Universiteit Leiden
The Netherlands

Data Science and Artificial Intelligence

Generating metadata for UML class models from natural language requirements texts using NLP

B.M. den Boef, s2829452

Supervisors:
Dr. G.J Ramackers and Dr. S. Verberne

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

19/07/2023

Abstract

In order to capture and define the requirements of software systems properly, UML class models are frequently utilised. However, these models quickly expand in size and complexity, which inhibits the understandability of these models for non-experts. In addition, the process of creating these models using existing modelling tools is a cumbersome manual process. In an effort to overcome these issues, a transformer-based model has been implemented in this thesis to automate this process, such that metadata for UML class models can be automatically constructed from natural language requirements texts. A BERT model is used to perform Named Entity Recognition (NER), in combination with a rule-based relationship extraction model to perform Relation Extraction (RE). Using a dataset consisting of 9 documents, the BERT model manages to reach a weighted average F1-score of 56.8%. Furthermore, the rule-based RE model is capable of extracting nearly half of the relations. The latter is primarily due to the influence of the NER model, since incorrect label predictions result in missed relations or wrong predictions. However, even though the results do not outperform the current state-of-the-art, they provide a solid basis for further research, as well as proving the potential of a transformer-based approach to automate the generation of UML class models. This research is part of the LIACS Prose to Prototype / ngUML research and development tool project.

Contents

1 Introduction

Capturing requirements in an accurate way is essential for software development. Often this is a manual process where business analysts interview customers or end-users and subsequently create UML models. These models can grow quickly for large software systems [1], causing an increase in complexity that impacts the understandability time and efficiency [2]. As a result, interactions with these models become harder for non-experts. This lowers the usefulness of the tool, as well as limiting its usage to domain experts. Moreover, the process of creating a UML model can be a time-consuming process, especially if you are not already an expert [3], which adds another barrier.

UML models can be very useful in the software development process, especially during the transition from requirements listing to development, which is thought to be one of the most complicated tasks in software development [4]. However, they can be time-consuming to produce and can grow quickly in complexity [1, 2]. Therefore, a tool that can help design UML models semi-automatically is desired. The model has to be able to convert natural language requirements texts into metadata for UML class models. Specifically, the model needs to be able to do Named Entity Recognition (NER), which is the recognizing of entities (Class, Attribute, Association...) in the text, as well as Relation Extraction (RE), which consists of identifying relations between entities and labelling them accordingly. A model with these capabilities should be capable of recognizing UML elements from a text and place it in the UML class diagram through its relations to other elements.

1.1 Research Objectives

In this thesis I will evaluate the capabilities of BERT models for NER in the domain of UML modelling and create a relationship extraction heuristics model. The expectation is that the implementation of a BERT model will be very useful for extracting important words and references from the text, because it can take the context of surrounding words into consideration when predicting a label [5].

It is very interesting to see how this approach performs on this task against existing methods, which leads to the following research question:

RQ1 *Which machine learning model performs the best at generating metadata for UML class models from natural language requirements texts?*

Since this thesis is concerned with two tasks, NER and RE, we will need to evaluate the performance of our models on those two tasks, resulting in two additional research questions:

RQ2 *How does a BERT model perform on a Named Entity Recognition task on natural language requirements texts compared to other baselines?*

RQ3 *How well does a rule-based relationship extraction model perform on natural language requirements texts of UML class models?*

In order to train and test the model, an annotated dataset is needed. I need to set up an annotation scheme with corresponding annotation guidelines, in order to guarantee consistent

annotations. Subsequently, this leads to the following research objectives:

RO1 *Annotate a dataset of natural language requirements texts to train and evaluate the model on.*

In addition, the model itself needs to be thoroughly researched and implemented, resulting in the research objective:

RO2 *Thoroughly study the BERT model architecture and relevant heads for NER, and synthesize a pipeline through which the model is trained and evaluated.*

1.2 Thesis Overview

Section 1 introduces the problem and explains the approach to solve this, as well as outlining the research questions and objectives. Then, Section 2 is used to provide the necessary background information on UML class models and the two tasks: NER and RE. Finally, this section discusses some related works in Section 2.4. The next section, Section 3, is concerned with the dataset that is used and how the annotation process was conducted. Section 4 goes more in-depth on the approach for the two tasks and explains the design choices for both models. I discuss my experiments and results in Section 5. This section also explain the process of optimizing the hyperparameters, as well as the comparison against related works. Finally, I draw my conclusions and suggest possible routes for further research in Section 6.

2 Background and Related Work

In this chapter necessary context and background information will be discussed. Specifically, UML class models will be explained and how requirements texts can be used to create them. Moreover, I will also introduce the two tasks: Named Entity Recognition (NER) and Relations Extraction (RE).

2.1 UML Class Models

In order to transition from requirements listings into the design phase quickly, UML class models are used. They provide a clear overview of processes and workflows withing a software system, that allow the user to capture requirements of a systems in a quick and efficient manner. There are other modeling languages, like Business Process Modeling (BPM) [6] or Systems Modeling Language (SysML) [7], but according to Miles and Hamilton [8], UML offers six main advantages: it is *fast*, *concise*, *formal*, *comprehensive*, *scaleable*, *built on lessons learned* and it is the *standard*. In essence, this means that the language is a standardized set of best-practices, that is well-applicable to different sizes of projects.

There are multiple different types of UML diagrams and in this thesis I will focus on the Class Diagram. Classes can have a name, attributes, operations and different types of relations to other classes. Figure 1 shows an example of a class, with multiple attributes: 'name', 'publicURL' and 'authors'. The plus or minus determines the visibility of the attribute in this context and the type is shows after the semicolon. This class also has an operation 'addEntry', which can be seen in the bottom section of the class.

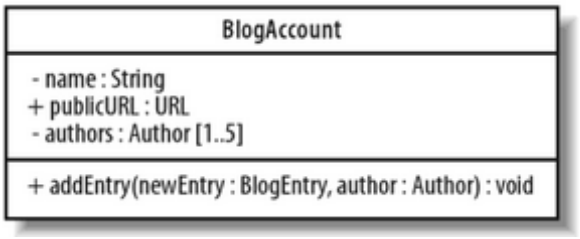

Figure 1: Example of a small UML Class Diagram. A class 'BlogAccount' is displayed, with three attributes 'name', 'publicURL' and 'authors', as well as an operation 'addEntry()' [8].

There can be associations between classes, which can have different multiplicities on both ends of the associations: 'exactly one', 'one-or-more' or 'zero-or-more'. Figure 2 shows an example of an association between classes with multiplicities. This association would be interpreted as 'One blog account has zero or more blog entries.'

Figure 2: Example of an association between two classes. This would be interpreted as 'One blog account has zero or more blog entries.'[8].

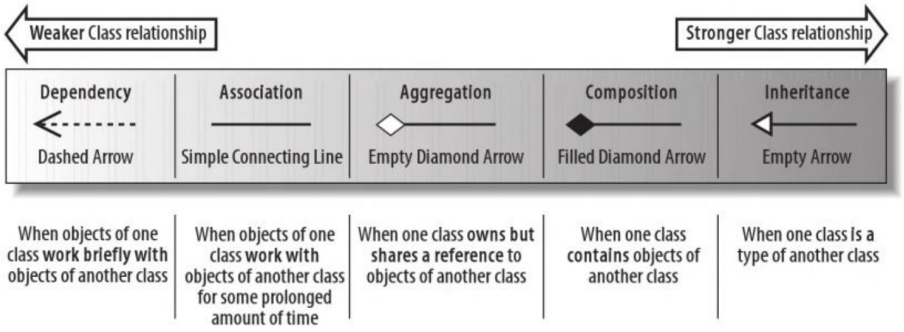

Figure 3: The different types of relationships in UML class models [8].

There are five different types of relationships in UML class diagrams, which can be seen in Figure 3. In this thesis, the focus will be on associations mostly, but also somewhat on aggregations and compositions. Composition relationships can be interpreted as an 'is-part-of' relations, where one parent class contains objects of another child class. In this case, the child class cannot exist on its own. Aggregations share similarities with compositions, as it is also a parent-child relation. However, with aggregations the child class can also exist on its own.

2.2 Named Entity Recognition

The first step in extracting relevant UML-related information from texts is recognizing and correctly labelling entities in the text, which is the task of Named Entity Recognition (NER). Each word gets assigned a label, which the model can learn through either unsupervised learning or supervised learning approaches [9]. Figure 4 shows an illustration of how this works. A sentence gets split up into words, on which the model will make its predictions. For supervised learning approaches this task is treated as a multi-class classifications task, in which it will use the true labels of these words to learn accurate predictions, since it can positively reinforce correct predictions and penalize false predictions by adjusting the weights of the network. This way the model can extract entities from a text.

Sometimes entities consist of multiple words, in which case it is called a *span*. For example: the entity 'earlier this week' in Figure 4 is one entity of the 'Date' class. However, the knowledge that these words belong together would be lost if we would only do predictions on each separate word. To overcome this issue, the BIO tagging scheme is often implemented, which assigns a label 'B', 'I' or 'O' to a word, depending on whether the token is at the beginning of a span, inside a span or outside of a span, respectively. These labels are combined with the regular class labels. For example: the span '*date of birth*', which is one entity of the 'ATTRIBUTE' class, would be labeled as ['B-ATTRIBUTE' (*date*), 'I-ATTRIBUTE' (*of*), 'I-ATTRIBUTE' (*birth*)] for the purpose of NER.

Recently, transformer models, like the BERT model, have been able to make good progress in this task, due to its use of continuous real-valued vector representations of words, which can also take surrounding context of a word into consideration [5, 9]. Originally this task was used to extract relatively simple concepts from newspapers, such as locations and persons from the CoNLL-2003 benchmark, which is a common benchmark to compare NER models on [10]. However, this principle can also be extended to extract more advanced and abstract concepts from UML class models from

Figure 4: An example of how Named Entity Recognition works. The model makes predictions for each word and manages to extract multiple entities from the text [11].

texts, by choosing the right labels and using a dataset that is annotated for this purpose.

The biggest challenge for this field of natural language processing is concerned with the data, since deep learning models require a lot of data to properly learn [9]. That means that a large dataset is necessary for good performance of the model. Moreover, this adds the challenge of annotation, as this has to be done consistently. Finally, the quality of the data needs to be of a proper standard as well, as it is used to train the model. All of this adds extra difficulty to the task of NER.

2.3 Relation Extraction

When the NER task is done, relations need to be extracted and correctly labeled from the texts. This is the task of Relation Extraction (RE), where pairs of entities get classified as a relation and labeled with the right label. Figure 5 displays an example of how RE is performed. A sentence containing two entities is put into a RE model, which can predict the relation within that sentence. This can be accomplished in multiple ways. The pioneering methods utilized pattern-matching models, where a large number of matching patterns are constructed, which are used to extract relations. These patterns usually require some form of human involvement [12]. Later, statistical methods were applied, as well as neural networks [12]. Both are favored over pattern-matching, as they can obtain better performance, generalize better and require less human involvement [12]. Lately, transformers have become the new state-of-art for this task [13, 5].

Even though good scores can be reached on this task, it is a complicated one. Just like NER, a large annotated dataset is required, which is difficult, as collecting and annotating can be time-consuming. Furthermore, for many long and specific relations it is difficult to train a model, because there are not enough examples for the model to learn. Additionally, relations are often not contained within one sentence. This means that when searching for relations, the model has to search within multiple sentences, which can add to the computation time [12].

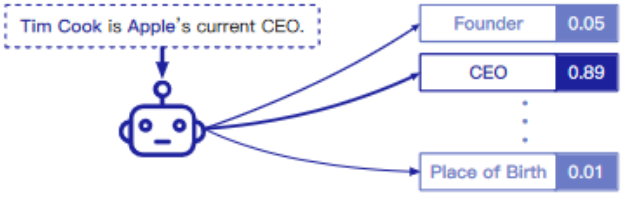

Figure 5: An example of how Named Relation Extraction works. Given a sentence containing two entities, RE models can extract the relation [12].

For the purpose of this thesis, the output of the NER model will be used as input for the RE model. However, the quality of the output of the NER model greatly influences the RE performance, which creates additional complexity. Moreover, there can be more than one relation in a sentence, which adds to the difficulty too, since there is no indication of when you have extracted all of the relations from a sentence.

2.4 Related Work

In 2022, Rigou and Khriss built a model using deep learning techniques to automate the process of generating UML class models from textual requirements [14]. Their model consisted of a BERT encoding layer, with three feed-forward neural network layers to act as decoders on top. This model performs three tasks: NER, RE and coreference resolution. They used a dataset of 20 documents, which was hand-annotated by one annotator. They tested a number of sub-models, but their most-promising model included the BERT-model, in combination with a 'weighted average' technique, that produces an encoded vector representation from a weighted average of the encodings of the tokens in a span [14]. This model managed to achieve promising results, with F1-scores of 89.3%, 54.9% and 73.6%, for NER, coreference resolution and RE respectively. Most notably, they mentioned that a bigger dataset is necessary to realize a full UML class model generation application.

Alternatively, rule-based approaches are used to extract information from natural language requirements to texts. Research was conducted in 2015 to generate UML class models using this approach, resulting in a model called the Automatic Builder of Class Diagram (ABCD) [15]. They focused their effort on synthesizing rules that can recognize patterns in the text in order to extract information. Their approach included four steps. First is sentence splitting, where they split up a full text into individual sentences. The sentences are then tokenized by the Stanford Parser, which results in tokens that are words and punctuation marks [15, 16]. The next step is Part-of-Speech tagging, in order to retrieve semantic information from the text. The final step is syntactic parsing, where they apply their rule-matching. This method is quite capable, and is able to extract the names of two entities, their multiplicities and the relation between them with just one matching pattern. This model managed to achieve a recall and precision score of 89.0% and 92.3%, respectively. However, they also reported an overgeneration rate of 26%, which is a metric they introduced; it measures false predictions by the model. Specifically, this model has issues dealing with synonyms and it can confuse methods and associations, as they can both be identified by verbs.

A combined span-based NER and RE model was also built by Ebert & Ulges in 2019, named SpERT [17]. Their approach consisted of three steps: span classification, span filtering and relation

classification. The first step includes creating token embeddings of the input texts, using a BERT 'base-cased' model. They fuse these token embeddings with width embeddings and a context token and input this into the span classifier, which decides whether the input tokens form a span together. This way they manage to combine local and global features, which enhances the semantic representations [18]. The next step is to filter out the no-entity types, which leaves us with just the classified entities. Then, in the final step, pairs of spans are formed and combined with context embeddings, which is the input for the relation classifier. This way, the model can do span-based joint NER and RE. They tested their model on three benchmarks, about news articles (CoNLL04 [19]), abstracts of AI papers (SciERC [20]) and medical reports (ADE [21]). The results show that SpERT outperforms the state-of-the-art models of that time by up to 2.6%.

Generally, a transformer-based approach is most promising, as they are the state-of-the-art and can reach very good scores, given a large dataset [17]. For that reason a transformer-based approach has been adopted in this thesis for the NER task. For RE a transformer-based approach was not realistic, due to time restrictions. Therefore, pattern-matching is chosen, because it is more straightforward and still capable of obtaining a good performance, as can be seen in the ABCD model [15].

3 Data

For this thesis, I am using a dataset of natural language requirements texts consisting of 9 documents, collected from lectures, assignments and actual companies [22]. Since not all of these documents were written for this exact application and because natural language can be quite ambiguous [23], there is some fluctuation in quality, which can lower the usefulness of a document for the purpose of this thesis.

The length (amount of words) of a document can also vary between documents, with some documents having around 800 words, while others only contain 100 words. This is not a problem for the model, except that it can take a bit longer to run. The quality of a document, however, is usually a bit lower for longer documents, as certain elements of the document are likely to be repeated.

3.1 Data Annotation

The requirements texts are raw natural language, without labels. However, our model needs labeled training data to be able to learn. Therefore, we need to annotate the texts, which includes providing a label for each word in the text, as well as extracting the relations. This is a subjective process, due to the ambiguous nature of natural language [23]. In order to guarantee consistent annotations, I set up annotation guidelines in collaboration with a UML expert. I carried out the annotation process, which guarantees consistency in the annotations. For ambiguous and uncertain cases, in which it is unsure what the right annotations are, the UML expert was consulted.

The annotation guidelines for NER are described in Table 1. They are not very complicated, as we usually annotate words quite literally. If a class appears in the text, we annotate it as a class. We annotate the following entity types: Class, Attribute, Association, System, Operation. It is important to note that enumerations and notes/constraints are not annotated, as these rarely appear in the texts. Furthermore, during data preprocessing the 'O' label, from the BIO-tagging scheme that was mentioned in Section 2.2, is added to all words that have not been annotated, as they are not part of a span.

Table 1: The annotation guidelines for named entity recognition. These are combined with the BIO-tagging scheme at a later stage, where all entities have 2 forms: one with a 'B' tag and one with a 'I' tag. Everything that is not annotated gets the label 'O'.

Entity	**For**	**Notes**
Class	For UML classes	-
Attribute	For attributes of classes	-
Association	For the labels of associations	To retain label information and for multiplicity purposes
System	For higher-level overarching systems	These entities would otherwise likely be wrongly classified as classes
Operation	For operations of classes	-

For RE we have set up the annotation guidelines as can be seen Table 2. A more in depth

Table 2: The annotation guidelines for relation extraction. Stick to the text!

Relation	Direction of annotation	Notes
Association1..*	From class to association and association to class	For multiplicity 'one-or-more'
Association1	From class to association and association to class	For multiplicity 'exactly one'
Association*	From class to association and association to class	For multiplicity 'zero-or-more'
Composition	From parent class to child class	-
Aggregation	From parent class to child class	-
Subtype	From subtype to parent class	-
Attribute	From attribute to corresponding class	-
Operation	From corresponding class to operation	-
Span	From begin of span to end of span	For split up spans
Coref	From coreference word to original word	For coreference resolution

explanation of why labels were chosen for RE and NER can be found in Sections 4.1 and 4.2. Most of these rules are self-explanatory. Important to note is that the quality of the text heavily influences the quality of the annotations. When a sentence is ambiguous, the annotation will be unclear too. To maintain consistency, we have decided to stick to what the text says and tried to not interpret and reason too much what the text might mean in uncertain cases, as this is often also the most intuitive way to label the sentence for the model.

One of the difficulties of annotating natural language texts for UML class model purposes is saving relevant context. In particular, the labels of an association relation. In UML models, an association will exist between two classes with a label, such as 'is part of'. In natural language, this label will be a verb near the classes, which can be different for each relation. As a result, problems arise with the prediction made by our model, as a model performing RE can predict a relation between two entities and its relation type (subtype/association etc), but it will not predict the label of the relation. In order to overcome this, we have decided to split up the annotation of associations into two parts: the starting entity will have a relation of type 'association' to the corresponding verb in the sentence. This verb will get the NER-label 'association', and have an outgoing relation to the other entity, also with the relation label 'association'. This way, the label of of associations between entities can be saved. An example of how an association relation gets annotated can be seen in Figure 6.

In addition to labels of associations, multiplicity of associations adds another complication: when annotating a relation, it will take as information the entities that partake in the relation and the label of the relation ('subtype', for example). However, the relation 'association' has multiple different variations, because both sides of the relation can have a multiplicity label: 'exactly one', 'zero-or-more' and 'one-or-more'. The easiest approach for this, is to combine a multiplicity label with the relation label. This results in three labels for an association: 'ASSOCIATION1', 'ASSOCIATION*' and 'ASSOCIATION1..*, which represent the multiplicities 'exactly one', 'zero-or-more' and 'one-or-more' respectively.

In collaboration with the with the UML expert we decided to add another relation 'Span', in order to deal with spans that are split up. It would sometimes occur that a span was split up by other words in the sentence. Using this relation, we could annotate this span properly, by forming

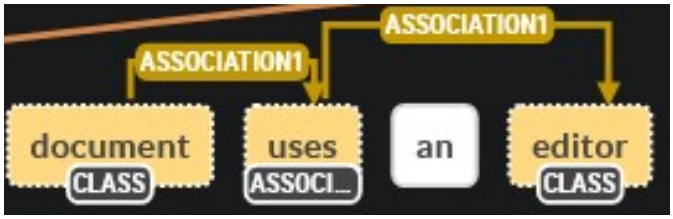

Figure 6: Example of an annotated association relation with the annotation tool Prodigy by SpaCy [24]. The UML association goes from 'Document' to 'editor' with label 'uses' in the class diagram, but it is split up into two relations for annotation purposes.

a relation 'Span' from the beginning entity of the span to the ending entity.

For the annotation process, I used the annotation tool Prodigy, by SpaCy [24]. This tool allows for joint entity and relation annotation, which is very useful. Furthermore, it produces a JSON output with all the data. This does require some extra data preprocessing, but this works pretty well once it is set up. An example of an annotation in the Prodigy tool can be seen in Figure 6.

There are 11 labels for the NER task, which will be explained more in depth in Section 4.1. Table 3 shows the distribution of NER labels, which is heavily skewed. Section 4.1 discusses my approach to solve this issue.

Table 3: The distribution of NER labels in the dataset.

Label	**Count**
B-CLASS	383
I-CLASS	92
B-ATTRIBUTE	119
I-ATTRIBUTE	91
B-ASSOCIATION	166
I-ASSOCIATION	58
B-SYSTEM	43
I-SYSTEM	35
B-OPERATION	10
I-OPERATION	1
O	1505

Table 4 shows the counts per relation-type in the dataset. Again, these are heavily skewed, with some relations barely appearing in the texts at all. Section 4.2 discusses my approach to tackle this problem.

Table 4: The distribution of relation labels in the dataset.

Relation	Count
ASSOCIATION1	199
ASSOCIATION1..*	101
ASSOCIATION*	9
ATTRIBUTE	114
SUBTYPE	45
OPERATION	11
COMPOSITION	2
AGGREGATION	4
SPAN	3
COREF	24

4 Method

In this section the general approach for tackling the problems of Named Entity Recognition (NER) and Relation Extraction (RE) is discussed, as well as the reasoning behind certain design choices.

4.1 Named Entity Recognition

NER, in essence, is the task of token classification, in which the input text is reduced to tokens by the tokenizer and each token has a label. It is the task of the model to be able to accurately predict the UML labels of each token. In this thesis I used the BERT 'base-uncased' model with the BertFastTokenizer to perform this task, both directly loaded from the transformer package of HuggingFace [25].

The to-be-predicted NER label consists of two parts: a UML label and a *chunking* label, which refers to the role of a token in an entity. Since the NER model will make predictions on single words, it is likely that the knowledge of what word belongs to what span will get lost. To overcome this issue, I have implemented the aforementioned BIO-tagging scheme (Section 2.2). This chunking label is combined with the UML label, resulting in a total of 11 possible token labels, which can be seen in Table 3. Most of these labels are self-explanatory, as they belong in a UML class model. The exceptions here are the system label (label 7 and 8) and the O label (label 11). The reason for adding a system label to these labels, is to make a distinction between classes and higher-level, overarching systems, which would be classified as a class otherwise, as they appear similar to classes. The addition of this label allows our model to more precisely label classes. The O label is used to label everything that is not relevant for a UML class model. Since natural language can be both ambiguous and redundant there is a lot of irrelevant text [23]. As a result, the O class is the majority class.

Furthermore, I have chosen not to includes notes/constraints in my labels, as they do not occur often in texts. Consequently, they would be very hard for my model to identify. Moreover, they would make the prediction of the other labels harder, since there would be a greater number of labels.

The approach for applying a transformer model for a NER task involves a lot of data preprocessing, as we are dealing with unstructured natural language as input. For training purposes, the labeled input data needs to be tokenized, such that the NER labels correspond the tokens. This includes the necessary step of making sure that when a word get split up into subwords, that all of the subwords have the NER label of the original word. This is easier for inference, as we do not have to worry about the corresponding NER labels of subtokens for this case.

As discussed before, the class distribution is very imbalanced, because the O class is a very large majority and classes like 'I-SYSTEM' or 'I-OPERATION' do not appear very often. As a result, the model would often predict (almost) everything as the majority class. To overcome this, we would have to penalize these types of wrong predictions harder, in order for the model to learn to predict other labels. To do this, I have constructed a custom loss function, which overwrites the standard loss function of the Huggingface Trainer. This custom function performs cross-entropy loss with weights per class, relative to the number of appearances of that class. This way, wrong predictions of the majority class get penalized harder than wrong predictions of the minority class.

4.2 Relation Extraction

I decided to develop a rule-based RE method because the amount of labelled data is too small to train a supervised transformer-based classifier that can recognize all relation classes. The extraction method uses the predicted NER labels by my NER model. Using a rule-based approach to extract relations is a familiar approach and usually achieves good results, like in the the ABCD model, mentioned in Section 2.4 [15]. The heuristic rules were formed and optimized on two documents from the dataset and later tested on the entire dataset. The python code for the implementation of the RE rules can be found in Appendix A.1.

These are the heuristics rules used for extracting relations from the text, with examples:

- Association (exactly one and one-or-more): When an entity x with the NER-label 'Association' is encountered, make a relation (y, x, 'ASSOCIATION1') from the first 'Class' entity y to the left of entity x. Also, form a relation (x, z, 'ASSOCIATION1') from x to the first 'Class' entity on the right of x. In both cases, if the class entity ends in an 's', it is likely plural. Therefore, make it an 'ASSOCIATION1..*' relation instead. Skip the first class found on the left if the association has 'and', 'or' or ',' in front of it. A typical association relation is in this format:

  '*Each document uses an editor.*' (words with the NER label 'class' are denoted in red and 'association' in blue)
  Contains relations:
  - (*document*, *uses*, 'ASSOCIATION1')
  - (*uses*, *editor*, 'ASSOCIATION1')

- Association (zero-or-more): When an entity x with the NER-label 'Association' is found and the entities $x-1$ and $x-2$ together form the words 'can be', search for the first class entity y on the left of x and form relation (y, x, 'ASSOCIATION*'). Also form a relation (x, z, 'ASSOCIATION*') to first class entity z on the right, if x is one of the following words: 'score', 'scores', 'sends', 'controls', 'list'. This is an example of an association relation with the multiplicity of zero-or-more:

'*A football player can be sold to another football team.*
Contains relations:
- (*football player*, *sold to*, 'ASSOCIATION*')

- Subtype: If a listing of three or more classes $x_i, x_{i+1}, ..., x_n$ is found, only separated by ',' or 'and', search for the first class y on the left of the listing within the same sentence. Form relations (x_j, y, 'SUBTYPE'), where $i \leq j \leq n$. If the entity $y-2$ contains the word 'in', skip this entity. A typical sentence containing subtype relations would look like this:

  '*... basically just four controls (jump, move forward, move backward, duck).*'
  Contains relations:
  - (*jump*, *controls*, 'SUBTYPE')
  - (*move forward*, *controls*, 'SUBTYPE')
  - (*move backward*, *controls*, 'SUBTYPE')
  - (*duck*, *controls*, 'SUBTYPE')

- Attribute: If an entity x with the NER label 'Attribute' is found, make a relation (x, y, 'ATTRIBUTE') to the closest 'Class' entity y on the left of x within the sentence. If the entity $y-2$ contains the word 'in', skip this entity. An example sentence looks like this:

  '*The instructor has a name, surname, title, and specialty.*' (words with the NER label 'attribute' are denoted in orange)
  Contains relations:
  - (*name*, *instructor*, 'ATTRIBUTE')
  - (*surname*, *instructor*, 'ATTRIBUTE')
  - (*title*, *instructor*, 'ATTRIBUTE')
  - (*specialty*, *instructor*, 'ATTRIBUTE')

- Operation: If an entity x with the NER label 'Operation' is found, make a relation (y, x, 'OPERATION') with the closest 'Class' entity y to the left of x within the same sentence. If the entity $y-2$ contains the word 'in', skip this entity. An example of a typical operation relation looks like this:

  ' *After running for a certain distance or given time, the level will be cleared.*' (words with the NER label 'operation' are denoted in green)
  Contains relation:
  - (*level*, *cleared*, *'OPERATION'*)

- Composition: When two entities x and y, both with the NER-label 'Class' are found, with only the word 'of' in between, form relation (x, y, 'COMPOSITION'). An example:

  '*When earth receives a threat that a nearby colony of aliens is about to launch an attack, ...*'
  Contains relation:
  - (*colony*, *aliens*, *'COMPOSITION'*)

- Aggregation: When a listing of at least three class entities $x_i, x_{i+1}, ..., x_n$ is found with the words 'of' in front of it, search for the first class y on the left of the listing within the same sentence. Form relations (x_j, y, 'SUBTYPE'), where $i \leq j \leq n$. If the entity $y-2$ contains the word 'in', skip this entity. An example:

'*The application will allow updating the list of loan items, user registration, lending transactions, and reservations.*'
Contains relations:
- (*list, loan items*, 'AGGREGATION')
- (*list, user registration*, 'AGGREGATION')
- (*list, lending transaction*, 'AGGREGATION')
- (*list, reservations*, 'AGGREGATION')

- Coreference: Manage a list of regular coreference words (e.g. 'his', 'they', etc.). If an entity x in the text matches one of the known coreference words, seek the second-closest 'Class' entity y to the left of x, as the closest 'Class' entity is likely the end of an association. If the entity $y-2$ contains the word 'in', then do not use this class entity and seek the next class entity on the left. Form the relation (x, y, COREF). This an example of a coreference relation:

  '*There is a goalkeeper in the defense, and he plays to clear the ball.*'
  Contains relation:
  - (*he, goalkeeper*, 'COREF')

- Additional note: do not form relationships with entities that are padding. This can occur due to false predictions by the NER model.

Associations in UML class models go from class to class. However, we would lose information by annotating it this way, such as the name of relation. Therefore, we split up associations into two, separated by the association entity, with one relation from the first class to the association and a second relation from the association to the second class. As a result, when we encounter an association we have to find two relations, which are usually to and from the closest class to the left and right of the association. There are exceptions in the case of multiple verbs for the same subject, for example:

'*The customer pays the bill and orders the meal.*'

In this case 'orders' would be labeled as an association. If we would follow the heuristic rule, we would find 'bill' as the first class entity to the left and form the relation (bill, order, 'ASSOCIATION1'). This is incorrect, as it should have skipped this class and found 'customer' instead. For this reason I edited the rule to skip the first class found if the association entity has 'and', 'or' or ',' in front of it. Furthermore, in most cases where there was a multiplicity of 'one-or-more', the class entity would end in an 's', which is why I used this to extract multiplicities. The 'zero-or-more' multiplicity did not occur frequently, which is why the rules are quite specific for the cases I did encounter. Most of the associations had a multiplicity of 'exactly one'. Consequently, this is the default multiplicity when an association is formed.

Making a generalizable rule for subtypes is difficult, as subtypes occur in many different ways in a text. Most often, they occurred in short sentences, that list multiple subtypes of a class entity. For example:

'*Positions are forward, midfield and defense.*'

This is a listing and it would therefore be recognized by the extraction rule. The classes 'forward', 'midfield' and 'defense' would all form a relation 'SUBTYPE' to 'Positions'. Subtypes are also found in alternative structures, but most often it was in this form. For that reason, I have chosen this structure to base my extraction rule on.

The rule to extract attribute relations is quite simple. The NER model should have appropriately labeled attributes, so when we encounter an entity with the NER label 'Attribute', we only have to search for the closest 'Class' entity on the left to form a relation. There is an exception:

'Cars in the gallery have a license plate'.

In this sentence, 'license plate' is an attribute of the class 'Cars'. However, the closest 'Class' entity on the left of 'license plate' is 'gallery'. To avoid this, I have added this edge case to the rule. This makes sure that sentences in this format are appropriately analyzed.

The operation relation works in exactly the same manner as the attribute relation, and is triggered by encountering an entity with the NER label 'Operation' instead of 'Attribute'.

In order to extract coreference relations, I keep a list of known 'coreference words', like 'his' or 'they'. When an entity matches one of the words we look for the 'Class' entity on the left of this word. Just like with attributes and operations, we skip a class if it has the word 'in' in front of it. Additionally, we also skip the first class found, as this is usually the end of an association. For example:

'When the goalkeeper saves the goal, the system announces his number and name.'

In this sentence, 'his' would be recognized as a coreference word and we would search for the 'Class' entity on the left of this word. The first class we would encounter is 'goal', which is not the correct class here as it is the end of the association relation between 'goalkeeper' and 'goal'. For this reason, we skip the first found 'Class' entity. Previous research on rule-based coreference resolution has also shown this is a typical structure for coreference relations [26]. This extraction rule can extract these types of relations without doing actual syntactic parsing.

For the aggregation, composition and association* relation there were very few appearances, so the extraction rules often included specific words on which the rule is triggered. That means that these rules work on the examples in this dataset, but should be extended to be more complete if there was more data available. The approach for these rules is also in line with the work of TianTian [27].

I have decided to leave out the span relation, since this relation did not appear in the data often enough to form a robust relation extraction rule.

5 Experiments and Results

This chapter will discuss the results of the model after hyperparameter optimization, as well as the experiments conducted with the models.

5.1 Hyperparameter Optimization

Transformers have a lot of different *hyperparameters*: settings that can be adjusted, that alter the performance of the model, such as learning rate, weight decay or the number of training epochs. All of these hyperparameters together form a high-dimensional grid of configurations, where each hyperparameter is a dimension. Finding the best-performing configuration of hyperparameters in this high-dimensional grid is an optimization problem, for which I used the hyperparameter sweep function of Weights and Biases, which is a developer tool for machine learning [28]. To traverse this hyperparameter grid, I used a Naive Bayes algorithm, which is integrated into Weights and Biases. The result of this hyperparameter sweep can be seen in Figure 7, where each column depicts a hyperparameter and the most-right column the F1 score. Each line represent a single run with a certain configuration of hyperparameters. Each run was conducted on one fold from a 4-fold cross-validation split, consisting of a train set of six documents and a evaluation set of three documents. A total of 39 runs were conducted. The configuration of hyperparameters of the best-performing run can be found in Table 5.

Table 5: The best-performing configuration of hyperparameters, found after the hyperparameter sweep by the Naive Bayes algorithm of Weights and Biases [28].

Hyperparameter	**Value**
Learning rate	8×10^{-5}
Epochs	50
Weight decay	0
Warmup steps	5
Train batch size	4
Eval batch size	32

5.2 Named Entity Recognition Results

Using the hyperparameter configuration found during the sweep, I conducted a 4-fold cross-validation run of the NER model, using the hyperparameters found in Table 5. I split the dataset into a train set, consisting of six documents and a evaluation set, consisting of three documents. The results of this run can be seen in Table 6. In this table, the averages $\pm$ the standard deviations of the four different runs are shown. The support column shows the number of samples of that specific label in the true labels. I calculated the weighted average, as well as the macro averages to gain a balanced insight into the results, because the weighted averages show the global performance of the model, while the macro averages show the influence of each individual class. The 'O' label has not been included for the evaluation, which is common when the BIO-tagging scheme is used. This is

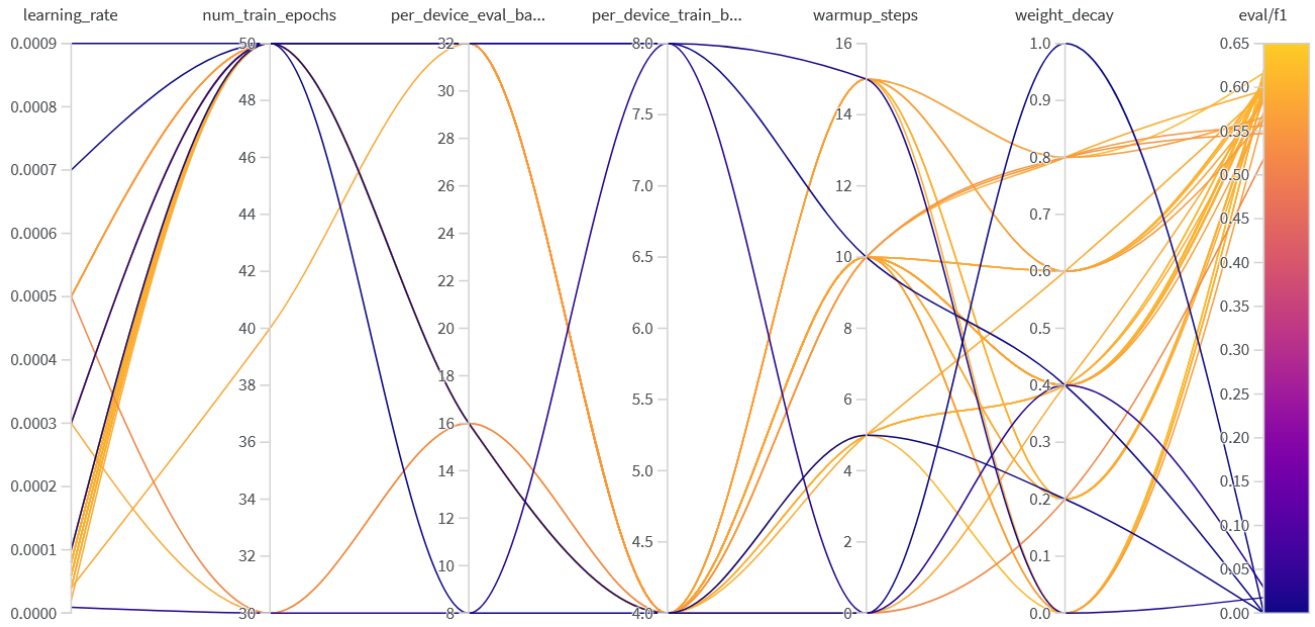

Figure 7: The results of the hyperparameter optimization of the NER model. Columns in this graph represent features with different values. Each line represents a run with a certain configuration of hyperparameters. The most right column is the F1-score on the evaluation set. The best configuration reached a F1-score of 62.0%.

because the 'O' class is always a majority and including this in the evaluation would give an unfair impression of the results.

We can see that there is a clear difference between classes with higher support and classes with lower support, which is a perfect example of why a large dataset is necessary when working with deep learning models, because the labels with a lot of support are predicted much more accurately by the model than the labels with lower support. This makes sense, because the model has more examples to learn how to identify the entity from.

Often, the labels that contain a 'B' have more support, because these labels can exist on their own, in the form of single-word entities, while 'I'-labels can only exist following a 'B'-label. It is therefore logical that 'B'-labels are identified better than their corresponding 'I'-label by the model. Furthermore, it is noticeable that the majority class scores much higher, which is also due to its higher support. The labels 'B-OPERATION' and 'I-OPERATION' on the other hand have not been correctly identified by the model, because they have extremely low support. For most labels between the majority and minority class, the performance is decent. The variance for these mid-range classes is also a bit higher.

The weighted averages are between 56.8% and 60.7%, which is decent, but this is also due to the influence of the 'B-CLASS' label. Therefore, I calculated the macro-average as well. This is important, because the macro-average gives equal importance to each class. That means that the influence of the majority class is neutralized. Now, the averages lie between 34.3% and 42.8%, which is a lot lower. One reason that this is much lower, is because the model fails to predict operation entities. However, it is important to keep in mind that this experiment was conducted with a small dataset of only 9 hand-annotated documents. We can see that the classes that are well-represented

Table 6: Results for the Named Entity Recognition task, run with 4-fold cross-validation. The results show the mean score ± the standard deviation. The final column shows the number of samples in the true labels. The O label is not included for the calculation of the weighted and macro average, because this class is not relevant for the UML model itself.

Label	**Precision** (%)	**Recall** (%)	**F1** (%)	**Support**
B-CLASS	71.3 ±11.0	81.5 ±11.8	75.0 ±3.9	540
I-CLASS	52.8 ±23.8	49.5 ±7.0	49.0 ±12.9	124
B-ATTRIBUTE	59.5 ±6.1	51.3 ±18.9	54.0 ±11.9	171
I-ATTRIBUTE	70.0 ±10.3	47.0 ±22.7	52.3 ±12.0	134
B-ASSOCIATION	53.0 ±4.7	57.0 ±12.3	54.8 ±7.8	223
I-ASSOCIATION	37.2 ±44.8	14.8 ±12.1	17.0 ±13.5	77
B-SYSTEM	70.5 ±22.3	25.3 ±21.0	31.3 ±16.4	62
I-SYSTEM	13.3 ±23.1	7.3 ±12.7	9.7 ±16.7	44
B-OPERATION	0	0	0	11
I-OPERATION	0	0	0	1
Weighted average	**60.7**	**58.4**	**56.8**	
Macro average	**42.8**	**33.4**	**34.3**	

are classified quite accurately, which suggest that if more data was available the problematic classes would be identified better too.

5.3 Relation Extraction Results

In order to develop a rule-based model that is generalizable for out-of-sample texts, I developed the rules on two documents from my dataset, that were selected based on quality of the documents. This is in order to not overfit the rules on specific elements of a text, but rather keep them general so that they work for other texts too. I tested my rule-based extraction method in a 4-fold cross-validation run, where I used the predictions of the NER model as labels for the texts. The split was the same as with NER: each fold contained a train set of six documents and an evaluation set of three documents. The results of this run are presented in Table 7. It is worth noting that I excluded the 'SPAN' relation, since I was unable to construct an extraction rule, because it barely appeared in the data.

The relation extraction model manages to extract nearly half of the relations, but in the process also makes a lot of false predictions, which explains the low scores. This is not completely unexpected, as there are multiple factors that influence the performance of this model: (1) the quality of the data, (2) the quality of the annotations and (3) the quality of the predictions of the NER model. First off, (1) the quality of data is important, because this allows us to form practical extraction rules. When data is ambiguous, or some relations are underrepresented in the dataset, it is significantly harder to form proper extraction rules. For some relations, it was significantly harder to form an extraction rule, due to a relation only occurring in very specific cases or there not being an apparent pattern to match. This was the case for the 'ASSOCIATION*', 'COMPOSITION',

Table 7: Results of the relation extraction rules for 4-fold cross-validation, using the predicted labels of the NER model.

Relation	**Recall** (%)	**Precision** (%)	**F1** (%)	**Support**
ASSOCIATION1	32.6 ±14.2	30.0 ±8.3	30.3 ±10.3	265
ASSOCIATION1..*	27.3 ±7.4	35.2 ±9.7	30.7 ±8.2	133
ASSOCIATION*	34.4 ±47.2	29.2 ±34.4	30.7 ±38.6	14
ATTRIBUTE	20.2 ±5.7	25.8 ±7.4	22.3 ±5.8	162
SUBTYPE	3.2 ±6.5	6.7 ±13.4	4.4 ±8.7	63
OPERATION	0	0	0	15
COMPOSITION	16.7 ±28.9	33.3 ±57.7	22.2 ±38.5	2
AGGREGATION	0	0	0	4
COREF	38.5 ±45.6	33.3 ±23.6	31.9 ± 30.7	30
Weighted Average	**25.3**	**27.2**	**25.3**	
Macro Average	**19.2**	**21.5**	**19.2**	

'AGGREGATION' and 'SPAN' relations. For the first three cases, we did find an extraction rule. However, for the last relation, the 'SPAN' relation, it was not possible to form an extraction rule on so few occurrences. Secondly, (2) the quality of the annotations is also important, as inconsistencies can occur. In an ideal world, there would be three independent domain experts to carry out the annotations, but because of time restrictions, this was not possible. Finally, (3) the quality of the NER output influences the performance of the rule extraction model. It happens often that the NER model wrongly predicts an association entity and as a result, the association extraction rule is triggered, leading to two falsely predicted association relations. Moreover, all of the words in a span need to be correctly predicted, in order to combine it into the correct span, because a single word missing from a span that is part of a relation will result in the entire relation being incorrect. This adds additional complexity to the task. For the 'OPERATION' relation, there was the issue that the NER-label 'B-OPERATION' and 'I-OPERATION' were very underrepresented in the dataset. As a result, the NER model cannot accurately predict operation entities, which leads to no correctly predicted operation relations. The 'AGGREGATION' relation was also not extracted, due to wrong predictions of the NER model. However, despite all of these challenges, the model was still capable of predicting nearly half of the relations correctly, which does prove its potential.

5.4 Comparison to Related Work

This section will compare the model proposed in this thesis with other, comparable models. It is worth noting that these are not direct comparisons, as they were not trained and evaluated on the same datasets.

Rigou and Khriss

Rigou and Khriss proposed a model, consisting of a BERT encoding layer with three feed-forward neural networks on top as decoders, to automate the process of generating UML class models [14]. They reported F1-scores of 89.3%, 54.9% and 73.6% on the tasks NER, coreference resolution and RE, respectively. This outperforms the model proposed by this thesis on both NER and RE. This is likely due in part to the data; they used a dataset of 20 documents, which were of significant length. This means that their model had more examples to learn from, which can explain their increased performance. Furthermore, the use of feed-forward neural networks can also influence the performance and therefore may also be a possible explanation for the difference in performance.

Notably, they only classified two types of relations: associations and attributes. They ignored the multiplicity and label information of associations as well, which our model does not. It is likely that their performance would have been lower, had they chosen to include this information, because this would make the task more complex.

Eberts and Ulges

In their paper, Eberts and Ulges introduced SpERT: a transformer-based joint entity and relation extraction model, which features strong negative-sampling [17]. They trained and evaluated their model on three different benchmarks: CoNLL04 [19], SciERC [20] and ADE [21], which gives them the advantage of having a large dataset to train on. As for results, they managed to score between 70.3% and 89.3% F1 for NER and between 50.8% and 79.2% F1 for RE on the different datasets. This is quite a good performance on these tasks and it outperformed the state-of-the-art models of that time, as well as the model proposed by this thesis. This is presumably because of the large amounts of data used, as well as the negative sampling technique. Furthermore, they employed localized context, instead of entire sentences as context, which allowed their model to focus solely on relevant parts of the sentence, which lowers the amount of noise. This strategy, combined with negative sampling and the large amount of data, is likely the reason for their high performance.

Bozyigit et al.

Bozyigit et al. proposed a rule-based approach to tackle the problem for both NER and RE [22]. The dataset used in this thesis is a subset of the dataset used in their study, however, with different annotations, meaning that this is also not a direct comparison. Their approach can extract the classes, attributes and operations, as well as identify and label relations between classes. They managed to extract these concepts correctly and only sometimes miss or wrongly extract an entity, which is a very good performance.

Our model appears to score a bit lower, however, this difference in performance can be explained. First off, the difference in annotations matter, as their extraction rules are based on their annotations, they might work differently on alternative annotations. Besides that, our model tries to extract more information from the text, like labels of associations and multiplicity, which their model does not do. As a result, the task is a bit more complicated for our model. Finally, they have a larger dataset, which gives them more examples to base their extraction rules on. For this thesis, more data would have improved the NER model, as transformers require a lot of data to learn properly [29]. Moreover, this would have allowed me to make more extensive extraction rules for my RE

model. These differences are likely the reason for the difference in performance between the model proposed by Bozyigit et al. and our model [22].

6 Conclusion

In this thesis I have attempted to automate the process of generating metadata for UML class models from natural language requirements texts. I investigated this matter through three research questions. The first question was:

RQ1 *Which machine learning model performs the best at generating metadata for UML class models from natural language requirements texts?*

To answer this question, we have to consider the studies by Eberts and Ulges [17] and Rigou and Khriss [14], discussed in Sections 2.4 and 5.4. The model proposed in this thesis did not outperform the related work, likely due to less and lower-quality data. The model proposed by Rigou and Khriss seems to be the most promising, achieving the highest scores on both NER and RE. Therefore, I conclude that, as of this moment, their model, consisting of a BERT encoding layer, with three feed-forward neural networks, performs the best at generating metadata for UML class models from natural language requirements texts.

The model proposed by Eberts and Ulges is also promising, but scores slightly lower. Moreover, that model has not been tested on this specific context, but rather has been tested on the tasks NER and RE in general.

The second research question I investigated during this thesis was:

RQ2 *How does a BERT model perform on a Named Entity Recognition task on natural language requirements texts compared to other baselines?*

The BERT model that was implemented for this thesis managed to achieve a weighted average F1 score of 56.8% and a macro-F1 score of 34.3%. For certain classes, like the 'O' class, the F1 score was much higher, meaning that the model could accurately extract this information. However, the model still struggled extracting entities that had the 'I' label from the BIO-tagging scheme, since these appeared less often in the data. Still, the model managed to accurately extract labels with high support, which proves that this is a viable approach. These findings are also in line with the results of both Eberts and Ulges [17] and Rigou and Khriss [14], as they also manage to achieve a good performance with the use of a BERT model.

The final research question was:

RQ3 *How well does a rule-based relationship extraction model perform on natural language requirements texts of UML class models?*

The rule-based relationship extraction model proposed in this thesis managed to extract nearly half of the relationships, reaching a weighted average F1-score of 25.3%, which is lower than expected. However, it must be considered that the output of the NER model played a large role. Had the performance of the NER model been higher, this model would have likely performed better as well. Furthermore, Bozyigit et al. has proven that rule-based approaches are a viable option in the process of automating UML class models from natural language [22].

This thesis has showed that a BERT model, in combination with a rule-based relationship extraction model, has a lot of potential for the tasks NER and RE and that given a large dataset,

it has good potential for further development.

The data used in this thesis was of sufficient size to showcase the potential of the approach, however, in order to reach better results, more data is necessary, since deep learning models require a lot of data to perform well [29]. This will likely improve the performance of the NER model and subsequently improve the performance of the RE model, because there will be less misclassified NER labels. In addition, more annotators can help guarantee more consistent annotations, which in turn make it easier for the BERT model to learn patterns. Moreover, this would make it easier to set up extraction rules, as there would be no inconsistencies between annotations. Another interesting consideration for further research would be utilizing syntactic sentence structures in the relation extraction rules to increase the performance. Furthermore, it would be interesting to see how this study can be extended to properly include more relations, like composition and aggregations, or entities, like constraints. Finally, it would be an interesting experiment to see what the performance would be if the rule-based RE model was replaced by a BERT model.

References

[1] Hatice Koç, Ali Mert Erdoğan, Yousef Barjakly, and Serhat Peker. Uml diagrams in software engineering research: a systematic literature review. *Multidisciplinary Digital Publishing Institute Proceedings*, 74(1):13, 2021.

[2] José A Cruz-Lemus, Ann Maes, Marcela Genero, Geert Poels, and Mario Piattini. The impact of structural complexity on the understandability of uml statechart diagrams. *Information Sciences*, 180(11):2209–2220, 2010.

[3] Dave R Stikkolorum, Truong Ho-Quang, and Michel RV Chaudron. Revealing students' uml class diagram modelling strategies with webuml and logviz. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 275–279. IEEE, 2015.

[4] Deva Kumar Deeptimahanti and Ratna Sanyal. Semi-automatic generation of uml models from natural language requirements. In *Proceedings of the 4th India Software Engineering Conference*, pages 165–174, 2011.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[6] Ruth Sara Aguilar-Savén. Business process modelling: Review and framework. *International Journal of production economics*, 90(2):129–149, 2004.

[7] Matthew Hause et al. The sysml modelling language. In *Fifteenth European Systems Engineering Conference*, volume 9, pages 1–12, 2006.

[8] Russ Miles and Kim Hamilton. *Learning UML 2.0: a pragmatic introduction to UML.* " O'Reilly Media, Inc.", 2006.

[9] Jing Li, Aixin Sun, Jianglei Han, and Chenliang Li. A survey on deep learning for named entity recognition. *IEEE Transactions on Knowledge and Data Engineering*, 34(1):50–70, 2020.

[10] Erik F Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050*, 2003.

[11] Named entity recognition and its types. https://www.shaip.com/blog/named-entity-recognition-and-its-types/. Accessed: 29-06-2023.

[12] Xu Han, Tianyu Gao, Yankai Lin, Hao Peng, Yaoliang Yang, Chaojun Xiao, Zhiyuan Liu, Peng Li, Maosong Sun, and Jie Zhou. More data, more relations, more context and more openness: A review and outlook for relation extraction. *arXiv preprint arXiv:2004.03186*, 2020.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[14] Yves Rigou and Ismaïl Khriss. A deep learning approach to uml class diagrams discovery from textual specifications of software systems. In *Intelligent Systems and Applications: Proceedings of the 2022 Intelligent Systems Conference (IntelliSys) Volume 2*, pages 706–725. Springer, 2022.

[15] Wahiba Ben Abdessalem Karaa, Zeineb Ben Azzouz, Aarti Singh, Nilanjan Dey, Amira S. Ashour, and Henda Ben Ghazala. Automatic builder of class diagram (abcd): an application of uml generation from functional requirements. *Software: Practice and Experience*, 46(11):1443–1458, 2016.

[16] The Stanford Natural Language Processing Group. The stanford parser.

[17] Markus Eberts and Adrian Ulges. Span-based joint entity and relation extraction with transformer pre-training. *arXiv preprint arXiv:1909.07755*, 2019.

[18] Bin Ji, Shasha Li, Jie Yu, Jun Ma, and Huijun Liu. Boosting span-based joint entity and relation extraction via squence tagging mechanism. *arXiv preprint arXiv:2105.10080*, 2021.

[19] Dan Roth and Wen-tau Yih. A linear programming formulation for global inference in natural language tasks. Technical report, Illinois Univ at Urbana-Champaign Dept of Computer Science, 2004.

[20] Yi Luan, Luheng He, Mari Ostendorf, and Hannaneh Hajishirzi. Multi-task identification of entities, relations, and coreference for scientific knowledge graph construction. *arXiv preprint arXiv:1808.09602*, 2018.

[21] Harsha Gurulingappa, Abdul Mateen Rajput, Angus Roberts, Juliane Fluck, Martin Hofmann-Apitius, and Luca Toldo. Development of a benchmark corpus to support the automatic extraction of drug-related adverse effects from medical case reports. *Journal of biomedical informatics*, 45(5):885–892, 2012.

[22] Fatma Bozyigit, Tolgahan Bardakci, AliReza Khalilipour, Moharram Challenger, Guus Ramackers, Onder Babur, and Michel R. V. Chaudron. Generating models from text using natural language processing: A benchmark dataset and experimental comparison of tools. *Draft*, 2023.

[23] KR Chowdhary. Natural language processing. *Fundamentals of artificial intelligence*, pages 603–649, 2020.

[24] Prodigy by spacy. https://prodi.gy/. Accessed: 20-06-2023.

[25] Huggingface. https://huggingface.co/. Accessed: 20-06-2023.

[26] Brendan O'Connor and Michael Heilman. Arkref: A rule-based coreference resolution system. *arXiv preprint arXiv:1310.1975*, 2013.

[27] TianTian Tang, G.J. Ramackers, and S.A. Raaijmakers. *From Natural Language to UML Class Models: An Automated Solution Using NLP to Assist Requirements Analysis*. PhD thesis, 2020.

[28] Weights & biases. https://wandb.ai/site. Accessed: 27-06-2023.

[29] Qingsong Wen, Tian Zhou, Chaoli Zhang, Weiqi Chen, Ziqing Ma, Junchi Yan, and Liang Sun. Transformers in time series: A survey, 2023.

A Appendix

A.1 Relation Extraction Code

Listing 1 contains the python code of the implementation of the rule-based RE method.

```
# Relation Extraction Rules
# In this file, raw tokens are combined into full entities
# based on the predicted NER labels.
# Then, the RE rules are applied

from data_generator import load_tokenizer
from constants import ID2LABEL, COREFERENCES


class rel_extractor:
    """
    Wrapper class for the rule-based RE method.
    Contains methods:
    - extract_relatiosn(input_ids, ner_predictions)
    - combine_entities(input_ids, predictions)
    - find_full_entity(label, index, tokenizer, prediction, tok, token)
    """

    def __init__(self):
        # Contains extracted relations
        self.relations = []

    def extract_relations(self, input_ids, ner_predictions):
        """
        Applies the RE rules on the combined entities.
        Returns the extracted relations.
        """

        # Combine broken up entities based on NER predictions
        all_entities, all_labels = self.combine_entities(input_ids, ner_predictions)

        for entities, labels in zip(all_entities, all_labels):
            relations_in_batch = []
            index = 0
            for entity, label in zip(entities, labels):
                # Apply heuristics

                # Do not use entities that are padding
                if entity == "[PAD]":
                    continue

                # Rule for extracting association relations
                if label == "ASSOCIATION":
```

```
                    skip_first = False

                    if index > 1:
                        i = index
                        if (
                            entities[i - 1] == "and"
                            or entities[i - 1] == "or"
                            or entities[i - 1] == ","
                        ) and labels[i - 2] != "ASSOCIATION":
                            skip_first = True

                        # Search for correct class on the left side
                        while i > 0 and entities[i - 1] != ".":
                            if labels[i - 1] == "CLASS":
                                if skip_first:
                                    i -= 1
                                    skip_first = False
                                    continue

                                # If "can be" is in front of association entity,
                                # classify as zero-or-more multiplicity
                                if (
                                    entities[index - 2] == "can"
                                    and entities[index - 1] == "be"
                                ):
                                    relations_in_batch.append(
                                        [entities[i - 1], entity, "ASSOCIATION*"]
                                    )
                                    break

                                # Check if last char is s for mulitplicity
                                elif entities[i - 1][-1] == "s":
                                    relations_in_batch.append(
                                        [entities[i - 1], entity, "ASSOCIATION1..*"]
                                    )
                                    break
                                else:
                                    relations_in_batch.append(
                                        [entities[i - 1], entity, "ASSOCIATION1"]
                                    )
                                    break
                            i -= 1

                        # Search for correct class on the right side
                        i = index
                        while i < len(labels) - 1 and entities[i + 1] != ".":
                            if labels[i + 1] == "CLASS":
                                # If the association entity is one of these
                                # words, it is likely zero-or-more
                                if (
                                    entity == "controls"
                                    or entity == "list"
                                    or entity == "score"
                                    or entity == "scores"
```

```
                                    or entity == "sends"
                                ):
                                    relations_in_batch.append(
                                        [entity, entities[i + 1], "ASSOCIATION*"]
                                    )
                                    break

                                # Check if last char is s for multiplicity
                                if entities[i + 1][-1] == "s":
                                    relations_in_batch.append(
                                        [entity, entities[i + 1], "ASSOCIATION1..*"]
                                    )
                                    break
                                else:
                                    relations_in_batch.append(
                                        [entity, entities[i + 1], "ASSOCIATION1"]
                                    )
                                    break
                            i += 1

                    # Rule for extracting subtype and aggregation relations
                    if label == "CLASS":
                        subtype = True
                        count = 0
                        i = index
                        potential_rels = []

                        # Check if this is a listing of 3 or more classes
                        while subtype and i < len(labels):
                            if labels[i + 1] == "CLASS":
                                potential_rels.append(entities[i + 1])
                                count += 1
                                i += 1
                            elif entities[i + 1] == "," or entities[i + 1] == "and":
                                i += 1
                                continue
                            elif count >= 3:
                                potential_rels.append(entity)
                                break
                            else:
                                subtype = False
                                potential_rels = []
                                count = 0

                        # If listing, add relations subtype or aggregation
                        # if the word in front of listing is "of"
                        if subtype and potential_rels:
                            aggregation = False
                            if index > 0:
                                i = index

                                # If the word before the listing is "of"
                                # classify as aggregation
                                if entities[index - 1] == "of":
```

```
                                aggregation = True

                            while i > 0 and entities[i - 1] != ".":
                                if labels[i - 1] == "CLASS":
                                    if i > 2 and entities[i - 3] == "in":
                                        i -= 1
                                        continue
                                    for ent in potential_rels:
                                        if aggregation:
                                            relations_in_batch.append(
                                                [entities[i - 1], ent, "AGGREGATION"]
                                            )
                                        else:
                                            relations_in_batch.append(
                                                [ent, entities[i - 1], "SUBTYPE"]
                                            )
                                    potential_rels = []
                                    break
                                i -= 1

                    # RE rule for attribute relations
                    if label == "ATTRIBUTE":
                        if index > 0:
                            i = index

                            # Search for correct class on the left side
                            while (
                                i > 0
                                and entities[i - 1] != "."
                                and entities[i - 1] != "!"
                                and entities[i - 1] != "?"
                            ):
                                if labels[i - 1] == "CLASS":
                                    if i > 2 and entities[i - 3] == "in":
                                        i -= 1
                                        continue
                                    relations_in_batch.append(
                                        [entity, entities[i - 1], "ATTRIBUTE"]
                                    )
                                    break
                                i -= 1

                    # RE rule for operation relations
                    if label == "OPERATION":
                        if index > 0:
                            i = index

                            # Search for correct class on the left side
                            while (
                                i > 0
                                and entities[i - 1] != "."
                                and entities[i - 1] != "!"
                                and entities[i - 1] != "?"
                            ):
```

```
                            if labels[i - 1] == "CLASS":
                                if i > 2 and entities[i - 3] == "in":
                                    i -= 1
                                    continue
                                relations_in_batch.append(
                                    [entities[i - 1], entity, "OPERATION"]
                                )
                                break
                            i -= 1

                    # RE rule for coreference resolution
                    if entity in COREFERENCES:
                        if index > 0:
                            i = index
                            first_class_found = False
                            while i > 0 and entities[i - 1] != ".":
                                if labels[i - 1] == "CLASS":
                                    if not first_class_found:
                                        first_class_found = True
                                        i -= 1
                                        continue
                                    if i > 2 and entities[i - 3] == "in":
                                        i -= 1
                                        continue
                                    relations_in_batch.append(
                                        [entity, entities[i - 1], "COREF"]
                                    )
                                    break
                                i -= 1

                    if (
                        entity == "of"
                        and labels[index - 1] == "CLASS"
                        and labels[index + 1] == "CLASS"
                    ):
                        relations_in_batch.append(
                            [entities[index - 1], entities[index + 1], "COMPOSITION"]
                        )

                    index += 1
                self.relations.append(relations_in_batch)
            return self.relations

    def combine_entities(self, input_ids, predictions):
        """
        Turns list of raw subtokens with their predictions
        into full entities, using the BIO tagging scheme.
        Returns lists of entities and list of corresponding
        labels.
        """

        tokenizer = load_tokenizer()
        all_entities = []
        all_labels = []
```

```
260
261         for token, prediction in zip(input_ids, predictions):
262             entities_in_batch = []
263             labels_in_batch = []
264             index = 0
265
266             # For each token, if it has a B tag,
267             # search for the full entity and combine.
268             for tok, pred in zip(token, prediction):
269                 if pred != -100 and tokenizer.convert_ids_to_tokens(tok)[0] != "#":
270                     if ID2LABEL[pred] == "O":
271                         entities_in_batch.append(
272                             tokenizer.convert_ids_to_tokens(tok)
273                         )
274                         labels_in_batch.append(ID2LABEL[pred])
275
276                     if ID2LABEL[pred] == "B-CLASS":
277                         entity = self.find_full_entity(
278                             "CLASS", index, tokenizer, prediction, tok, token
279                         )
280                         entities_in_batch.append(entity)
281                         labels_in_batch.append("CLASS")
282
283                     if ID2LABEL[pred] == "B-ATTRIBUTE":
284                         entity = self.find_full_entity(
285                             "ATTRIBUTE", index, tokenizer, prediction, tok, token
286                         )
287                         entities_in_batch.append(entity)
288                         labels_in_batch.append("ATTRIBUTE")
289
290                     if ID2LABEL[pred] == "B-ASSOCIATION":
291                         entity = self.find_full_entity(
292                             "ASSOCIATION", index, tokenizer, prediction, tok, token
293                         )
294                         entities_in_batch.append(entity)
295                         labels_in_batch.append("ASSOCIATION")
296
297                     if ID2LABEL[pred] == "B-SYSTEM":
298                         entity = self.find_full_entity(
299                             "SYSTEM", index, tokenizer, prediction, tok, token
300                         )
301                         entities_in_batch.append(entity)
302                         labels_in_batch.append("SYSTEM")
303
304                     if ID2LABEL[pred] == "B-OPERATION":
305                         entity = self.find_full_entity(
306                             "OPERATION", index, tokenizer, prediction, tok, token
307                         )
308                         entities_in_batch.append(entity)
309                         labels_in_batch.append("OPERATION")
310
311                 index += 1
312             all_entities.append(entities_in_batch)
313             all_labels.append(labels_in_batch)
```

```
        return all_entities, all_labels

    def find_full_entity(self, label, index, tokenizer, prediction, tok, token):
        """
        Look ahead in the NER predictions to combine
        tokens into entities, using the BIO tagging scheme.

        """
        entity = tokenizer.convert_ids_to_tokens(tok)

        # Edge case for 's associations
        if entity == "'":
            entity += tokenizer.convert_ids_to_tokens(token[index + 1])
            return entity

        # Check if next token is also part of the same entity
        i = index
        while (
            i < len(prediction) - 1
            and prediction[i + 1] != -100
            and (
                ID2LABEL[prediction[i + 1]] == f"I-{label}"
                or ID2LABEL[prediction[i + 1]] == f"B-{label}"
            )
        ):
            # Sometimes a word is split into subtokens
            # In this case, the subtokens also start with a B tag
            # even though it is part of the same entity
            # Subtokens can be recognized, because they start with "#"
            if (
                ID2LABEL[prediction[i + 1]] == f"B-{label}"
                and tokenizer.convert_ids_to_tokens(token[i + 1])[0] != "#"
            ):
                break
            # Add token to entity
            entity += " " + tokenizer.convert_ids_to_tokens(token[i + 1])
            i += 1

        return entity
```

Listing 1: Implementation of the relation extraction rules.