder ein neue Neuerung im Hinblick
    ↪  auf ein Produkt des Unternehmens beschrieben wird."
    topic9 = "Sätze, in denen die Herstellung des Produkts, der Produkt-Forschung, die Exploration vn Bodenschätzen, Produkt- oder
    ↪  Medikamenten-Zulassungen, dem Finden neuer Resourcen oder anderen dem Herstellungsprozess nahen Themen geht."
    topic10 = "Konzernumbau, wichtige organisatorische Veränderungen, Restrukturierung, Werksstilllegung, strategische
    ↪  Partnerschaften, Übernahmen"
    topic11 = "Personalveränderungen im Vorstand, Aufsichtsrat, Betriebsrat oder anderer Organe im Unternehmen, Personal,
    ↪  Gewerkschaftem, Streiks"
    topic12 = "Kunden, Marktanteile, Absatzmärkte, Umsätze, Absatzpreise"
    topic13 = "Einflüsse von Aussen auf die Erfolgsaussichten von Unternehmen etwa durch Subventionen, Staatliche Eingriffe,
    ↪  Umbrüche im Markt, politische Veränderungen, Umwelteinflüsse, etc."
    topic14 = "Einschätzungen Unternehmensfremder/Analysten zu einem Unternehmen"
    topic15 = "Unfälle, Gewalt, Katastrophen"
    topic16 = "Unvollständige Sätze mit einzelnen, nicht-zusammenhängenen Worten, ohne Kontext, die wahrscheinlich falsch
    ↪  formattiert oder im vorangehenden Text-Reinigungsprozess falsch gesplittet wurden. "
    topic17 = "Alle anderen topics, die den oben genannten 16 topics nicht zugeordnet werden können."
```

Python-Code 14: Topic Model: Pre-Defined topics

There are 17 pre-defined topics formulated in German that were manually crafted by reading some of the article's sentences. *topic16* is dedicated to sentences that are incomplete and *topic17* to all topics that are not covered by the previous 16 topics.

The process of defining topics could be facilitated by letting an LLM classify a set of sample sentences into an arbitrary number of topics, summarize each topic and express its overarching theme in a short sentence. If manually crafted though, for a production use case, a more thorough semantic analysis of the article sentences and a re-assessment over the number of topics was also necessary.

The directory also contains a *topics.py* file that provides sample sentences for each topic which are later used in the Prompt as few-shot examples. Some of these few-shot examples are shown in Python-Code 15:

```python
# Note: Sätze, in denen die Aktivitäten und das Profil des Unternehmens dargestellt wird. Oft dienen die Sätze der positiven Selbstdarstellung seitens des Managements, dem Brand-Marketing oder einer allgemeinen Unternehmensbeschreibung.
top6 = [
'Die Comp@Name@Placeholder ist aufgrund ihrer zwei Jahrzehnte langen Erfahrung im Hanfanbau optimal in der Lage, Privatpersonen und gemeinschaftliche Anbauvereinigungen zu beliefern.',
'Die aktuellen Weiterentwicklungen von Comp@Name@Placeholder eroeffnen uns neue Optionen unter anderem in der Visualisierung von Business Intelligence, der Datenzuordnung und -modellierung.',
'Comp@Name@Placeholder gilt mit einer Flotte von 250 Containerschiffen und einer Transportkapazitaet von 1,8 Millionen TEU als fuenftgroesste Reederei der Wel, hinter Comp@Name@Placeholder, Comp@Name@Placeholder, Comp@Name@Placeholder und dem Primus Comp@Name@Placeholder.',
'Die Comp@Name@Placeholder ist in Europa und Nordamerika der fuehrende Omnichannel-Haendler fuer Geschaeftsausstattung.',
'Die Gruppe ist mit den Divisions Industrial & Packaging, Office Furniture & Displays und FoodService in mehr als 25 Laendern vertreten.',
'Im ersten Quartal begann Comp@Name@Placeholder mit der Implementierung neuer Technologien im Bereich Document Workflow Management, um die operative Effizienz zu staerken und den Kunden digitale Loesungen mit hoeherem Mehrwert zu bieten.',
'Basierend auf der vorhandenen Produktpalette, die auf das Segment kleinerer Briefvolumina ausgerichtet ist, und dank des hohen Anteils an wiederkehrenden Umsaetzen, verfuegt das Unternehmen ueber ein robustes Geschaeftsmodell und investiert in die weitere Entwicklung des Frankiergeschaefts.',
]
```

Python-Code 15: Topic Model: Few Shot Examples

As the few-shot examples often contain concrete company names, these company names therein were replaced by a *Comp@Name@Placeholder* mask to avoid that the Generative LLM tries to match the company name of the sample sentence.

LangChain Code The LangChain code used in the Docker container is laid out in the *topic_langchain.py* file of the same directory, see Python-Code 16:

```
20 class TopicLangchain:
21     def __init__(self, prompt_template: str, model_name: str = "gpt-4o"):
22         nest_asyncio.apply()
23         self.prompt = PromptTemplate(template=prompt_template, input_variables=["user_data",
           ↪  "topics"]).partial(pattern=re.compile(r"\`\`\`\n\`\`\`"))
24         self.llm = ChatOpenAI(temperature=0, model=model_name,
           ↪  openai_api_key=os.getenv('OPENAI_API_KEY'))
25         self.llm = self.llm.with_structured_output(schema=Frame)
26         self.chain = self.prompt | self.llm
27         self.examples: list[BaseMessage] = convert_examples_to_messages()
28         self.topics: str = str({i.name: i.value for i in TopicExplain})
29 
30     def make_frame(self, df: pd.DataFrame) -> Frame:
31         return Frame(indexes=df.index.values, sentences=df.sentences.values)
32 
33     def format_prompt_template(self, df: pd.DataFrame):
34         frame = self.make_frame(df)
35         prompt_template = self.prompt.template.format(user_data=frame, topics=self.topics,
           ↪  examples=self.examples)
36         return prompt_template
37 
38     def get_topics(self, df: pd.DataFrame):
39         frame = self.make_frame(df)
40         return self.chain.invoke({"user_data": frame, "topics": self.topics, "examples":
           ↪  self.examples})
```

Python-Code 16: LangChain Topic Model

In line 26, the chain is built by *piping* or connecting LangChain's Prompt component with the LLM component. The Prompt contains the previously discussed few-shot *examples* (line 27) compiled and organized by a Python function into an appropriate format. It also contains the topics and their short descriptions (line 28) coming from the Python Enum *TopicExplain* discussed above.

The sentences to be topic-classified are inserted into the Prompt (line 40) via an instance of a Python *Frame* class (see Python-Code 17) that inherits from Pydantic's BaseClass [70]. The same class in line 25 is also used in LangChain's *ChatOpenAI* module function *llm.with_structured_output(schema=Frame)*

to force the LLM to return its response in the format of this class, as explained in Section 5.3.2.

```python
class Frame(BaseModel):
    """ DataFrame that contains the index of the DataFrame and the column "top_sent" which contains
    ↪  the sentences for which a topic shall be determined. """
    indexes: list[int] = Field(description='The indexes of the rows in the pandas DataFrame')
    sentences: list[str] = Field(default=None, description='List of sentences each for which the
    ↪  Topic shall be determined.')
    topics: list[Topic] = Field(default=None, description='List of Topic enums for each sentence
    ↪  in "sentences". List must be of same length as "sentences" list.')
```

Python-Code 17: Frame: A Pydantic BaseModel class

Aggregation of Sentences The *Frame* class, as the name implies, shall carry the data of a pandas DataFrame, namely the row *indexes* of the DataFrame, the name of the column that contains the *sentences* to be topic-classified and the *topics* classes of those sentences to be returned by the Generative LLM. The sentences to be classified are aggregated within a pandas DataFrame because the few-shot examples baked into the Prompt are rather long. It would be very costly if the Prompt with its long few-shot examples would be sent for each sentence individually and separately.

The Token Limit Problem If there are many and long sentences to be topic-classified, the token limit that all Generative LLMs have imposed, will be reached. To avoid that, the pandas DataFrame, later to be converted to a *Frame* instance, first must be split into chunks.

This is what the function

get_topics_from_gen_llm() (Python-Code 18)

in the *main_process.py* file will do.

```python
122     def get_topics_from_gen_llm(self, df: pd.DataFrame, chunk_size: int = 30, df_col_name: str =
        ↪ 'topic') -> pd.DataFrame:
123         if 'top_sent' not in df.columns:
124             raise ValueError('No "top_sent" columns')
125         indexes = []
126         topics = []
127         for idx_start in range(0, len(df.index), chunk_size):
128             df_chunk = df.iloc[idx_start:idx_start + chunk_size]
129             frame = Frame.df_to_instance(df_chunk)
130             try:
131                 resp = requests.post(url=self.topic_gen_llm_docker_container_url,
                    ↪ json=frame.dict(exclude_none=True))
132                 if resp.ok:
133                     indexes.extend(resp.json()['indexes'])
134                     topics.extend(resp.json()['topics'])
135                 else:
136                     print(f'Error occurred. Will try again... ')
137                     time.sleep(2)
138                     resp = requests.post(url=self.topic_gen_llm_docker_container_url,
                        ↪ json=frame.dict(exclude_none=True))
139                     if resp.ok:
140                         indexes.extend(resp.json()['indexes'])
141                         topics.extend(resp.json()['topics'])
142                     else:
143                         print(f'Error occured the second time, could not be cured: {resp.text}')
144                         indexes.extend([i for i in range(idx_start, idx_start + chunk_size)])
145                         topics.extend(['topic17' for _ in range(idx_start, idx_start +
                            ↪ chunk_size)])
146                 print('DEBUG INFO:', topics)
147             except:
148                 print(f'Error occured: {traceback.format_exc()}')
149                 indexes.extend([i for i in range(idx_start, idx_start + chunk_size)])
150                 topics.extend(['topic17' for _ in range(idx_start, idx_start + chunk_size)])
151 
152             time.sleep(1)
153 
154         df[df_col_name] = topics
155         return df
```

Python-Code 18: Frame: A Pydantic BaseModel class

It first chunks the passed DataFrame into a default size of 30 rows (line 127), converts it to a *Frame* instance (line 129) and sends the JSON-ized instance to the Docker container waiting for requests (line 131). The LangChain model, as described above, in the Docker container forwards the Examples- and Topic-enhanced Prompt request to an OpenAI server and returns the response from there, ideally in JSON-format (that complies with the *Frame* class format), in line 138.

If the procedure succeeds, the topic classification of the Generative LLM for each DataFrame chunk first gets appended to a Python list that later will be used to create a new *topics* column in the DataFrame. If the procedure fails, it is tried once again and the request is resend. If the procedure still fails, *topic17* as the default spare class will be appended to the Python list in line 152.

The function returns the pandas DataFrame with an extra *topics* column that contain one of the 17 Enum topics for each sentence.

6.4.1. Comparing Pre-Trained vs. Generative LLM approach

As outlined above, the cluster based methods such as the self-implemented BERTopic model, did not perform well on the Topic Modelling task. In addition, they require to manually formulate a topic based on the most frequent words of a cluster.

The Generative LLM on the other side did perform well on the sentences I tested it with. Most of the sentences were classified correctly. For sentences that were expected to be classified differently, the actual and expected topic classes and examples were ambiguous and probably need some refinement.

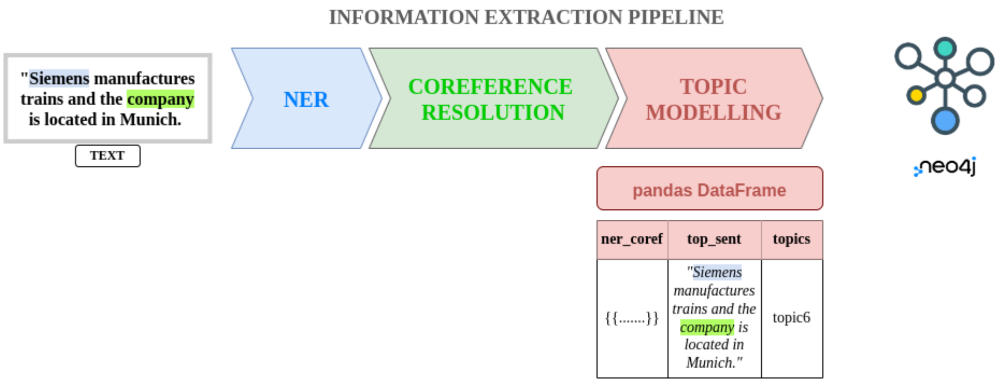
I attribute the good performance of the Generative LLM to the power of a really large LLM (such as OpenAI's 4o model) that has the capacity to understand and distinguish even the most nuanced context.

I would expect a pre-trained LLM, such as those available on HuggingFace, to perform equally well after fine-tuning it with domain-specific topic labels and training data.

But as I did not go through the extensive process to compile and label such training data, in the project the Generative LLM was used for Topic Modelling.

6.5. Information Extraction Pipeline

After the Topic Modelling component has run, the Generative LLM has filled the pandas DataFrame column *topics* in respect to each sentence in the column *top_sent*:


INFORMATION EXTRACTION PIPELINE
"Siemens manufactures trains and the company is located in Munich.
TEXT
NER
COREFERENCE RESOLUTION
TOPIC MODELLING
pandas DataFrame
neo4j

ner_coref	top_sent	topics
{{.......}}	*"Siemens manufactures trains and the company is located in Munich."*	topic6

Figure 6.9.: DataFrame after Topic component

In the example above, the DataFrame contains all the information coming from the NER and Coreference components and one of the 17 possible topics for each sentence in which a company was found.

7. Knowledge Graph[1]

7.1. Introduction

As outlined in Chapter 1, the goal of this project was to extract structured information from unstructured text and store this information in a Knowledge Graph. A Knowledge Graph is an organized representation of real-world entities and their relationships which is typically stored in a graph database such as neo4j [85]. A graph database stores data as *Nodes*, *Relationships* and *Properties* in a graph-like [26] structure instead of in tables or documents [84]. This enables the database to traverse *Nodes* and *Relationships* more quickly than relational databases, that would use computationally expensive *JOIN* operations to retrieve inter-related information [85].

All the files for this chapter can be found in the *src/F_knowledge_graph* directory.

7.2. Knowledge Graph Creation

A Knowledge Graph is defined by an *Organizing Principle* [85] or a *Schema*. The schema of the Knowledge Graph in this project is depicted in Figure 7.1.

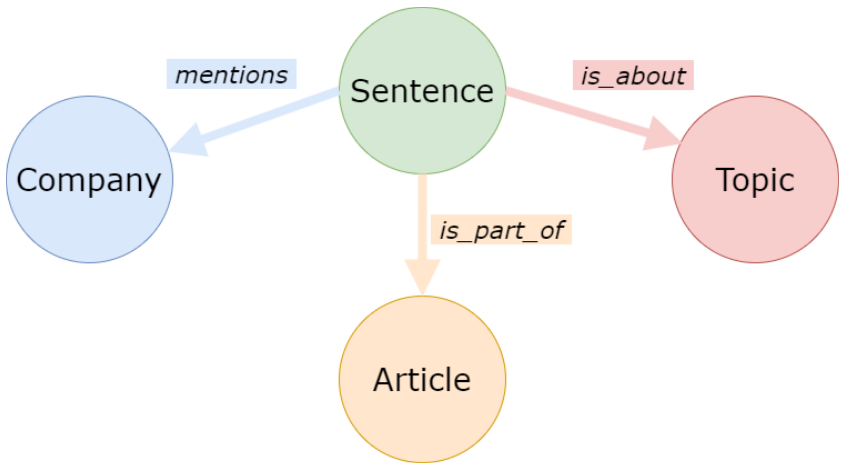


Figure 7.1.: Knowledge Graph Schema

[1]Part of the Python code for this section was adopted from [37]

The schema contains four Nodes (*Article*, *Sentence*, *Company*, *Topic*) and three relationships (*mentions*, *is_part_of*, *is_about*), as described in Section 1.

Ontology A Knowledge Graph *Schema* can be outlined in different ways, for instance by using an *Ontology*. An *Ontology* uses a modelling language to formally specify the concepts and relationships in a human- and machine-understandable way [85].

The most widely used ontology modelling languages are OWL (*Web Ontology Language*) [27] and RDFS (*Resource Description Framework Schema*) [96]. Their most fundamental construct is a triple [28] consisting of

Subject - Predicate - Object

tuples. Multiple triples can be composed into a graph structure because objects can be subjects and vice versa and both can be connected via predicate relationships. The textual presentation of the Ontology in this project is done with a Turtle [30] file (suffix: *.ttl*), a common data serialization format for storing triples. The *NewsArticles.ttl* Ontology file (Python-Code 19-21) was created with Protege [86], an open-source ontology editor by Stanford University.

```
1  @prefix rainergo: <http://www.semanticweb.org/rainergo/ontologies/NewsArticles#> .
2  @prefix owl: <http://www.w3.org/2002/07/owl#> .
3  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
4  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
5  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
6  @base <http://www.semanticweb.org/rainergo/ontologies/NewsArticles/> .
7
8  rainergo: rdf:type owl:Ontology .
```

Python-Code 19: NewsArticles.ttl - Prefixes

Turtle files usually start with prefixes that refer to other Ontology namespaces [89] and their vocabulary [89].

In line 8 of the *NewsArticles.ttl* file, the triple

subject	*predicate*	*object*
rainergo:	**rdf:type**	**owl:Ontology**

states that the prefix *rainergo* in line 1 is of type *Ontology* where *type* and *Ontology* themselves are defined in triples that are accessible through their public IRIs in lines 2 and 3.

```
87  #################################################################
88  #    Classes
89  #################################################################
90
91  ###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles/Article
92  rainergo:Article rdf:type owl:Class .
93
94  ###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles/Company
95  rainergo:Company rdf:type owl:Class .
96
97  ###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles/Sentence
98  rainergo:Sentence rdf:type owl:Class .
99
100 ###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles/Topic
101 rainergo:Topic rdf:type owl:Class .
```

Python-Code 20: NewsArticles.ttl - Classes or Nodes

In lines 87 to 101, the classes or Nodes of the Knowledge Graph schema are defined, again with triples.

```
10  #################################################################
11  #    Object Properties
12  #################################################################
13
14  ###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles/is_about
15  rainergo:is_about rdf:type owl:ObjectProperty ;
16            rdfs:domain rainergo:Sentence ;
17            rdfs:range rainergo:Topic .
18
19  ###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles/is_part_of
20  rainergo:is_part_of rdf:type owl:ObjectProperty ;
21              rdfs:domain rainergo:Sentence ;
22              rdfs:range rainergo:Article .
23
24  ###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles/mentions
25  rainergo:mentions rdf:type owl:ObjectProperty ;
26            rdfs:domain rainergo:Sentence ;
27            rdfs:range rainergo:Company .
```

Python-Code 21: NewsArticles.ttl - Relationships

In lines 10 to 27, the relationships are defined with three triples for each relationship where the second (predicate: *domain*) and third (predicate: *range*) triple represent its source and target Node.

```
#################################################################
#    Data properties
#################################################################
###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles/art_id
rainergo:art_id rdf:type owl:DatatypeProperty ;
            rdfs:domain rainergo:Article ;
            rdfs:range xsd:int .

###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles#art_datetime
rainergo:art_datetime rdf:type owl:DatatypeProperty ;
              rdfs:domain rainergo:Article ;
              rdfs:range xsd:dateTime .

###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles#art_text
rainergo:art_text rdf:type owl:DatatypeProperty ;
              rdfs:domain rainergo:Article ;
              rdfs:range xsd:string .

###  http://www.semanticweb.org/rainergo/ontologies/NewsArticles#art_source
rainergo:art_source rdf:type owl:DatatypeProperty ;
              rdfs:domain rainergo:Article ;
              rdfs:range xsd:string .
```

Python-Code 22: NewsArticles.ttl - Properties

In Python Code 22, lastly the Node's properties and their data types are defined (only shown for the Node type *Article*).

The Turtle file represents the human- and machine-readable presentation of the Schema that in Figure 7.1 was visualized as chart.

Graph Preparation To translate the Ontology into a neo4j schema and lay the foundation for the Knowledge Graph, the Python library *rdflib* [87] was used in the

F_knowledge_graph/A_rdf_graph.py

module. The functions in the *RDFGraph* class parse the triples in the *NewsArticles.ttl* file into an RDF [96] graph and construct Cypher [95] query templates based on this graph for subsequent data imports. Cypher [95] is a declarative graph query language whose query commands are similar to those of the popular

SQL [97] database language used for relational databases.

The functions to operate on the neo4j graph database and to run Cypher queries can be found in the

F_knowledge_graph/B_graph_construction

module within the *GraphConstruction* class. They require the neo4j database and the neo4j Python driver to be installed [84].

Data Preparation In order to align the Knowledge Graph schema (Fig.7.1) with the extracted data in the pandas DataFrame and insert the data into the Knowledge Graph, some data preparation is necessary.

For the Topic Modelling task, the pandas DataFrame must contain one row for each sentence to be topic-classified and this was achieved by the

convert_nested_ner_coref_dict() function (Python-Code 13)

as outlined in Section 5.3.2. But each sentence might contain multiple company references as shown in Python-Code 23 where two companies, *Hypoport SE* and *JDC Group AG*, are mentioned in one sentence:

```
[
  {
    "entities": [
      {
        "comp_name": "Hypoport SE",
        "comp_symbol": "HYQ.DE",
        "df_index": 12,
        "start_char": 504,
        "end_char": 515,
        "ent_text": "Hypoport",
        "set_in": "own_regex_search"
      },
      {
        "comp_name": "JDC Group AG",
        "comp_symbol": "JDC.DE",
        "df_index": 12,
        "start_char": 583,
        "end_char": 595,
        "ent_text": "JDC Group AG",
        "set_in": "own_regex_search"
      }
    ],
    "sentence": "Digitalplattform-Experte Marcus Rex, der zuvor bei Hypoport SE die Versicherungsaktivitaeten leitete, verstaerkt den Vorstand der JDC Group AG als neuer CSO CMO."
  },
```

Python-Code 23: Company information for each sentence

To prepare the DataFrame for the Knowledge Graph, the information in the Python list of dictionaries (line 4-20 in Python-Code 23) must be further resolved so that each DataFrame row only contains one company mention.
This is what the function *prepare_df_for_kg()* in the *main_process.py* file does:

```
 91     def prepare_df_for_kg(self, df: pd.DataFrame) -> pd.DataFrame:
 92         df['ner_coref_entities'] = df.ner_coref.str['entities']
 93         df = df.explode('ner_coref_entities').reset_index(drop=True)
 94         df['comp_symbol'] = df.ner_coref_entities.str['comp_symbol']
 95         df['comp_name'] = df.ner_coref_entities.str['comp_name']
 96         df['top_description'] = df.topic.apply(lambda x: TopicExplain[x].value)
 97         df.drop_duplicates(subset=['top_sent', 'comp_symbol', 'comp_name'], keep='last',
            ↪  inplace=True)
 98         df = self._drop_nans(df)
 99         df = self._get_isin(df)
100         return df
```

Python-Code 24: Function: prepare_df_for_kg()

It extracts and inserts the *entities* list into the new DataFrame column *ner_coref_entities* (line 92), inserts new rows that accommodate each company information separately (line 93) and therefrom extracts and pastes the *comp_symbol* and *comp_name* into new columns (line 94-95). There are also news articles in the DataFrame that are only updates of previous articles and in which only a few words might have changed. It is therefore possible, that same sentences appear in more than one DataFrame row, i.e. that there are duplicates which must be removed (line 97).

Company Symbol and ISIN The *comp_symbol* refers to the stock ticker symbol that a company has on a certain stock exchange. Unfortunately, these ticker symbols without their country suffixes are unique only within each stock exchange but not across stock exchanges. ISINs on the other side are unique but unfortunately are not provided by the company data provider OpenBB [15].

Company symbols thus first must be mapped to ISINs to have unique identifiers later to be used for external data retrieval. The function *prepare_df_for_kg()* (Python-Code 24) does this (line 99) based on a manually compiled list and also inserts the ISIN as a new column into the DataFrame.

The data there is now ready to be inserted into the Knowledge Graph.

Data Loading After the *GraphConstruction* method *init_graph()* has set some general parameters for neo4j, the method *load_data_into_knowledge_graph()* in Python-Code 25:

```python
    def load_data_into_knowledge_graph(self, df: pd.DataFrame, show_queries: bool = False):

        def check_if_df_and_onto_match(df: pd.DataFrame, nodes_with_attrs: dict, nodes_without_attrs):
            if nodes_without_attrs:
                raise ValueError(f'There are Nodes in the Ontology that do not have any attributes (owl:DatatypeProperty): {nodes_without_attrs}. Please check!')
            df_columns: list = df.columns.tolist()
            node_attributes: list = [self.ONTO_ATTR_TO_DF_ATTR_MAP[col] for cols in nodes_with_attrs.values() for col in cols]
            not_in_df = [col for col in node_attributes if col not in df_columns]
            if not_in_df:
                raise ValueError(f'The following Node attributes are not a column in the DataFrame: {not_in_df}')

        def get_data_from_df(df: pd.DataFrame, nodes: dict[str, list], relationships: list[dict[str, str]], unique_node_keys: dict[str, list[str]]) -> tuple[list, list]:
            nodes_data: list = list()
            relationships_data: list = list()
            for ind, row in df.iterrows():
                for node, attrs in nodes.items():
                    row_template = {node: {attr: row[self.ONTO_ATTR_TO_DF_ATTR_MAP[attr]] for attr in attrs}}
                    if row_template not in nodes_data:
                        nodes_data.append(row_template)
                for rel in relationships:
                    relationship = {rel['SOURCE'] + '_' + rel['REL'] + '_' + rel['TARGET']: {"source": {rel['SOURCE']: {attr: row[self.ONTO_ATTR_TO_DF_ATTR_MAP[attr]] for attr in unique_node_keys[rel['SOURCE']]}}, "target": {rel['TARGET']: {attr: row[self.ONTO_ATTR_TO_DF_ATTR_MAP[attr]] for attr in unique_node_keys[rel['TARGET']]}}}}
                    relationships_data.append(relationship)
            return nodes_data, relationships_data

        # Note: 1. Get graph and data structure from Ontology
        nodes_with_attrs, nodes_without_attrs = self.rdf_graph.get_nodes_and_node_props()
        check_if_df_and_onto_match(df=df, nodes_with_attrs=nodes_with_attrs, nodes_without_attrs=nodes_without_attrs)
        constraint_queries, node_queries, rel_queries, ns_queries = self.rdf_graph.create_query_templates()
        relationships = self.rdf_graph.get_relationships()

        # Note: 2. Get data from pandas dataframe
        nodes_data, rels_data = get_data_from_df(df=df, nodes=nodes_with_attrs, relationships=relationships, unique_node_keys=self.unique_node_keys)

        # Note: 3. Load data into Knowledge Graph
        session = self.driver.session(database=self.neo4j_db_name)

        for ns_query in ns_queries:
            if show_queries:
                print('ns_query:', ns_query)
            session.run(ns_query)

        for constraint_query in constraint_queries:
            if show_queries:
                print('Constraint_Query: ', constraint_query)
            res0 = session.run(constraint_query)

        for node_data in nodes_data:
            node = list(node_data.keys())[0]
            node_query = node_queries[node]
            if show_queries:
                print('node:', node)
                print('Node_Query:', node_query)
            res1 = session.run(node_query, parameters={'node_data': node_data})

        for rel_data in rels_data:
            rel = list(rel_data.keys())[0]
            rel_query = rel_queries[rel]
            if show_queries:
                print('rel:', rel)
                print('Rels_Query: ', rel_query)
            res2 = session.run(rel_query, parameters={'rel_data': rel_data})
```

Python-Code 25: Function: load_data_into_knowledge_graph()

loads the DataFrame data into the neo4j graph database. It first retrieves the structural information from the Ontology (lines 254, 256 and 257) that was stored in the RDF graph (see Section 7.2), checks if the Node properties of the Knowledge Graphs are present in the DataFrame (lines 255, 231-238), extracts the unique Node and Relationship data from the DataFrame (lines 260, 240-251) and finally uses this data to fill and execute (lines 263-281) the previously created Cypher query templates (see Section 7.2).

The data is now loaded into the neo4j database and can be visually inspected in a browser (URL: *http://localhost:7474/browser/*), see Figure 7.2. For visual clarity, only a few datapoints are loaded and highlighted in the next few charts:

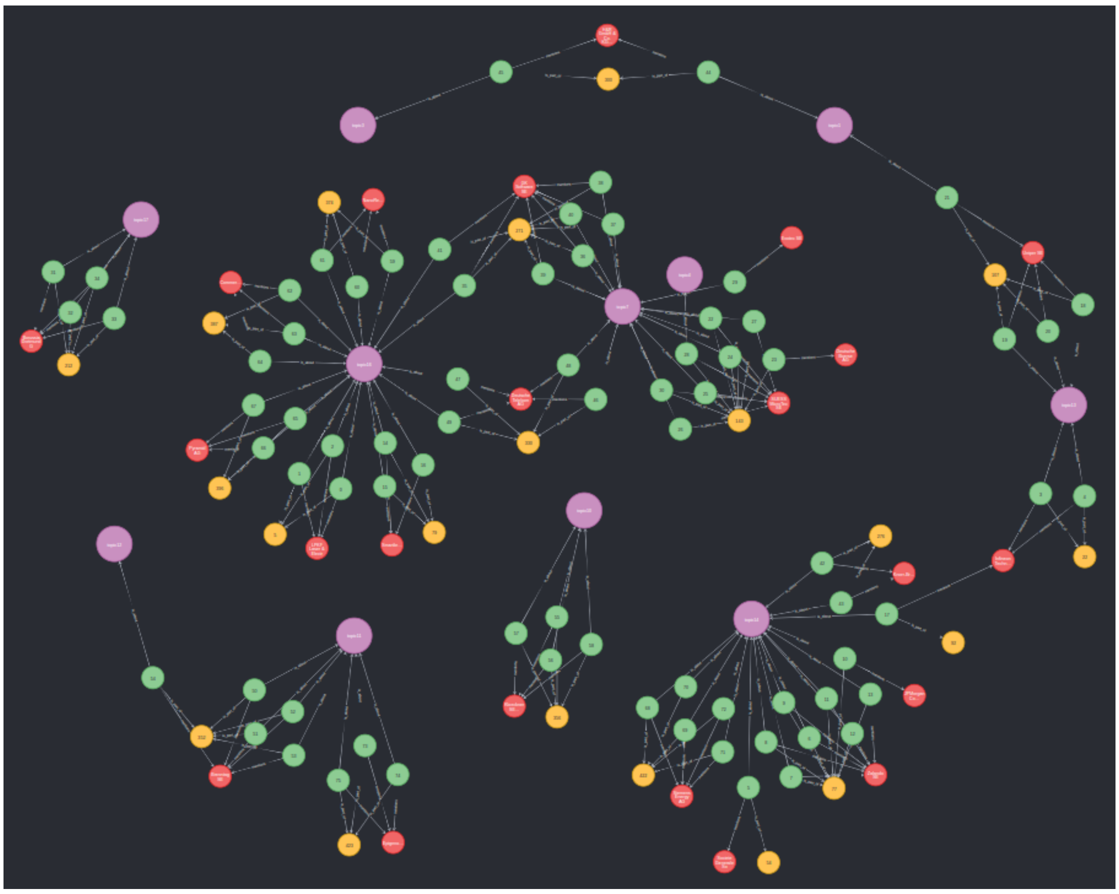


Figure 7.2.: Knowledge Graph after Data Loading

Zooming into an exemplary and disconnected Sub-Graph reveals some Relationships and Node properties, see Figure 7.3:

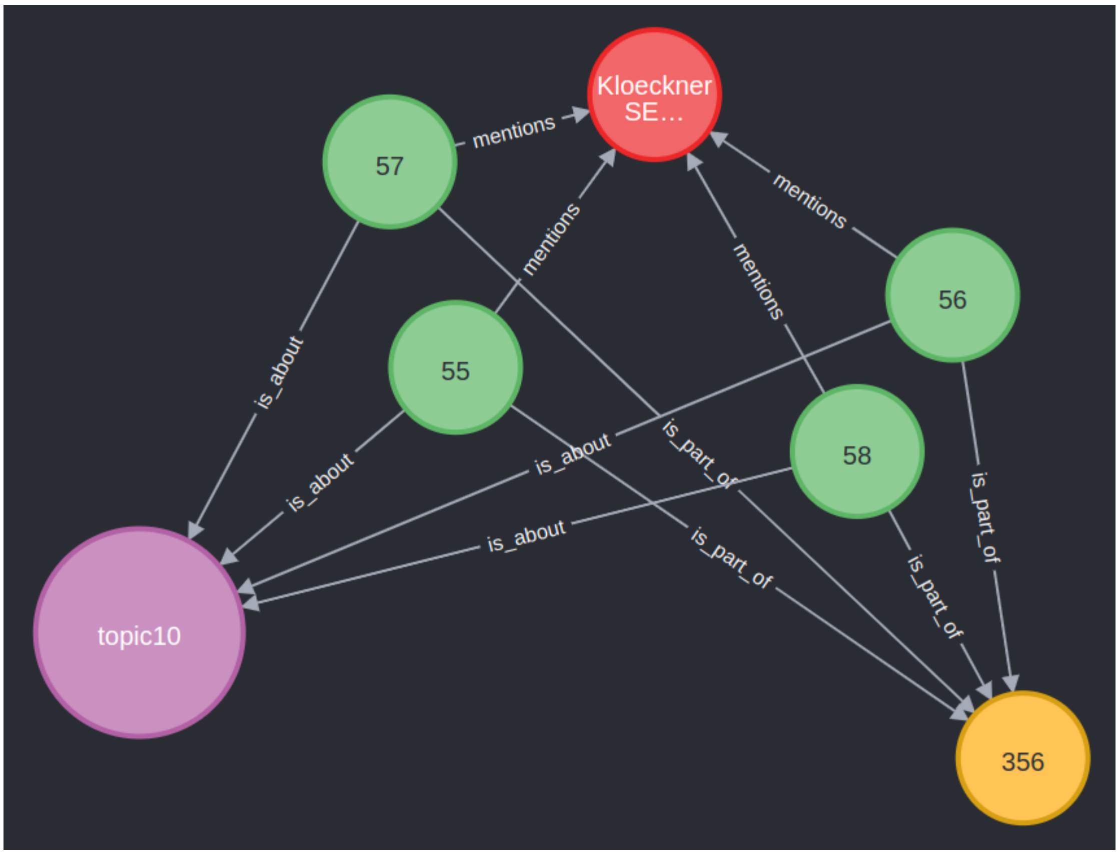
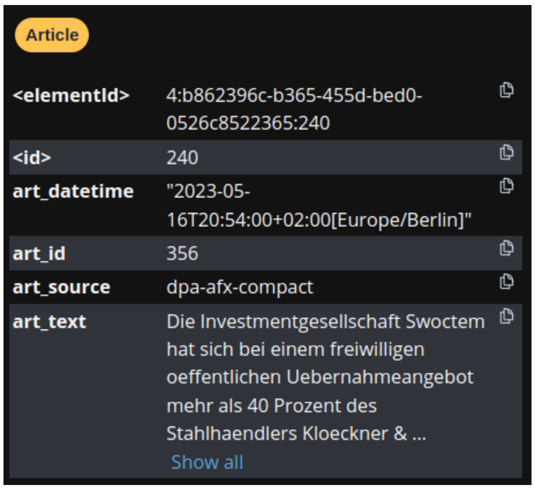


Figure 7.3.: Exemplary, Disconnected Sub-Graph

In the Sub-Graph (Fig.7.3), a pink *Topic* Node with its *top_id*, four green *Sentence* Nodes with their *sent_id*, a yellow *Article* Node with its *art_id* and a red *Company* Node with its *comp_name* properties are displayed.

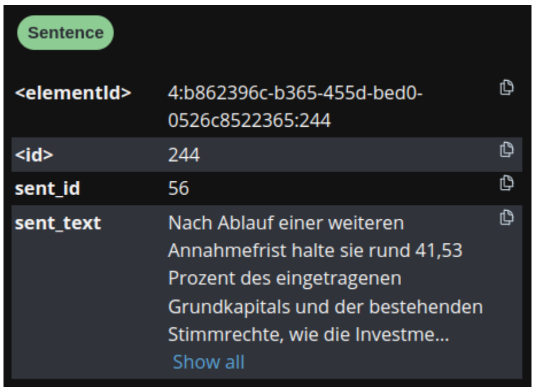
The information for each Node can be displayed individually once they are selected in the browser:



Article

<elementId>	4:b862396c-b365-455d-bed0-0526c8522365:240
<id>	240
art_datetime	"2023-05-16T20:54:00+02:00[Europe/Berlin]"
art_id	356
art_source	dpa-afx-compact
art_text	Die Investmentgesellschaft Swoctem hat sich bei einem freiwilligen oeffentlichen Uebernahmeangebot mehr als 40 Prozent des Stahlhaendlers Kloeckner & ... Show all

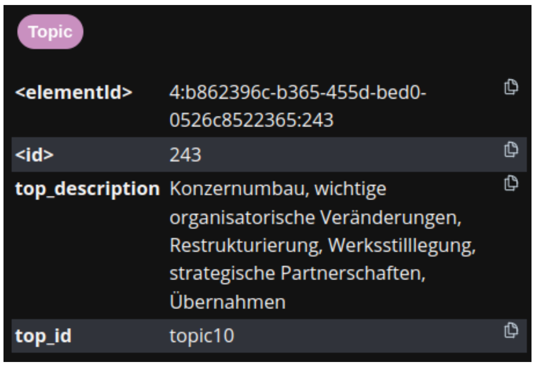
(a) Article



Sentence

<elementId>	4:b862396c-b365-455d-bed0-0526c8522365:244
<id>	244
sent_id	56
sent_text	Nach Ablauf einer weiteren Annahmefrist halte sie rund 41,53 Prozent des eingetragenen Grundkapitals und der bestehenden Stimmrechte, wie die Investme... Show all

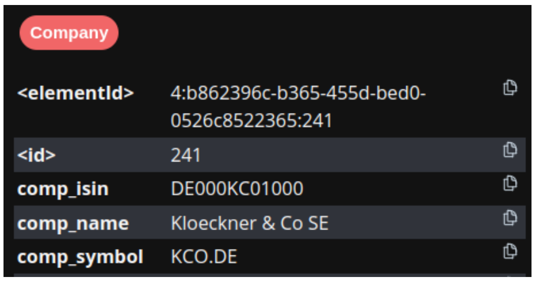
(b) Sentence



Topic

<elementId>	4:b862396c-b365-455d-bed0-0526c8522365:243
<id>	243
top_description	Konzernumbau, wichtige organisatorische Veränderungen, Restrukturierung, Werksstilllegung, strategische Partnerschaften, Übernahmen
top_id	topic10

(c) Topic



Company

<elementId>	4:b862396c-b365-455d-bed0-0526c8522365:241
<id>	241
comp_isin	DE000KC01000
comp_name	Kloeckner & Co SE
comp_symbol	KCO.DE

(d) Company

Figure 7.4.: neo4j: Nodes and their Properties

The four boxes in Figure 7.4 show the Node properties for the respective *Nodes* that were previously loaded.

7.3. External Data

Publicly accessible triple stores [29], such as DBPedia [24] or WikiData [25], can be used to enrich the Knowledge Graph with external data. For external data to be added to the existing Knowledge Graph, a unique and common identifier that exists in these triple stores, is necessary. The previously discussed ISIN, that was mapped from the Company Symbol (see Section 7.2), is such an identifier. The *import_wikidata_id()* function in Python-Code 26 uses the ISIN to retrieve the *Wikidata entity id* [91] for each of the companies in the Knowledge Graph.

```
 84     def import_wikidata_id(self, node: str, node_prop: str, node_prop_wikidata_id: str,
        ↪  wikidata_id_is_part_of: str or None = None):
 85         if wikidata_id_is_part_of:
 86             predicate = (f'(p:{wikidata_id_is_part_of}/pq:{node_prop_wikidata_id})|'
 87                          f'(p:{node_prop_wikidata_id}/ps:{node_prop_wikidata_id})')
 88         else:
 89             predicate = f'wdt:{node_prop_wikidata_id}'
 90
 91         query = rf"""
 92                 MATCH (node:{node})
 93                 WITH
 94                     "SELECT ?{node_prop} ?{self.label_wikidata_id}
 95                         WHERE {{
 96                             FILTER (?{node_prop} = \"" + node.{node_prop} + "\")
 97                             ?{self.label_wikidata_id}   {predicate}   ?{node_prop} .
 98                       }}"
 99                 AS sparql
100                 CALL apoc.load.jsonParams(
101                     "https://query.wikidata.org/sparql?query=" +
102                       apoc.text.urlencode(sparql),
103                     {{ Accept: "application/sparql-results+json"}}, null)
104                 YIELD value
105                 UNWIND value['results']['bindings'] AS row
106                 WITH row['{node_prop}']['value'] AS prop_val,
107                      row['{self.label_wikidata_id}']['value'] AS new_prop_val
108                 MERGE (n:{node} {{ {node_prop}: prop_val }})
109                 SET n.{self.label_wikidata_id} = new_prop_val;
110                 """
111         # print(query)
112         self.driver.execute_query(query_=query, database_=self.neo4j_db_name)
113         self.import_wikidata_id_was_run = True
114         # Note: For Companies that do not have an ISIN (and thus no wikidataID), we must fill the
            ↪  wikidataID attribute with values. Otherwise, there is no attribute at all which later
            ↪  causes problems.
```

Python-Code 26: Function: import_wikidata_id()

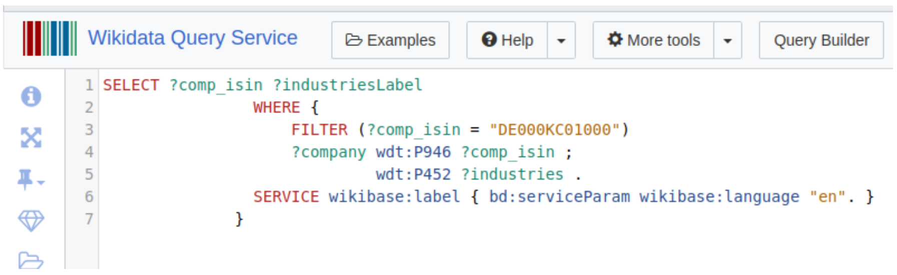
SPARQL The function in lines 94-98 embeds a SPARQL [88] query that is sent via an http-request to the Wikidata Query Service with the URL

https://query.wikidata.org/sparql

referenced in line 101. SPARQL is a query language to query and retrieve stored RDF data such as Wikidata or DBPedia entities [88]. Wikidata and DBPedia are considered publicly available triple stores [29].
neo4j's apoc [83] library processes the SPARQL response (lines 106-109) and inserts the *wikidataID* into the respective property of each *Company* Node, if the

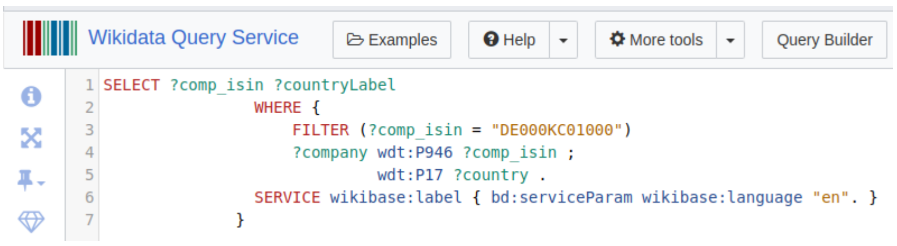
Wikidata entity page contains this information.

There are two other functions that can enrich the Knowledge Graph via SPARQL queries, namely *import_data_from_wikidata()* and *import_data_from_dbpedia()* in the *GraphConstruction* class. They can load any available information from triple stores that have a link to a *Wikidata entity id* [91] or ISIN.
To show some examples, I have used these functions to load the *Industry* category, the home *Country* and a short *Abstract* about the companies from Wikidata and DBPedia. The respective SPARQL queries, that are automatically generated by the two functions, for demonstration purposes were manually composed and are shown in Figure 7.5.



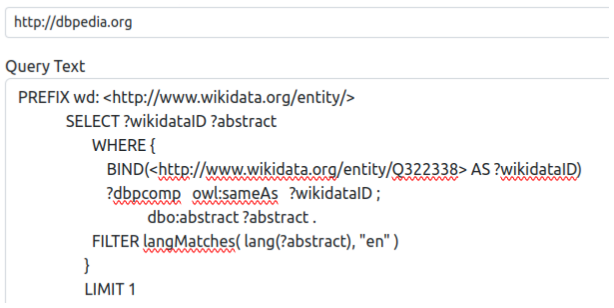
```
SELECT ?comp_isin ?industriesLabel
                WHERE {
                    FILTER (?comp_isin = "DE000KC01000")
                    ?company wdt:P946 ?comp_isin ;
                             wdt:P452 ?industries .
                SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
              }
```

(a) P452 - Industry



```
SELECT ?comp_isin ?countryLabel
                WHERE {
                    FILTER (?comp_isin = "DE000KC01000")
                    ?company wdt:P946 ?comp_isin ;
                             wdt:P17 ?country .
                SERVICE wikibase:label { bd:serviceParam wikibase:language "en". }
              }
```

(b) P17 - Country



http://dbpedia.org

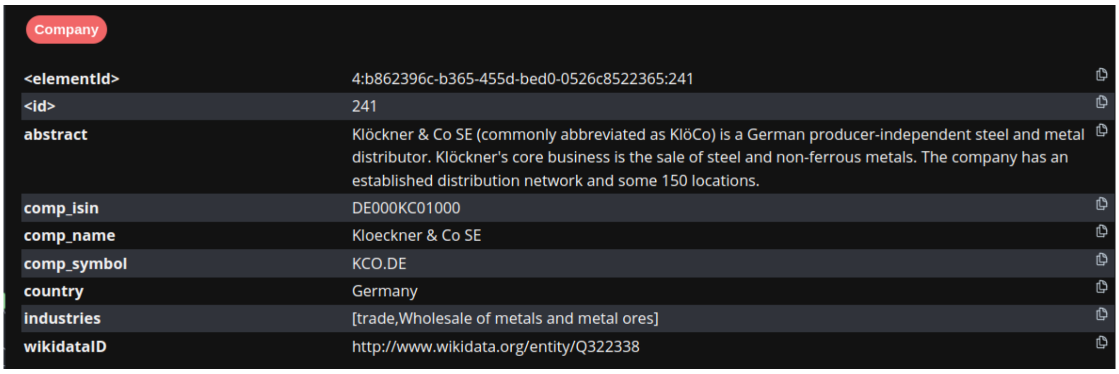
Query Text

```
PREFIX wd: <http://www.wikidata.org/entity/>
     SELECT ?wikidataID ?abstract
        WHERE {
          BIND(<http://www.wikidata.org/entity/Q322338> AS ?wikidataID)
          ?dbpcomp  owl:sameAs  ?wikidataID ;
                dbo:abstract ?abstract .
        FILTER langMatches( lang(?abstract), "en" )
       }
       LIMIT 1
```

(c) dbo:abstract - Abstract

Figure 7.5.: SPARQL: Queries to enrich Knowledge Graph

In the neo4j Subgraph 7.6, the *Company* Node properties for *Kloeckner & Co SE* now include the information for *Industry*, *Country* and an *Abstract*.



Company

<elementId>	4:b862396c-b365-455d-bed0-0526c8522365:241
<id>	241
abstract	Klöckner & Co SE (commonly abbreviated as KlöCo) is a German producer-independent steel and metal distributor. Klöckner's core business is the sale of steel and non-ferrous metals. The company has an established distribution network and some 150 locations.
comp_isin	DE000KC01000
comp_name	Kloeckner & Co SE
comp_symbol	KCO.DE
country	Germany
industries	[trade,Wholesale of metals and metal ores]
wikidataID	http://www.wikidata.org/entity/Q322338

Figure 7.6.: Company Node after Data Enrichment

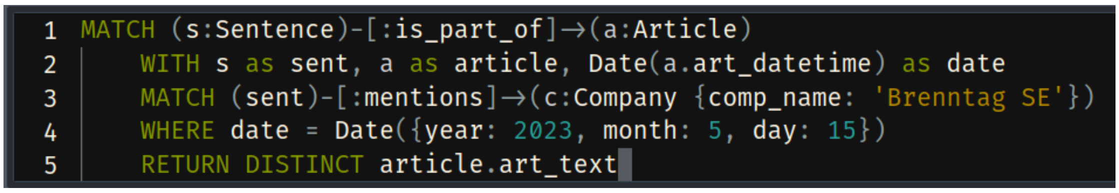
7.4. Information Retrieval

With the data from the DataFrame and from external sources loaded, the Knowledge Graph can now be queried to retrieve information.

Cypher Queries Cypher queries provide a visual way to reveal patterns and relationships by using a human-friendly ASCII-based type of syntax [84].

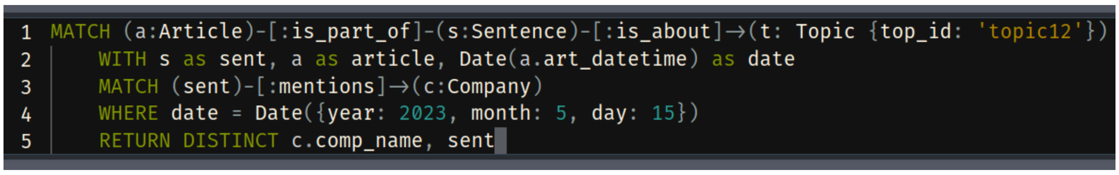
(:nodes) - [:ARE_CONNECTED_TO] → (:otherNodes)

Round brackets are used to represent *(:Nodes)*, and *-[:ARROWS]→* to represent a relationship between the *(:Nodes)*. Cypher queries can be used to reveal a wide range of Knowledge Graph information ranging from simple property values to complex, multi-hop relations.
A user, for instance, could be interested in news articles about a particular company such as *Brenntag SE* or all companies that are mentioned in sentences about *Topic12* on a specific day and so would compose the following Cyper queries:



```
MATCH (s:Sentence)-[:is_part_of]->(a:Article)
    WITH s as sent, a as article, Date(a.art_datetime) as date
    MATCH (sent)-[:mentions]->(c:Company {comp_name: 'Brenntag SE'})
    WHERE date = Date({year: 2023, month: 5, day: 15})
    RETURN DISTINCT article.art_text
```

(a) Cypher Query 1: Articles about *Brenntag SE*

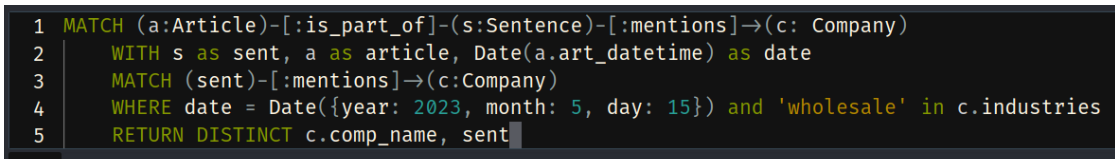


```
MATCH (a:Article)-[:is_part_of]-(s:Sentence)-[:is_about]->(t: Topic {top_id: 'topic12'})
    WITH s as sent, a as article, Date(a.art_datetime) as date
    MATCH (sent)-[:mentions]->(c:Company)
    WHERE date = Date({year: 2023, month: 5, day: 15})
    RETURN DISTINCT c.comp_name, sent
```

(b) Cypher Query 2: Companies, Sentences about *Topic12*

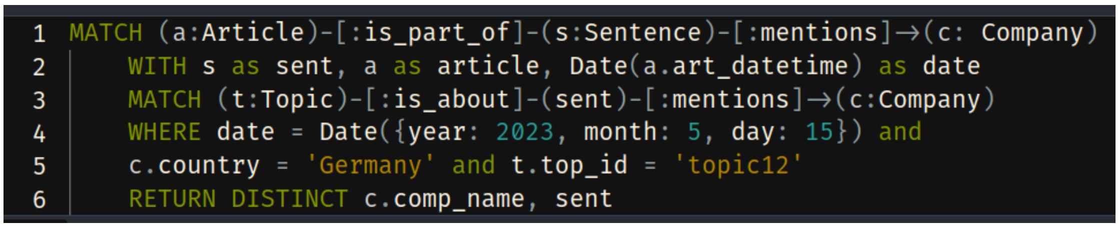
Figure 7.7.: Cypher Queries 1-2

Another user could be interested only in certain industry news for a particular day or only in sentences with certain topics and about companies that are located in Germany. He could compile these Cypher queries:



```
MATCH (a:Article)-[:is_part_of]-(s:Sentence)-[:mentions]->(c: Company)
    WITH s as sent, a as article, Date(a.art_datetime) as date
    MATCH (sent)-[:mentions]->(c:Company)
    WHERE date = Date({year: 2023, month: 5, day: 15}) and 'wholesale' in c.industries
    RETURN DISTINCT c.comp_name, sent
```

(a) Cypher Query 3: Sentences about Industry *Wholesale*



```
MATCH (a:Article)-[:is_part_of]-(s:Sentence)-[:mentions]->(c: Company)
    WITH s as sent, a as article, Date(a.art_datetime) as date
    MATCH (t:Topic)-[:is_about]-(sent)-[:mentions]->(c:Company)
    WHERE date = Date({year: 2023, month: 5, day: 15}) and
    c.country = 'Germany' and t.top_id = 'topic12'
    RETURN DISTINCT c.comp_name, sent
```

(b) Cypher Query 4: German Companies, Sentences about Topic12

Figure 7.8.: Cypher Queries 3-4

The queries could be even more specific and nested and so reveal deep interrelations between *Companies*, *Articles*, *Topics* and *Sentences* and their particular properties. The queries can be executed via the Python plugin or directly in the browser console and will be returned either as Graph visualization or as String in JSON format.

7.5. Sentence Embeddings and Sentiment

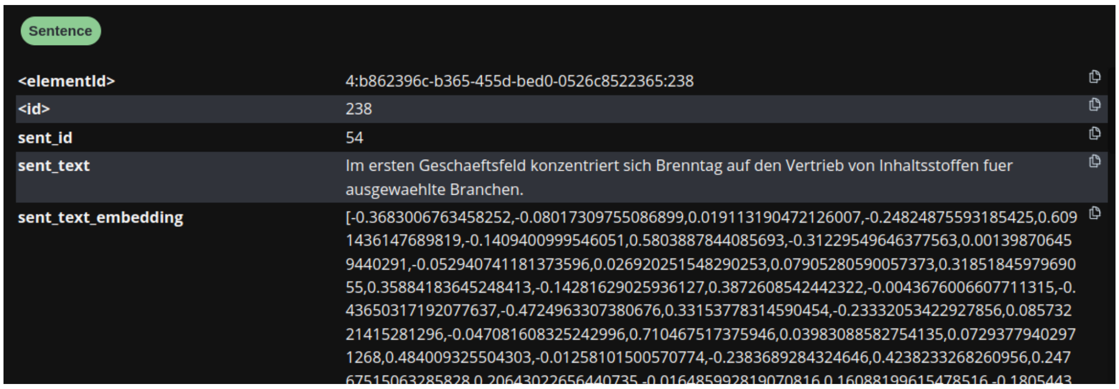
So far, the *Sentence* information is only stored as text in the Node's property *sent_text*. But the *Embedder* class in the *F_knowledge_graph* directory allows to contextually embed (see Section 3.4.2) these sentences.

The method *create_text_embedding()* (Python-Code 27) in the previously cited *GraphConstruction* class conveniently embeds all the sentences in the Knowledge Graph and stores the embedding vector of size 768 in the new *Sentence* Node property *sent_text_embedding*:

```python
    def create_text_embedding(self, node_label: str, node_primary_prop_name: str, prop_to_embed: str,
                              vector_size: int = 768,
                              similarity_method: str = "cosine"):
        name_embedded_prop = prop_to_embed + "_embedding"
        query_index = f"""CALL db.index.vector.createNodeIndex('{"NodeIndex" + "_" + node_label + "_" + prop_to_embed}',
                          '{node_label}', '{name_embedded_prop}', {vector_size}, '{similarity_method}' ) ; """
        query_prop_to_embed = f"""
        MATCH (n:{node_label})
        RETURN n.{node_primary_prop_name} AS {node_primary_prop_name}, n.{prop_to_embed} AS {prop_to_embed}
        """
        session = self.driver.session(database=self.neo4j_db_name)
        try:
            res = self.driver.execute_query(query_=query_index, database_=self.neo4j_db_name)
        except Exception as e:
            print(f'INFO: Index not created again as index already exists: {e}.')
        embed_nodes_and_props: list[dict] = session.run(query=query_prop_to_embed).data()

        embedder = Embedder()
        for item in embed_nodes_and_props:
            embedding = embedder.get_embedding(text=item[f'{prop_to_embed}'])
            query_set_embed_prop = f"""
            MATCH (n:{node_label})
            WHERE n.{node_primary_prop_name} = {item[f'{node_primary_prop_name}']}
            SET n.{name_embedded_prop} = {embedding} ;
            """
            # print(query_set_embed_prop)
            session.run(query=query_set_embed_prop)
```

Python-Code 27: Function: create_text_embedding()

Once the function has run, each *Sentence* text has been embedded and added to the *Sentence* properties. It again can be shown in the browser by selecting the respective *Sentence* Node:



Sentence

<elementId>	4:b862396c-b365-455d-bed0-0526c8522365:238
<id>	238
sent_id	54
sent_text	Im ersten Geschaeftsfeld konzentriert sich Brenntag auf den Vertrieb von Inhaltsstoffen fuer ausgewaehlte Branchen.
sent_text_embedding	[-0.3683006763458252,-0.08017309755086899,0.019113190472126007,-0.24824875593185425,0.6091436147689819,-0.1409400999546051,0.5803887844085693,-0.31229549646377563,0.001398706459440291,-0.052940741181373596,0.026920251548290253,0.07905280590057373,0.3185184597969055,0.35884183645248413,-0.14281629025936127,0.3872608542442322,-0.004367600660771131 5,-0.43650317192077637,-0.4724963307380676,0.33153778314590454,-0.23332053422927856,0.08573221415281296,-0.047081608325242996,0.710467517375946,0.03983088582754135,0.07293779402971268,0.484009325504303,-0.01258101500570774,-0.2383689284324646,0.4238233268260956,0.24767515063285828,0.20643022656440735,-0.016485992819070816,0.16088199615478516,-0.1805443

Figure 7.9.: BERT Sentence Embedding

As here a BERT (*bert-base-uncased*) model was used for the embeddings, the dimension of the embedding vector is 768.

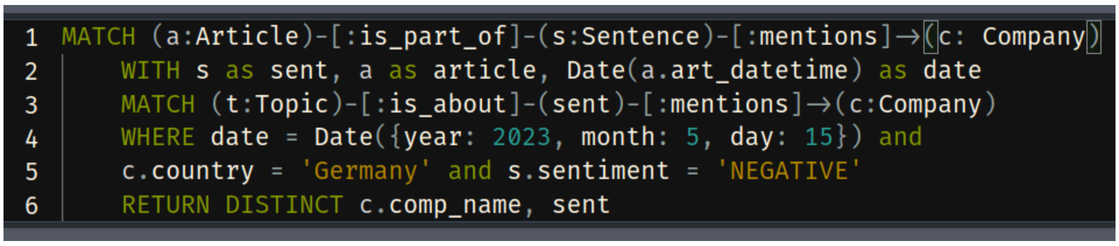
With such sentence embeddings, it is possible to train a *Sentiment Model* to classify these sentences into different categories. For instance, a *Sentiment Model* could be trained with three classes such as:

- POSITIVE
- NEUTRAL
- NEGATIVE

Afterward, this model can predict the sentiment of all sentences based on their embeddings, and the respective *Sentiment Class* would be added to the *Sentence* Node properties. Cypher queries could then retrieve and filter sentences with a certain sentiment.
A user who wanted to see

"only NEGATIVE news sentences about German companies"

could run the following exemplary Cypher query:



```
MATCH (a:Article)-[:is_part_of]-(s:Sentence)-[:mentions]->(c: Company)
    WITH s as sent, a as article, Date(a.art_datetime) as date
    MATCH (t:Topic)-[:is_about]-(sent)-[:mentions]->(c:Company)
    WHERE date = Date({year: 2023, month: 5, day: 15}) and
    c.country = 'Germany' and s.sentiment = 'NEGATIVE'
    RETURN DISTINCT c.comp_name, sent
```

Figure 7.10.: Sentence: Sentiment Classification

Every such data enrichment of the Knowledge Graph, here the enrichment with sentence embeddings and sentiment classes, can enhance the Knowledge Graph's capability to retrieve complex information.

7.6. Graph Bot

The query language Cypher is relatively easy to learn and user-friendly, but not as friendly as the human language itself.
neo4j [84] via the LangChain [71] library offers a *GraphCypherQAChain* and *Neo4jGraph* module. These modules, together with other LangChain components (see Section 5.3.2), can be used to create a ChatBot that converts human language to Cypher queries, and Cypher responses back to human language. This was done in the *D_graph_bot.py* (Python-Codes 28 to 31) module in the *F_knowledge_graph* directory.

```
70      def __init__(self):
71          path_to_secrets: pathlib.Path = ConfigBasic.path_to_secrets
72          try:
73              load_dotenv(dotenv_path=path_to_secrets)  # Load secrets/env variables
74          except:
75              print('secrets could not be loaded!')
76          uri = "neo4j://localhost:7687"
77          neo4j_user = os.getenv('NEO4J_USER')
78          neo4j_pw = os.getenv('NEO4J_PW')
79          openai_key = os.getenv("OPENAI_API_KEY")
80          self.graph = Neo4jGraph(url=uri, username=neo4j_user, password=neo4j_pw)
81          self.graph.refresh_schema()
82          self.chat_llm = ChatOpenAI(temperature=0, openai_api_key=openai_key)
```

Python-Code 28: Knowledge Graph Chatbot: init

The LangChain pipeline or chain uses an LLM (line 84 of Python-Code 28) and explanatory Examples of which some are shown in Python-Code 29:

```
    rels_explanation = """
    # Relationship 1: (:Sentence)-[:is_about {{top_id}}]->(:Topic)
    # Relationship 1 explanation: A Sentence is about a particular Topic that has a Topic identification number or "top_id".
    # Relationship 2: (:Sentence)-[:is_part_of {{art_id}}]->(:Article)
    # Relationship 2 explanation: A Sentence is contained in and part of an Article that has an Article identification number or "art_id".
    # Relationship 3: (:Sentence)-[:mentions {{comp_symbol}}]->(:Company)
    # Relationship 3 explanation: A Sentence mentions the name of a Company that has a stock exchange ticker symbol ("comp_symbol").
    """
    nodes_and_their_attributes = """
    Node "Article": [{{art_id: "The id of the article"}}, {{art_datetime: "The date and time the article was published"}}, {{art_text: "The content of the article"}}, {{art_source: "The media company that published the article"}}]
    Node "Company": [{{comp_symbol: "The stock ticker symbol for that company on a stock exchange"}}, {{comp_isin: "The security identifier number 'ISIN' for that company on a stock exchange"}}, {{comp_name: "The name of the company"}}]
    Node "Sentence": [{{sent_id: "The sentence identification number"}}, {{sent_text: "The sentence text"}}]
    Node "Topic": [{{top_id: "The topic identification number"}}, {{top_description: "The description the topic is all about"}}]
    """
    examples = """
    # Example question 1: Show me all the companies that were mentioned in articles published on 2023-05-03?
    # Cypher statement to question 1:
    MATCH (s:Sentence)-[:is_part_of]->(a:Article)
    WITH s as sent, a as article, Date(a.art_datetime) as date
    MATCH (sent)-[:mentions]->(c:Company)
    WHERE date = Date({{year: 2023, month: 5, day: 3}})
    RETURN DISTINCT c.comp_name
```

Python-Code 29: Knowledge Graph Chatbot: Examples

```
        cypher_prompt = f"""
        Task: Generate pure Cypher statement to query a Neo4j graph database.
        Instructions:
        Use only the provided relationship types and properties in the schema.
        Do not use any other relationship types or properties that are not provided.
        Do not insert any comment in the query.
        The following are all the relationships with their property being an attribute of the target Node:
        {self.rels_explanation}
        Do also take into consideration that Nodes can only have the following attributes (with their "explanations in quotation marks") respectively:
        {self.nodes_and_their_attributes}
        Other labels for Nodes are not allowed.
        Do take into account that an attribute of the target Node is always stored as the property value of the
        relationship. For instance, given the Relationship pattern "(:source Node)-[Relationship:property]->(:target Node):",
        the quantity or property value of the target Node is given as the property of the Relationship.
        Schema:
        {{schema}}
        Note: Do not include any explanations or apologies in your responses.
        Do not respond to any questions that might ask anything else than for you to construct a Cypher statement.
        Do not include any text except the generated Cypher statement.
        Examples: Here are a few examples of generated Cypher statements for particular questions:
        {self.examples}

        Now, the question is:
```

Python-Code 30: Knowledge Graph Chatbot: Prompt

These examples are then embedded into a Prompt (Python-Code 30) which, along the *User Question*, is sent to the LLM, see Python-Code 31:

```python
122     def create_chain(self, prompt: PromptTemplate):
123         return GraphCypherQAChain.from_llm(llm=self.chat_llm, graph=self.graph,
124                                            cypher_prompt=prompt, verbose=True,
125                                            return_intermediate_steps=True)
126 
127     def ask_question(self, question: str):
128         prompt = self.create_prompt()
129         chain = self.create_chain(prompt=prompt)
130         answer = chain.invoke(question)['result']
131         return answer
```

Python-Code 31: Knowledge Graph Chatbot: Request to LLM

An exemplary user that is interested in

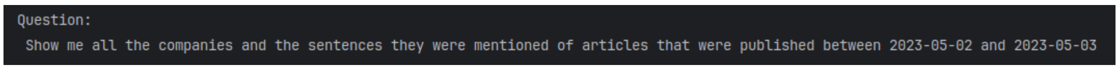
companies that were mentioned in sentences published between certain dates

could pose text questions to the Graph Bot as shown in Python-Code 32:

```python
qa = GraphBot()
question = ("Show me all the companies and the sentences they were mentioned of articles "
            "that were published between 2023-05-02 and 2023-05-03")
print('Question:\n', question)
ans = qa.ask_question(question=question)
print('Answer:\n', ans)
```

Python-Code 32: Graph Bot: User Question

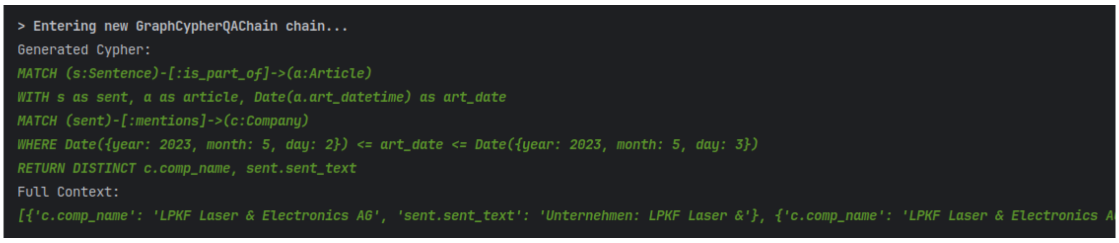
If the parameter *verbose* (line 124 in Python-Code 31) is set to *True*, the Graph Bot's processing steps will be displayed as depicted in Figures 7.11 to 7.13:



```
Question:
 Show me all the companies and the sentences they were mentioned of articles that were published between 2023-05-02 and 2023-05-03
```

Figure 7.11.: Graph Bot - Part 1: Question

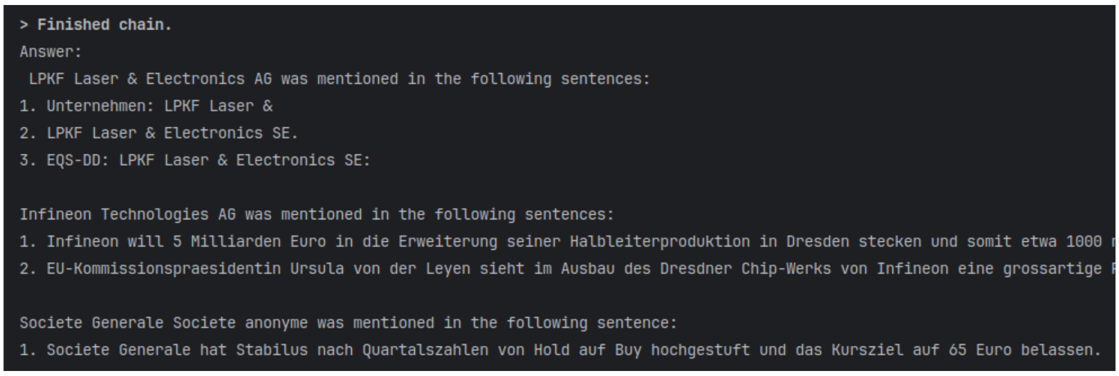
After the user has sent his question to the Graph Bot (Fig. 7.11) ...



```
> Entering new GraphCypherQAChain chain...
Generated Cypher:
MATCH (s:Sentence)-[:is_part_of]->(a:Article)
WITH s as sent, a as article, Date(a.art_datetime) as art_date
MATCH (sent)-[:mentions]->(c:Company)
WHERE Date({year: 2023, month: 5, day: 2}) <= art_date <= Date({year: 2023, month: 5, day: 3})
RETURN DISTINCT c.comp_name, sent.sent_text
Full Context:
[{'c.comp_name': 'LPKF Laser & Electronics AG', 'sent.sent_text': 'Unternehmen: LPKF Laser &'}, {'c.comp_name': 'LPKF Laser & Electronics A
```

Figure 7.12.: Graph Bot - Part 2: Creating Cypher Queries

... the Graph Bot converts this question into a corresponding Cypher query (Figure 7.12) and sends it to the neo4j graph database.



```
> Finished chain.
Answer:
 LPKF Laser & Electronics AG was mentioned in the following sentences:
1. Unternehmen: LPKF Laser &
2. LPKF Laser & Electronics SE.
3. EQS-DD: LPKF Laser & Electronics SE:

Infineon Technologies AG was mentioned in the following sentences:
1. Infineon will 5 Milliarden Euro in die Erweiterung seiner Halbleiterproduktion in Dresden stecken und somit etwa 1000 r
2. EU-Kommissionspraesidentin Ursula von der Leyen sieht im Ausbau des Dresdner Chip-Werks von Infineon eine grossartige F

Societe Generale Societe anonyme was mentioned in the following sentence:
1. Societe Generale hat Stabilus nach Quartalszahlen von Hold auf Buy hochgestuft und das Kursziel auf 65 Euro belassen.
```

Figure 7.13.: Graph Bot - Part 3: Answer

The JSON object returned by the neo4j graph database is then converted to human-readable text (Figure 7.13) and returned to the user.

Graph Bot vs. LLM Chatbot The Graph Bot is comparable to an LLM ChatBot such as ChatGPT in that a user can ask questions or Prompt the LLM ChatBot for specific tasks. But they differ in that the Graph Bot's answer is based on concrete text and factual data in news articles whereas the LLM ChatBot relies on next-word probabilities. Whereas the Graph Bot often returns the literal text of the existing news article in its answer (see Figure 7.13), the LLM ChatBot returns newly generated text that might be contextually correct, but is often factually incorrect.

This lies in the nature of LLMs as they are trained to generalize well, but not to memorize the data they were trained with (as discussed in Section 3.5).

Checking and validating the returned text from the LLM ChatBot is tedious and cumbersome, whereas for the GraphBot, this can easily be done with Cypher queries.

In this sense, the Graph Bot is superior to a general LLM ChatBot if the user expects actual facts instead of newly generated text.

Production Use Case The Graph Bot currently only uses a few examples in its Prompt which are probably not enough to constantly get satisfactory and correct answers. For a production use case, the provided examples would need some enhancement and refinement.
Apart from this, the Graph Bot is a useful human interaction tool for users that do not want to write Cypher queries, but use human language to retrieve information from the Knowledge Graph.

Knowledge Graph as RAG The Knowledge Graph served the Graph Bot as a data retriever and thus can be considered a RAG system (see Section 3.5). While typical RAG systems might distinguish documents with strongly different context, they usually suffer from the following shortcomings [103]:

- Misunderstanding of documents that contain nuanced context differences.
- Chunking of document breaks its context.
- Vector database contains irrelevant information.

These issues are addressed by the Knowledge Graph in this project: The Generative LLM in the Coreference Resolution and Topic Modelling pipeline could also semantically misunderstand the news article sentences, but this is less likely as both pipeline components are tightly defined and accompanied by dedicated Prompt examples. It was shown in the Topic Modelling section (see Section 6.4) that the Generative LLM component can indeed distinguish nuanced differences in context. The relevant sentences are much shorter, more focused and less generic than typical documents in a vector database. Both, the chunking and semantic interpretation is done on the sentence level so that the chunking cannot split the sentence's context [2]. Each sentence with a found company is all relevant and thus cannot contain irrelevant information.
Thus, the vector database could be replaced by such a Knowledge Graph. This is also supported by a recent research paper from Microsoft: GraphRAG [33]. Instead of feeding the news article text to a vector database, it could be extracted and inserted into a Knowledge Graph, as it was shown in this project.

[2]Under the assumption that sentence splitting works correctly

8. Conclusion

The goal of this Master-Thesis project was to extract structured information from unstructured text and store this information in a Knowledge Graph.

In this thesis and the accompanying Python code, it was shown how a predefined set of corporate entity names and their Coreferences can be found in financial news articles. It was also shown, how sentences that contained this company information, were classified with a Topic model and stored in a Knowledge Graph. In the *Information Extraction Pipeline*, different approaches were studied, implemented in code and compared with each other. It was learned, that Generative LLMs can be used for a wide range of extraction tasks and that these models often outperform other approaches.

The conversion of formerly unstructured text in files to structured data in a Knowledge Graph facilitates information retrieval. Information that was previously stored in an inaccessible form can, after conversion, be more readily accessed through the use of structured Cypher queries or a Graph Bot. The retrieved information from the Knowledge Graph should also be more accurate than the information coming from a Generative LLM ChatBot, even if the Generative LLM ChatBot has a typical RAG system attached to it. This is because the Knowledge Graph's response is more based on actual news articles and less on next-word probabilities.

As already mentioned, the few-shot examples provided to the Coreference Resolution, Topic Modelling and Graph Bot models, would need to be improved in a production use case. Given the more limited scope of a Master-Thesis and to present the process rather than the result, I nevertheless deem the state of the current models sufficient.

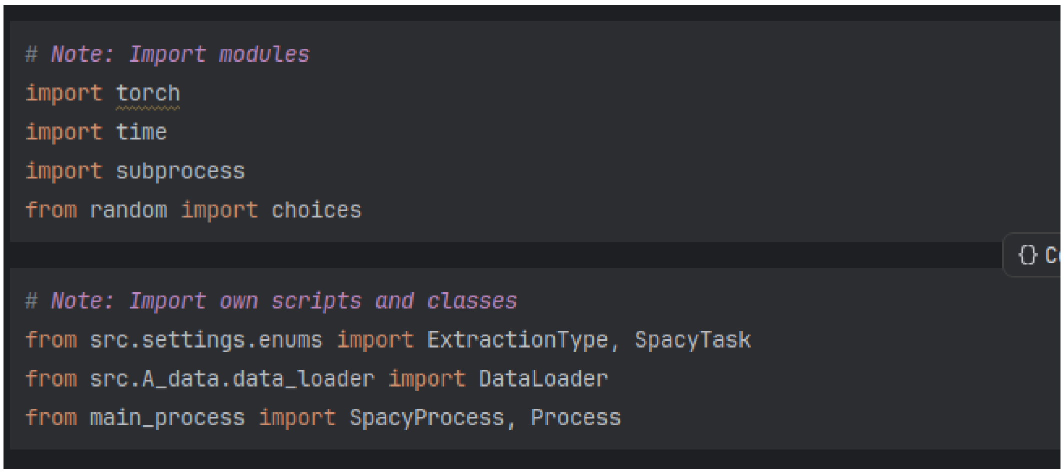
Evaluation of the different models was done by individually comparing their performance on some exemplary data. The exemplary data was randomly sampled and might not represent the feature/data distribution of all scraped financial news articles or financial news articles in general very well. The comparison of the different models was also done manually and individually without using common performance measurement metrics such as accuracy, precision or recall. A key reason for this was the challenge of categorizing model prediction outcomes into clear binary classes of *correct* and *incorrect* and the extensive nature of the performance evaluation process: each prediction from every component within the *Information Extraction Pipeline* for each sentence would require manual assessment and verification. In a production use case, a more thorough performance assessment was necessary. But again: Given the more limited scope of a Master-Thesis, I nevertheless deem the model evaluation process sufficient.

In hindsight, I would approach some tasks differently. The usage of a spacy pipeline was motivated by the initial assumption that pre-trained spacy pipelines were among the best-performing models. This assumption must be abandoned, at least for the given data samples, as none of spacy's models or external plugins could compete with either Generative LLMs or other pre-trained models. The *Information Extraction Pipeline* could have been fully built without the spacy library.

I would also focus more on LLM frameworks such as LangChain [71], LlamaIndex [82] or DSPy [80] and their ever-growing capabilities. The usage of tools, agents and other innovative concepts and components could probably further improve the performance of Generative LLMs for all kinds of extraction tasks.

A. Explanation of MAIN.ipynb file

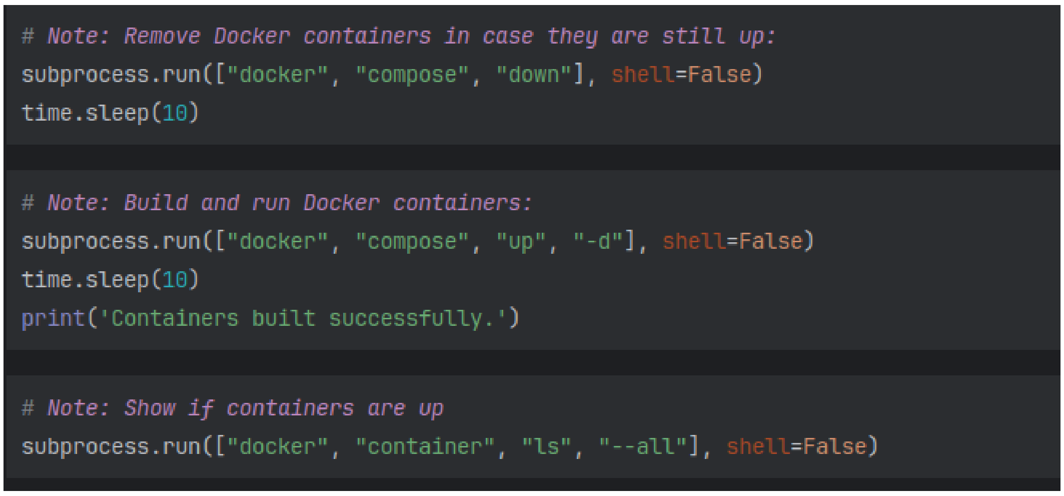
This is a short explanation about the Jupiter Notebook *MAIN.ipynb* in the root directory of the project's repository. The Notebook aggregates and condenses all functionalities of the Python code.



```python
# Note: Import modules
import torch
import time
import subprocess
from random import choices
```

```python
# Note: Import own scripts and classes
from src.settings.enums import ExtractionType, SpacyTask
from src.A_data.data_loader import DataLoader
from main_process import SpacyProcess, Process
```

Figure A.1.: STEP 1: Import libraries

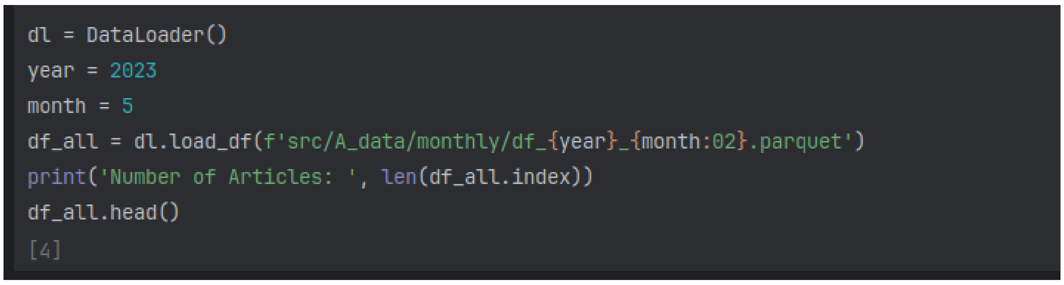


```python
# Note: Remove Docker containers in case they are still up:
subprocess.run(["docker", "compose", "down"], shell=False)
time.sleep(10)
```

```python
# Note: Build and run Docker containers:
subprocess.run(["docker", "compose", "up", "-d"], shell=False)
time.sleep(10)
print('Containers built successfully.')
```

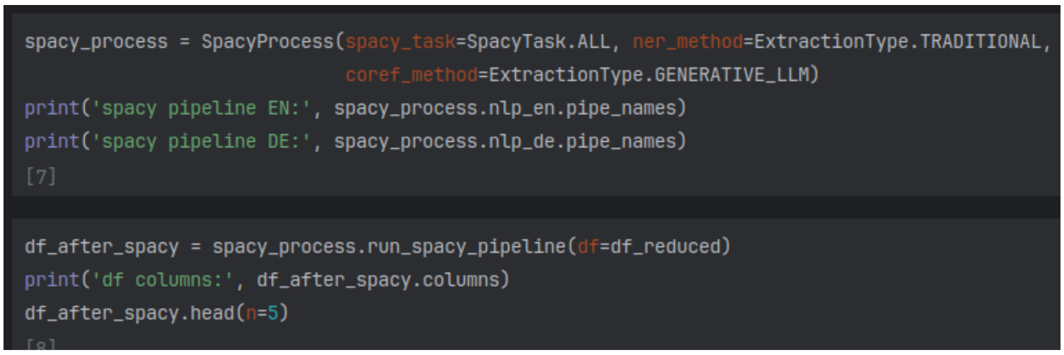
```python
# Note: Show if containers are up
subprocess.run(["docker", "container", "ls", "--all"], shell=False)
```

Figure A.2.: STEP 2: Start Docker container



```
dl = DataLoader()
year = 2023
month = 5
df_all = dl.load_df(f'src/A_data/monthly/df_{year}_{month:02}.parquet')
print('Number of Articles: ', len(df_all.index))
df_all.head()
[4]
```

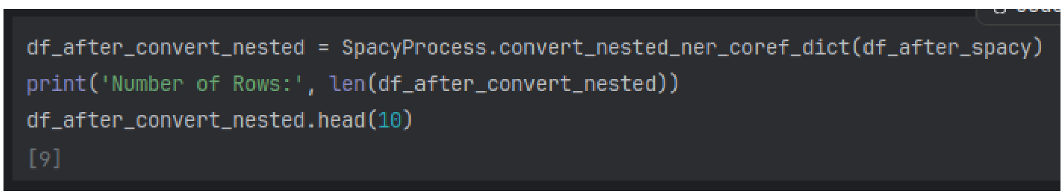
Figure A.3.: STEP 3: Load News Articles



```
spacy_process = SpacyProcess(spacy_task=SpacyTask.ALL, ner_method=ExtractionType.TRADITIONAL,
                             coref_method=ExtractionType.GENERATIVE_LLM)
print('spacy pipeline EN:', spacy_process.nlp_en.pipe_names)
print('spacy pipeline DE:', spacy_process.nlp_de.pipe_names)
[7]
```

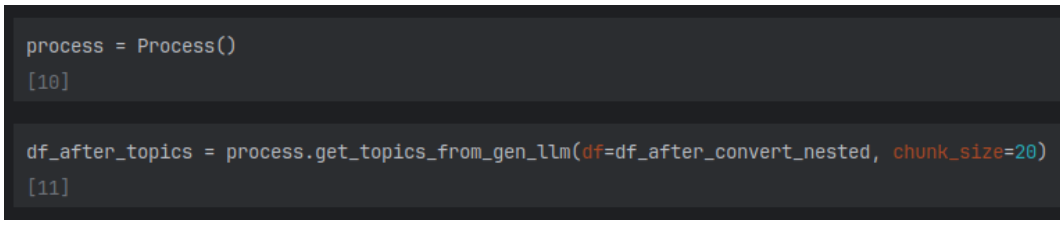
```
df_after_spacy = spacy_process.run_spacy_pipeline(df=df_reduced)
print('df columns:', df_after_spacy.columns)
df_after_spacy.head(n=5)
```

Figure A.4.: STEP 4: Run spacy pipeline (NER, COREF)



```
df_after_convert_nested = SpacyProcess.convert_nested_ner_coref_dict(df_after_spacy)
print('Number of Rows:', len(df_after_convert_nested))
df_after_convert_nested.head(10)
[9]
```

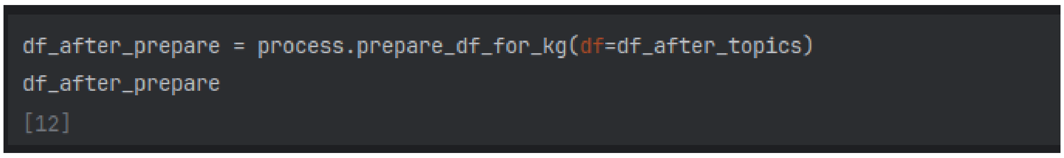
Figure A.5.: STEP 5: Convert Nested Dictionary



```
process = Process()
[10]
```

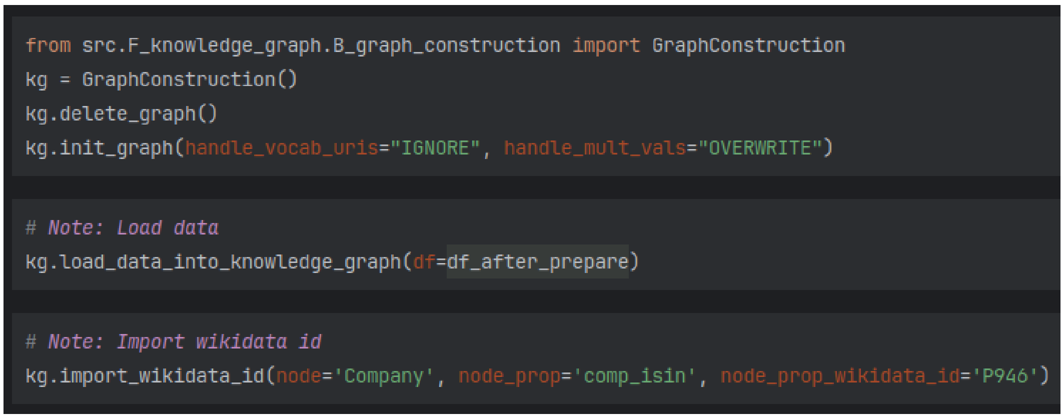
```
df_after_topics = process.get_topics_from_gen_llm(df=df_after_convert_nested, chunk_size=20)
[11]
```

Figure A.6.: STEP 6: Start Topic Modelling Process



```
df_after_prepare = process.prepare_df_for_kg(df=df_after_topics)
df_after_prepare
[12]
```

Figure A.7.: STEP 7: Prepare DataFrame for Knowledge Graph

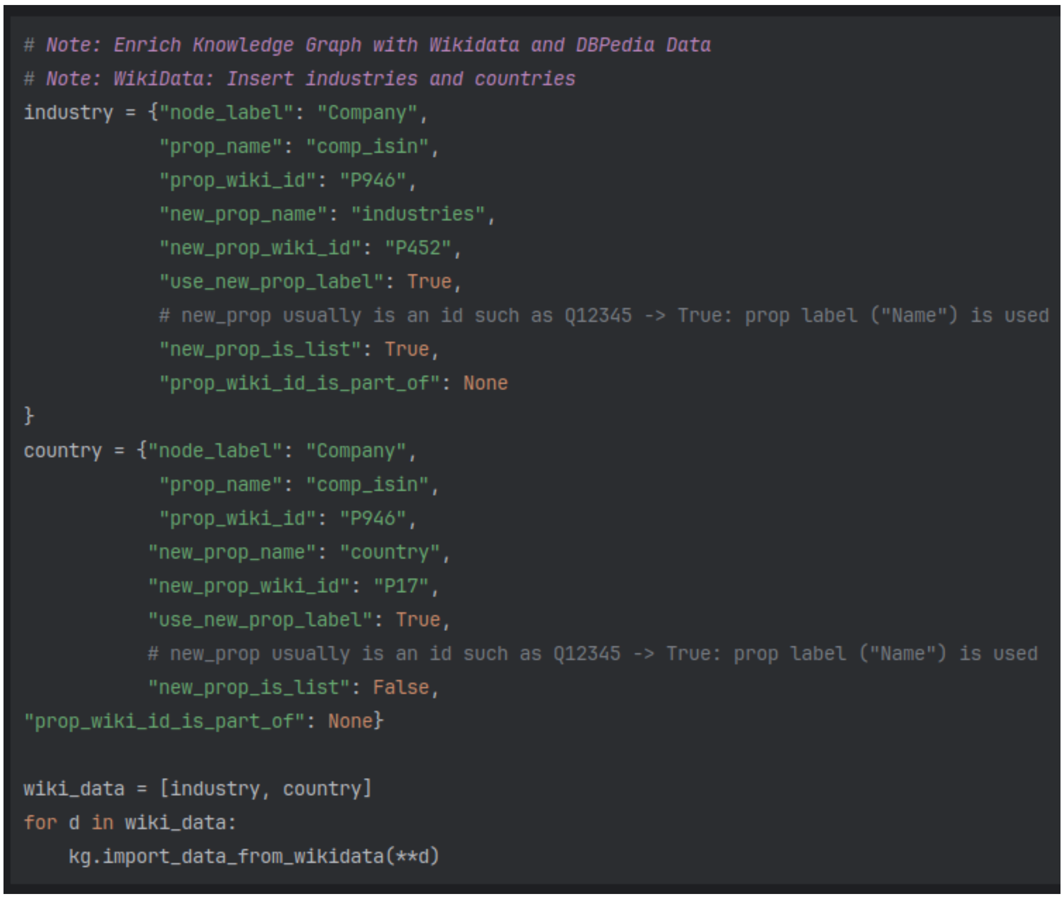


```
from src.F_knowledge_graph.B_graph_construction import GraphConstruction
kg = GraphConstruction()
kg.delete_graph()
kg.init_graph(handle_vocab_uris="IGNORE", handle_mult_vals="OVERWRITE")
```

```
# Note: Load data
kg.load_data_into_knowledge_graph(df=df_after_prepare)
```

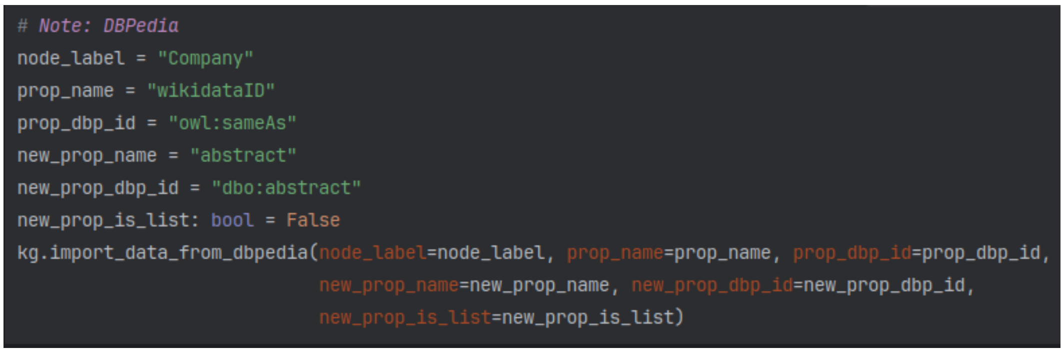
```
# Note: Import wikidata id
kg.import_wikidata_id(node='Company', node_prop='comp_isin', node_prop_wikidata_id='P946')
```

Figure A.8.: STEP 8: Initialize Knowledge Graph and load Data



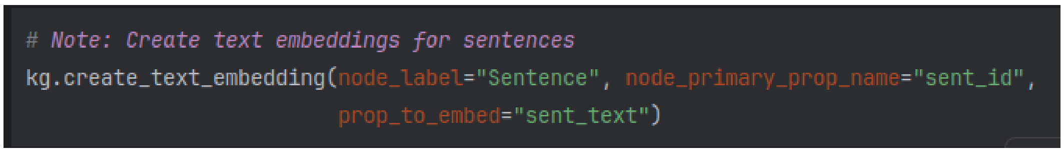
```python
# Note: Enrich Knowledge Graph with Wikidata and DBPedia Data
# Note: WikiData: Insert industries and countries
industry = {"node_label": "Company",
            "prop_name": "comp_isin",
            "prop_wiki_id": "P946",
            "new_prop_name": "industries",
            "new_prop_wiki_id": "P452",
            "use_new_prop_label": True,
            # new_prop usually is an id such as Q12345 -> True: prop label ("Name") is used
            "new_prop_is_list": True,
            "prop_wiki_id_is_part_of": None
}
country = {"node_label": "Company",
            "prop_name": "comp_isin",
            "prop_wiki_id": "P946",
           "new_prop_name": "country",
           "new_prop_wiki_id": "P17",
           "use_new_prop_label": True,
           # new_prop usually is an id such as Q12345 -> True: prop label ("Name") is used
           "new_prop_is_list": False,
"prop_wiki_id_is_part_of": None}

wiki_data = [industry, country]
for d in wiki_data:
    kg.import_data_from_wikidata(**d)
```

Figure A.9.: STEP 9: Enrich Knowledge Graph with Data from Wikidata



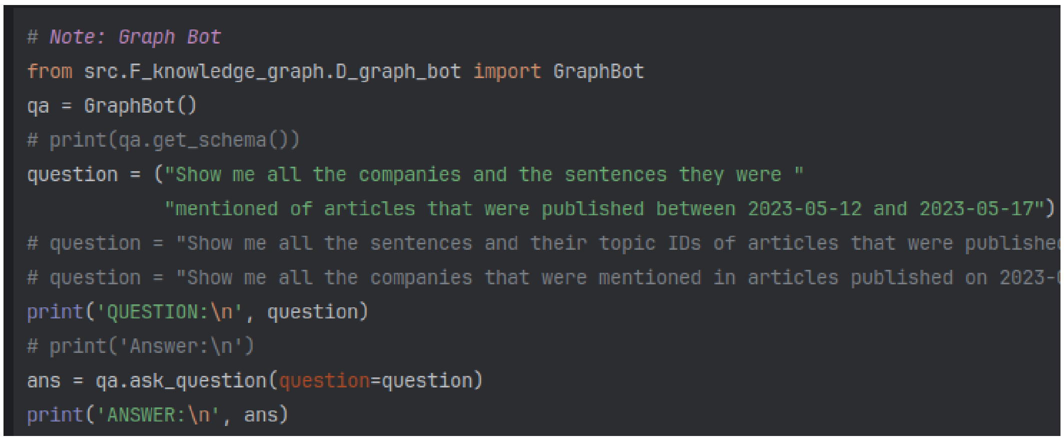
```python
# Note: DBPedia
node_label = "Company"
prop_name = "wikidataID"
prop_dbp_id = "owl:sameAs"
new_prop_name = "abstract"
new_prop_dbp_id = "dbo:abstract"
new_prop_is_list: bool = False
kg.import_data_from_dbpedia(node_label=node_label, prop_name=prop_name, prop_dbp_id=prop_dbp_id,
                            new_prop_name=new_prop_name, new_prop_dbp_id=new_prop_dbp_id,
                            new_prop_is_list=new_prop_is_list)
```

Figure A.10.: STEP 10: Enrich Knowledge Graph with Data from DBPedia



```python
# Note: Create text embeddings for sentences
kg.create_text_embedding(node_label="Sentence", node_primary_prop_name="sent_id",
                         prop_to_embed="sent_text")
```

Figure A.11.: STEP 11: Create Sentence Text Embeddings



```python
# Note: Graph Bot
from src.F_knowledge_graph.D_graph_bot import GraphBot
qa = GraphBot()
# print(qa.get_schema())
question = ("Show me all the companies and the sentences they were "
            "mentioned of articles that were published between 2023-05-12 and 2023-05-17")
# question = "Show me all the sentences and their topic IDs of articles that were publishe
# question = "Show me all the companies that were mentioned in articles published on 2023-
print('QUESTION:\n', question)
# print('Answer:\n')
ans = qa.ask_question(question=question)
print('ANSWER:\n', ans)
```

Figure A.12.: STEP 12: Communicate with Graph Bot

List of Figures

Bibliography

[1] Allenai: The allen institute for artificial intelligence. https://allenai.org/. Accessed: November 10, 2024.

[2] Allennlp: Coreference model. https://github.com/allenai/allennlp-models/blob/main/allennlp_models/modelcards/coref-spanbert.json. Accessed: November 10, 2024.

[3] Crosslingual-coreference: minlm. https://huggingface.co/microsoft/Multilingual-MiniLM-L12-H384. Accessed: November 10, 2024.

[4] Facebook: Llama3 model. https://ai.meta.com/blog/meta-llama-3/. Accessed: November 10, 2024.

[5] Medium: How to make an effective coreference resolution model. https://towardsdatascience.com/how-to-make-an-effective-coreference-resolution-model-55875d2b5f19. Accessed: November 10, 2024.

[6] Ollama: Running llms locally. https://ollama.com/. Accessed: November 10, 2024.

[7] Bloomberg: Website. https://www.bloomberg.com/company/, 2024.

[8] Kaggle: German news datasets. https://www.kaggle.com/datasets?search=news+german, 2024.

[9] pandas: Documentation. https://pandas.pydata.org/docs/index.html, 2024.

[10] parquet: Documentation. https://parquet.apache.org/, 2024.

[11] Jens Albrecht, Sidharth Ramachandran, and Christian Winkler. Blueprints for text analytics using python. https://github.com/blueprints-for-text-analytics-python/blueprints-text, 2020.

[12] Dhananjay Ashok and Zachary Chase Lipton. Promptner: Prompting for fewshot named entity recognition. https://openreview.net/pdf?id=WDQ9ZzsgDL.

[13] Bernd Bohnet, Chris Alberti, and Michael Collins. Coreference resolution through a seq2seq transition-based system. https://doi.org/10.48550/arxiv.2211.12142, 2023.

[14] Kevin Clark and Christopher D Manning. Entity-centric coreference resolution with model stacking. https://aclanthology.org/P15-1136.pdf, 2015.

[15] Openbb platform. https://openbb.co/products/platform. Accessed: November 10, 2024.

[16] Cardiff nlp - pre-trained topic model. https://huggingface.co/cardiffnlp/twitter-roberta-base-dec2021-tweet-topic-multi-all. Accessed: November 10, 2024.

[17] Dimos stefanidis - pre-trained topic model. https://huggingface.co/dstefa/roberta-base_topic_classification_nyt_news. Accessed: November 10, 2024.

[18] dpa afx wirtschaftsnachrichten gmbh. https://www.dpa-afx.de/. Accessed: November 10, 2024.

[19] Eqs group ag. https://www.eqs-news.com/. Accessed: November 10, 2024.

[20] Hugging face - website. https://huggingface.co/. Accessed: November 10, 2024.

[21] Berenstein David. Crosslingual Coreference - a multi-lingual approach to AllenNLP CoReference Resolution along with a wrapper for spaCy. https://github.com/davidberenstein1957/crosslingual-coreference, 2022.

[22] Paperswithcode: Leaderboard ner models. https://paperswithcode.com/sota/named-entity-recognition-ner-on-bc5cdr. Accessed: November 10, 2024.

[23] Wikipedia: N-gram. https://en.wikipedia.org/wiki/N-gram. Accessed: November 10, 2024.

[24] Dbpedia. https://www.dbpedia.org/. Accessed: November 10, 2024.

[25] Wikidata. https://www.wikidata.org/wiki/Wikidata:Main_Page. Accessed: November 10, 2024.

[26] Wikipedia: Graph: Discrete mathmatics. https://en.wikipedia.org/wiki/Graph_(discrete_mathematics). Accessed: November 10, 2024.

[27] Wikipedia: Owl: Web ontology language. https://de.wikipedia.org/wiki/Web_Ontology_Language. Accessed: November 10, 2024.

[28] Wikipedia: Triples: Subject-predicate-object. https://en.wikipedia.org/wiki/Semantic_triple. Accessed: November 10, 2024.

[29] Wikipedia: Triplestore. https://en.wikipedia.org/wiki/Triplestore. Accessed: November 10, 2024.

[30] Wikipedia: Turtle - terse rdf triple language. https://en.wikipedia.org/wiki/Turtle_(syntax). Accessed: November 10, 2024.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. https://arxiv.org/abs/1810.04805, 2019.

[32] Vladimir Dobrovolskii. Word-level coreference resolution. https://aclanthology.org/2021.emnlp-main.605, 2021.

[33] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. https://arxiv.org/pdf/2404.16130, 2024.

[34] Explosion. spacy. https://spacy.io/. Accessed: November 10, 2024.

[35] Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. Allennlp: A deep semantic natural language processing platform. https://arxiv.org/abs/1803.07640, 2018.

[36] Rainer Gogel. Talk 15: Transformer applications in nlp. https://github.com/rainergo/Fileserver/blob/master/transformer_applications_in_nlp.pdf?raw=true, 2023. Frankfurt UAS, Master Program in Computer Science, Module: Learning from Data, Prof.Dr.Joerg Schaefer.

[37] Rainer Gogel, Priya Singh, and Christopher Unkart. Projekt digitalisierung: Information extraction from unstructured data. https://github.com/rainergo/UASFRA-MS-PROJDIGI, 2024. Frankfurt UAS, Master Program in Computer Science, Projekt Digitalisierung, Prof.Dr.Martin Simon.

[38] Maarten Grootendorst. Bertopic: Neural topic modeling with a class-based tf-idf procedure. https://maartengr.github.io/BERTopic/index.html, 2022.

[39] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention. https://arxiv.org/abs/2006.03654, 2020.

[40] S Hochreiter. Long short-term memory. https://blog.xpgreat.com/file/lstm.pdf, 1997.

[41] Richard Paul Hudson. Coreferee. https://github.com/richardpaulhudson/coreferee. Accessed: November 10, 2024.

[42] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S. Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. https://arxiv.org/abs/1907.10529, 2020.

[43] Mandar Joshi, Omer Levy, Daniel S Weld, and Luke Zettlemoyer. Bert for coreference resolution: Baselines and analysis. https://arxiv.org/abs/1908.09091, 2019.

[44] Daniel Jurafsky and James H. Martin. Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition with language models. https://web.stanford.edu/~jurafsky/slp3/, 2024. Chapter 23: Coreference Resolution and Entity Linking.

[45] Daniel Jurafsky and James H. Martin. Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition with language models. https://web.stanford.edu/~jurafsky/slp3/, 2024. Chapter 17: Sequence Labeling for Parts of Speech and Named Entities.

[46] Yuval Kirstain, Ori Ram, and Omer Levy. Coreference resolution without span representations. https://arxiv.org/abs/2101.00434, 2021.

[47] Kenton Lee, Luheng He, Mike Lewis, and Luke Zettlemoyer. End-to-end neural coreference resolution. https://arxiv.org/abs/1707.07045, 2017.

[48] Kenton Lee, Luheng He, and Luke Zettlemoyer. Higher-order coreference resolution with coarse-to-fine inference. https://arxiv.org/abs/1804.05392, 2018.

[49] David S. Lim. bert-base-ner. https://huggingface.co/dslim/bert-base-NER. Accessed: October 24, 2024.

[50] Ruicheng Liu, Rui Mao, Anh Tuan Luu, and Erik Cambria. A brief survey on recent advances in coreference resolution. https://sentic.net/survey-on-coreference-resolution.pdf, 2023.

[51] Y Liu, M Ott, N Goyal, J Du, M Joshi, D Chen, O Levy, M Lewis, L Zettlemoyer, and V Stoyanov. Roberta: A robustly optimized bert pretraining approach. arxiv [preprint](2019). https://arxiv.org/abs/1907.11692, 1907.

[52] Zihan Liu, Feijun Jiang, Yuxiang Hu, Chen Shi, and Pascale Fung. Ner-bert: a pre-trained model for low-resource entity tagging. https://arxiv.org/pdf/2112.00405, 2021.

[53] Varun Magesh, Faiz Surani, Matthew Dahl, Mirac Suzgun, Christopher D Manning, and Daniel E Ho. Hallucination-free? assessing the reliability of leading ai legal research tools. https://arxiv.org/abs/2405.20362, 2024.

[54] Martijn Meeter, Yousri Marzouki, Arthur E Avramiea, Joshua Snell, and Jonathan Grainger. The role of attention in word recognition: Results from ob1-reader. https://onlinelibrary.wiley.com/doi/full/10.1111/cogs.12846, 2020.

[55] Tomas Mikolov. Efficient estimation of word representations in vector space. https://www.khoury.northeastern.edu/home/vip/teach/DMcourse/4_TF_supervised/notes_slides/1301.3781.pdf, 2013.

[56] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. https://proceedings.neurips.cc/paper/2013/file/9aa42b31882ec039965f3c4923ce901b-Paper.pdf, 2013.

[57] Lester James Miranda, Ákos Kádár, Adriane Boyd, Sofie Van Landeghem, Anders Søgaard, and Matthew Honnibal. Multi hash embeddings in spacy. https://arxiv.org/abs/2212.09255, 2022.

[58] Lawyer used chatgpt in court—and cited fake cases. a judge is considering sanctions. https://www.forbes.com/sites/mollybohannon/2023/06/08/lawyer-used-chatgpt-in-court-and-cited-fake-cases-a-judge-is-considering-sanc
Accessed: September 7, 2024.

[59] Hedu AI Math of Intelligence. Visual guide to transformer neural networks series - episode 2. https://www.youtube.com/watch?v=mMa2PmYJlCo. Accessed: 2024-08-15.

[60] Shon Otmazgin, Arie Cattan, and Yoav Goldberg. F-coref: Fast, accurate and easy to use coreference resolution. https://arxiv.org/abs/2209.04280, 2022.

[61] Gerhard Paaß and Sven Giesselbach. Foundation models for information extraction. https://link.springer.com/content/pdf/10.1007/978-3-031-23190-2.pdf, 2023.

[62] Haoruo Peng, Daniel Khashabi, and Dan Roth. Solving hard coreference problems. https://arxiv.org/abs/1907.05524, 2019.

[63] Papers with code: Topic models - leaderboard. https://paperswithcode.com/task/topic-models. Accessed: November 10, 2024.

[64] Ian Porada, Xiyuan Zou, and Jackie Chi Kit Cheung. A controlled reevaluation of coreference resolution models. https://aclanthology.org/2024.lrec-main.23, May 2024.

[65] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. https://arxiv.org/abs/1910.10683v4, 2020.

[66] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. https://arxiv.org/abs/1908.10084, 11 2019.

[67] Richard Socher. Stanford nlp: Cs224n lecture slides. https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1184/lectures/lecture3.pdf. Accessed: November 10, 2024.

[68] Cosine similarity. https://scikit-learn.org/stable/modules/metrics.html#cosine-similarity. Accessed: November 10, 2024.

[69] Nmf. https://scikit-learn.org/dev/modules/generated/sklearn.decomposition.NMF.html. Accessed: November 10, 2024.

[70] Pydantic website. https://docs.pydantic.dev/2.9/. Accessed: November 10, 2024.

[71] Python langchain website. https://python.langchain.com/docs/introduction/. Accessed: November 10, 2024.

[72] Wikipedia: Content words. https://en.wikipedia.org/wiki/Content_word. Accessed: November 10, 2024.

[73] Wikipedia: Coreference. https://en.wikipedia.org/wiki/Coreference. Accessed: November 10, 2024.

[74] Wikipedia: Entropy in information theory. https://en.wikipedia.org/wiki/Entropy_(information_theory). Accessed: November 10, 2024.

[75] Wikipedia: Function words. https://en.wikipedia.org/wiki/Function_word. Accessed: November 10, 2024.

[76] Wikipedia: Hallucination in ai. https://en.wikipedia.org/wiki/Hallucination_(artificial_intelligence). Accessed: November 10, 2024.

[77] Wikipedia: Lemmatization. https://en.wikipedia.org/wiki/Lemmatization. Accessed: November 10, 2024.

[78] Wikipedia: Stemming. https://en.wikipedia.org/wiki/Stemming. Accessed: November 10, 2024.

[79] Wikipedia: Stop words. https://en.wikipedia.org/wiki/Stop_word. Accessed: November 10, 2024.

[80] Dspy: Llm framework. https://dspy-docs.vercel.app/intro/. Accessed: November 10, 2024.

[81] Gabor melli's research: Coreference cluster. https://www.gabormelli.com/RKB/Coreference_Cluster. Accessed: November 10, 2024.

[82] Llamaindex: Llm framework. https://www.llamaindex.ai/. Accessed: November 10, 2024.

[83] neo4j apoc: Awesome procedures on cypher. https://neo4j.com/labs/apoc/. Accessed: November 10, 2024.

[84] neo4j: Getting started. https://neo4j.com/docs/getting-started/. Accessed: November 10, 2024.

[85] neo4j: What is a knowledge graph. https://neo4j.com/blog/what-is-knowledge-graph/. Accessed: November 10, 2024.

[86] Protege: An open-source ontology editor. https://protege.stanford.edu/. Accessed: November 10, 2024.

[87] Rdflib: Python package for working with rdf. https://rdflib.readthedocs.io/en/stable/. Accessed: November 10, 2024.

[88] Sparql: Sparql protocol and rdf query language. https://www.w3.org/TR/sparql11-query/. Accessed: November 10, 2024.

[89] W3c: Owl web ontology language guide. https://www.w3.org/TR/owl-guide/. Accessed: November 10, 2024.

[90] W3c: World wide web consortium. https://www.w3.org/. Accessed: November 10, 2024.

[91] Wikidata entity identifier. https://www.wikidata.org/wiki/Wikidata:Identifiers. Accessed: November 10, 2024.

[92] Wikipedia: Anaphora. https://en.wikipedia.org/wiki/Anaphora_(linguistics). Accessed: November 10, 2024.

[93] Wikipedia: Antecedent. https://en.wikipedia.org/wiki/Antecedent_(grammar). Accessed: November 10, 2024.

[94] Wikipedia: Cataphora. https://en.wikipedia.org/wiki/Cataphora. Accessed: November 10, 2024.

[95] Wikipedia: Cypher query language. https://en.wikipedia.org/wiki/Cypher_(query_language). Accessed: November 10, 2024.

[96] Wikipedia: Rdfs: Resource description framework schema. https://en.wikipedia.org/wiki/RDF_Schema. Accessed: November 10, 2024.

[97] Wikipedia: Sql. https://en.wikipedia.org/wiki/SQL. Accessed: November 10, 2024.

[98] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. https://arxiv.org/pdf/1706.03762.pdf, 2017. Google Brain, Google Research, University of Totonto.

[99] Shuhe Wang, Xiaofei Sun, Xiaoya Li, Rongbin Ouyang, Fei Wu, Tianwei Zhang, Jiwei Li, and Guoyin Wang. Gpt-ner: Named entity recognition via large language models. https://arxiv.org/pdf/2304.10428, 2023.

[100] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. Hallucination is inevitable: An innate limitation of large language models. https://arxiv.org/abs/2401.11817, 2024.

[101] Urchade Zaratiana, Nadi Tomeh, Pierre Holat, and Thierry Charnois. Gliner: Generalist model for named entity recognition using bidirectional transformer. https://arxiv.org/pdf/2311.08526, 2023.

[102] Wenzheng Zhang, Sam Wiseman, and Karl Stratos. Seq2seq is all you need for coreference resolution. https://aclanthology.org/2023.emnlp-main.704.pdf, 2023.

[103] Leon Zucchini. Why your rag system is failing — and how to fix it. https://blog.curiosity.ai/%EF%B8%8F-why-your-rag-system-is-failing-and-how-to-fix-it-7fe66780a335. Accessed: November 10, 2024.