Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MARTIN HOFMANN

MASTER THESIS

TEXT MINING FOR RELATIONSHIP EXTRACTION

Submitted on May 18, 2017

Supervisors: Andreas Kaufmann, M. Sc.
Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, May 18, 2017

License

Erlangen, May 18, 2017

Abstract

Qualitative Data Analysis (QDA) methods are based on manual coding of texts. To extract a domain model from a text corpus using QDA, information has to be extracted and compiled into the domain model by hand. This is especially a problem for cases where large amounts of data have to be analyzed.

For this purpose, We present a relationship extraction approach based on Natural Language Processing. It automates the extraction of relationships between codes that were provided by the coder. This speeds up the analysis process and helps to uncover relationships the human coder might have missed.

Our method produces a graphical overview of relationships that were found to exist between codes. It is evaluated by comparison with previously generated models from existing Qualitative Data Analysis projects.

Contents

1 Introduction **1**
1.1 Original Thesis Goals . . . 1
1.2 Changes to Thesis Goals . . . 1

2 Research Chapter **2**
2.1 Introduction . . . 2
2.2 Related Work . . . 2
2.2.1 Relationship Extraction in Software Engineering . . . 3
2.2.2 Relationship Extraction in Medicine . . . 3
2.2.3 Conclusion . . . 4
2.3 Research Question . . . 4
2.4 Research Approach . . . 5
2.4.1 Introduction . . . 5
2.4.2 Data Model . . . 6
2.4.3 Input . . . 7
2.4.4 Relationship Extraction Techniques . . . 7
2.4.5 Inference of Relations in the Code System . . . 15
2.4.6 Code System Clean-Up . . . 15
2.4.7 Output . . . 17
2.5 Used Data Sources . . . 18
2.6 Research Results . . . 18
2.6.1 Clustering Algorithm . . . 19
2.6.2 Dependency Trees Rules . . . 19
2.6.3 Code System Relationships . . . 22
2.6.4 All Methods Combined . . . 23
2.7 Results Discussion . . . 23
2.7.1 Results and Limitations . . . 23
2.7.2 Further Work . . . 25
2.8 Conclusion . . . 26

3 Elaboration Chapter **27**
3.1 Finding Relationship Candidates . . . 27

3.1.1 Sentence Parsing 27
3.1.2 Clustering Algorithm 29
3.2 Matching Codes to Linguistic Entities 32
3.2.1 Look-Up Table Approach 32
3.2.2 Weighting with WordNet 33
3.2.3 Web Search Engines as Knowledge Bases 36
3.3 The Code System 39
3.3.1 A Code System Meta Model 39
3.3.2 Code Naming Conventions 40
3.3.3 Impulses for Future Research Work 42

Appendices **45**
Appendix A Overview over the Software Artifact 45
A.1 General Information 45
A.2 Architecture 45
A.3 Scripts 45
A.4 Parameters 46
Appendix B Bill of Materials 51

Glossary **53**

Acronyms **54**

References **55**

List of Figures

2.1 Overview over the NLP pipeline of the prototype 5
2.2 Example of a dependency tree parse 8
2.3 Illustration of the sliding window algorithm 9
2.4 Look-up process for a linguistic entity e leading to a set of codes C. 12
2.5 Relationships extracted by the clustering algorithm using different modes for relationship creation. 19
2.6 Effectiveness of the dependency tree algorithms. 20
2.7 Effectiveness of improvements to the sliding window algorithm for dependency trees. 21
2.8 Effectiveness of various values for the sliding window boost. . . . 21
2.9 Effectiveness of entity-code table lookup improvements. 22
2.10 Effectiveness of adding additional knowledge bases to the entity-code table. 23
2.11 Number of relationships extracted purely from the code system hierarchy (y-axis adjusted for readability). 24
2.12 Results of combining all three sources of relationships in the Qualitative Data Analysis (QDA) project. 25

3.1 Sequence of the clustering steps performed on a set of eight integers by recursively dividing. 31
3.2 Entity-code table structure 33
3.3 Example of the Shortest Ancestral Path algorithm 36
3.4 Mistaken PoS parse 41

List of Tables

2.1 Look-up methods in the entity-code table 13

3.1 Query options used for the Bing API. 38
3.2 Undirected relationships and their correlation with the *Label* meta data annotation of the used codes. 40
3.3 Parameters in the configuration file of the prototypical relationship extraction system. 47
3.4 Dependencies of the prototype implemented in the course of this thesis. 51

1 Introduction

1.1 Original Thesis Goals

The goal of this thesis is to implement a prototypical Natural Language Processing pipeline specifically for extracting relationships from qualitative data analysis projects. Given a qualitative data analysis project, the tool should be able to extract relations

1.2 Changes to Thesis Goals

The thesis goals were not changed.

2 Research Chapter

2.1 Introduction

QDA is a research technique often found in social studies. A key part of this method is the annotation of a text corpus with a set of codes that are structured in a hierarchical code system. The documents in the text corpus can for example be interviews or scientific papers. Based on the results of the annotation process, a theory can be formed.

As a result of this process, a domain model that shows the relationships of entities in the code system can be created. Although the annotation process and the subsequent investigation of the code system are supported by computer software such as MaxQDA, much manual work is involved.

The drawback of this method is the need for time-consuming manual extraction of relationships in the code system as well as the possibility of human error.

In this thesis, we create a prototypical implementation of a tool that can automatically extract relationship information from a QDA project that has been annotated with codings. The contributions of this work are:

- We explore the application of Natural Language Processing (NLP) relationship extraction techniques to a QDA data set.
- We identify different sources for relationships in the data set.
- We show opportunities for further research in this area.

2.2 Related Work

Relationship extraction has been a popular research topic in NLP. We have found applications to be mostly centered around two domains: The automatic extraction of domain models from natural-language requirements in software engineering, and the analysis of research documents and patient data in medicine.

2.2.1 Relationship Extraction in Software Engineering

A system to extract conceptual data models from user requirements was first proposed by Black (1987). They propose a requirements engineering tool that can automatically build entity-relationship (ER) models from natural language requirements. The requirements are obtained through a dialog with the domain expert.

With the rise of standardized software engineering techniques, extraction of model information from requirements gained traction in the 1990s. Ambriola and Gervasi (1997) show a complex, web-based environment for requirements engineering with the ability to build object diagrams. Their system works on Italian texts. A predefined glossary of verbs is used to parse requirements from natural-language texts. The glossary does however not include domain-specific terms, making the system domain-agnostic.

Omar, Hanna, and McKevitt (2004) propose a fixed set of heuristics that aim to extract ER models from text, including both entities and relationships. However, they evaluate their methods only on a restricted test corpus of short text snippets taken from problem statements in exams and textbooks.

Du and Metzler (2006) expand on the use of heuristics to extract possible relationships from a text corpus. Unlikely relationships are removed by using WordNet as a knowledge database for entity identification and the Google Web Search corpus.

The methods employed in the domain of software engineering all work on a restricted subset of natural language: Ideally, relationship information can be found in short sentences following a pre-defined pattern. This is also the case in newer work such as a paper by Robeer, Lucassen, van der Werf, Dalpiaz, and Brinkkemper (2016) where NLP is applied to user stories with the aim of creating a model that shows relationships between the entities mentioned in the user stories.

2.2.2 Relationship Extraction in Medicine

In medical science, large quantities of data are available regarding molecular reactions of diseases and medication in patients' bodies. Relationship extraction in this area often focuses on building knowledge bases focused on observed connections between biochemical agents or genes (Garten & Altman, 2009).

Another area of work is the extraction of findings from sets of patient records. In a study by Roberts, Gaizauskas, and Hepple (2008), oncology patient narratives were analyzed in such a way using a supervised machine learning system. In this study, the task of entity recognition was assumed to perform perfectly to ease evaluation of the system.

More recent studies focus on the combination of rule-based systems with machine learning approaches (Quan, Wang, & Ren, 2014), aided by the size of the available biomedical corpus.

For relationships to be identified correctly, entities must be recognized in the text so that they can be connected by a relationship. In the area of medical literature, this task is often simplified by the fact that protein and gene names follow a unique naming scheme.

2.2.3 Conclusion

In contrast to the studies shown above, where the names of entities taking part in the relationship are extracted directly from the text corpus, we need to find relationships in a fixed set of entities from the code system, which do not necessarily match the entities in the text corpus. However, we have the advantage of using the code system structure of the QDA project as additional data source. Due to our limited data set, we decided to rely on a rule-based approach.

2.3 Research Question

We investigate the use of NLP relationship extraction techniques to extract relationships from a QDA project with the goal of automatically creating a domain model.

Additionally, we are interested which aspects of a QDA project contain information about relationships. We analyze a QDA project and identify different areas of the project that contain relationship information, and the methods which are suitable to retrieve relationship information. Accordingly, we implement a prototype to evaluate the efficiency of these methods.

We evaluate the accuracy of our methods by comparing the relationships to domain models that were generated by human coders. Based on our results, we identify opportunities for further research in the area.

2.4 Research Approach

2.4.1 Introduction

The program created in the course of this thesis is written in Java and makes use of the Stanford CoreNLP library to perform various NLP tasks. As input, it accepts a MaxQDA file and a directory of documents which may be in RTF or PDF format.

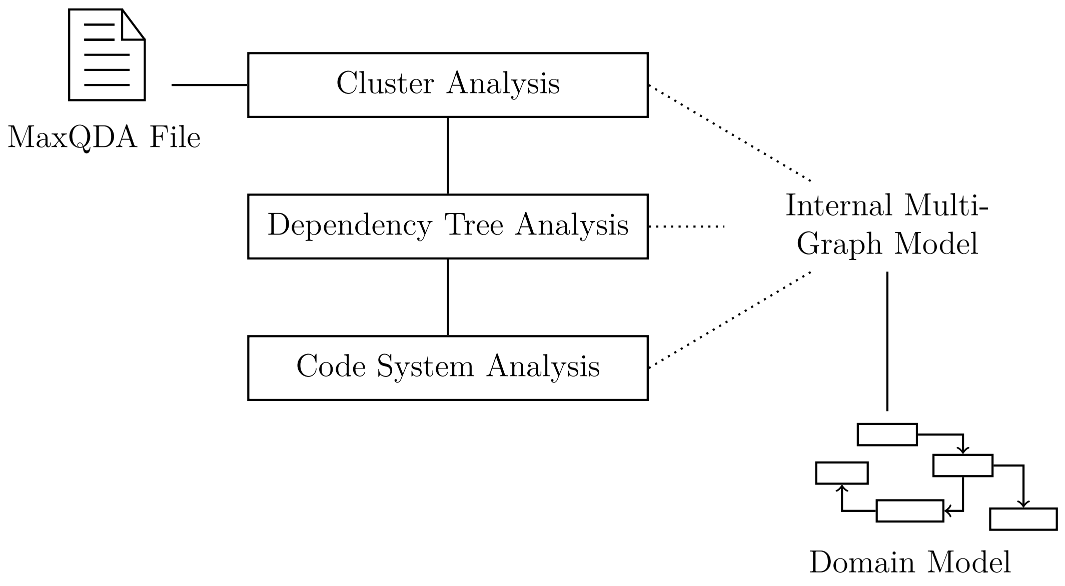


Figure 2.1: Overview over the steps taken to extract a domain model from a QDA project.

A mixed multi-graph is used internally to store information about possible relationships between codes in its edges. We will refer to each such edge as a relationship candidate. The edges are weighted, which allows to adjust the likeliness of a specific relationship during the run of NLP pipeline.

In total, three different aspects of the QDA project are used to extract relationship information:

- The locality of the codings in the documents is used via cluster analysis. The rationale behind this analysis is that codes that occur in close neighborhood in the texts are likely to be related.
- The text corpus is taken as a principal source for relationships. The NLP pipeline identifies relationships between nouns in the texts using dependency tree parses. Based on whether a noun taking part in the relationship is located in a coded text segment (and therefore being related to a code),

and co-occurrence of nouns and codes in the text corpus, relationship information for the code system is inferred. This creates a basic model of possible relationship candidates in the code system.

- Finally, the code system that is contained in the QDA project is used as a source for relationships. Here, structural information such as child-parent relationships in the code system tree is exploited. Code names are also analyzed for relationships.

After building the graph of relationship candidates, codes that are not suitable for the model are removed from the code system. The mixed multi-graph is then transformed to a mixed graph (without any parallel edges). In this step, the most promising relationship candidates are chosen.

Finally, a graph of found relationships between code system entities is produced as a simple “boxes-and-lines” figure in PDF format.

2.4.2 Data Model

The internal representation of all relationship candidates is stored as a mixed multi-graph $G = (V, E, A)$. In a mixed graph, both undirected edges (set E) as well as directed edges (set A) are possible.

In our case, the set of vertexes V corresponds to the codes in the code system. The directed component of G is a multi-graph, so that multiple edges between adjoining nodes are possible. Each edge $a \in A$ between nodes $v_1, v_2 \in V$ is a quadruple

$$a = (v_1, d, O, v_2)$$

with d being a string describing the relationship between the edges. Often, these strings are verb phrases such as “depend on”.

O is a set used to store the information source that backs this relationship. The set of values maps to the possible origins of relationships:

$$\begin{aligned} O \subseteq \{&\text{CLUSTER}, \text{DEPENDENCY\_TREE}, \\ &\text{CODE\_SYSTEM\_NAMING}, \text{CODE\_SYSTEM\_HIERARCHY}\} \end{aligned}$$

The clustering algorithm for relationship extraction is detailed in Table 2.4.4 and dependency trees are explained in subsection 2.4.4. For use of the code system naming conventions and the code system hierarchy as relationship source, refer to subsection 2.4.6 and subsection 2.4.5 respectively.

Additionally, a weight value w is stored for each edge:

$$w_{\text{dir}} : A \to \mathbb{R}$$

This edge weight is used to describe the level of importance of the edge and is modified through factors such as the occurrence count of this relationship instance in the text corpus. The use of a multi-graph allows to store multiple relationship candidates for each pair of vertexes. During the run of the NLP pipeline, the weight value w_{dir} can be incremented in case supporting evidence for the correctness of a certain relationship is found.

The undirected component E of G is used to store possible undirected relationships ("associations") between pairs of code words that can be created by the coding cluster analysis. Contrary to directed edges, multiple edges between nodes are not allowed here. Also, there is no description affixed to undirected edges. However, similar to directed edges, a weight value w_{undir} is stored for each undirected relationship in E. This allows to extract the most promising relationships.

2.4.3 Input

The prototypical application written for this thesis accepts files created by MaxQDA version 12 as input. Each MaxQDA file contains the code system hierarchy and coding annotations for documents. The documents are supplied separately either in PDF or RTF format.

2.4.4 Relationship Extraction Techniques

The NLP pipeline applies three main techniques for extracting potential relationships: A rule-based dependency tree parser on sentence level extracts relationships from the text corpus based on grammatical relations of words in sentences. A clustering algorithm uses the location and grouping of codings in documents as a source for relationships. Additionally, the structure of the code system itself is used to create relationships between ancestors in the code system.

Dependency Trees

Dependencies describe the grammatical relation between different entities in a sentence. A dependency relation is classified by an origin, a description, and a target. In this way a so called semantic tree can be constructed for each sentence.

For this thesis, we use the Enhanced++ dependency algorithm shipped with Stanford CoreNLP as described by Schuster and Manning (2016). An example for such a dependency tree can be seen in Figure 2.2. The verb "depends" is the root of the dependency tree. It is attached to the subject of the sentence by the

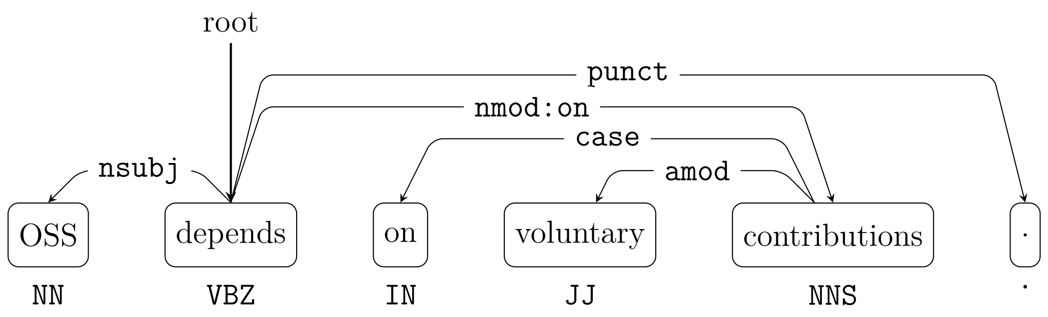


Figure 2.2: Example dependency tree parse for the sentence "OSS depends on voluntary contributions.". Dependency labels are attached to their respective edge. The part of speech (PoS) tags are displayed below the respective words of the sentence.

`nsubj` tag. The modifier "contributions" can be found by following the `nmod` edge in the dependency tree.

Based on the dependency trees, a rule-based approach is employed to extract potential relationships from sentences. There are two modes to determine the sentences that are analyzed: The *basic* algorithm that uses all sentences in the text corpus, and the *sliding window* algorithm as an improvement that narrows down the set of sentences to scan.

The sliding window filters the text corpus for sentences with a valid coding annotation. This way, sentences without a coding – that have been deemed irrelevant by the human coder – are removed. In addition, we allow a configurable number of sentences before and after each sentence with a coding annotation to be accepted into the set of valid sentences. These look-ahead and look-behind parameters allow the text surrounding the coding to be treated as a paragraph that is topically related to the sentence that contains the coding.

Figure 2.3 shows an example of the sliding window. Within the blind text one word has been annotated with a coding, shown red. The sentence containing the coding is accepted by the sliding window algorithm as overlap. Additionally, one sentence before and two sentences after the overlap are extracted as look-behind/look-ahead, leading to a total number of four sentences that will be scanned by the dependency tree rules.

We also use the sliding window algorithm to increase the weight of the code that is attached to the coding annotation when finding relations in the code system: Codes that are directly attached to the sentence containing a statement about a relation are more likely to take part in that relation as either subject or object.

For each sentence selected by the sliding window filter, we retrieve its dependency tree. Pattern matching is then used on the set of dependency tree rules. After

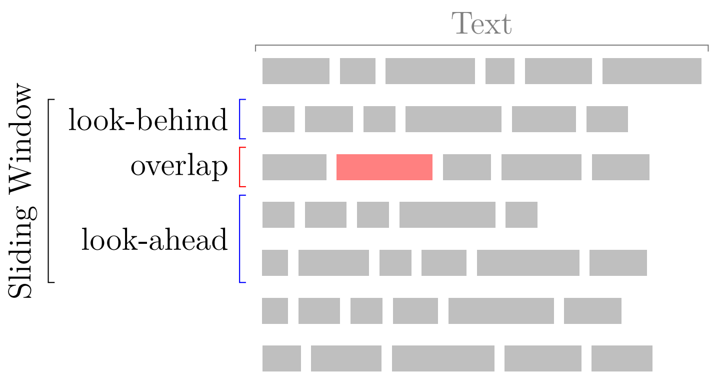


Figure 2.3: Example for the sliding window algorithm with a look-ahead of two sentences and look-behind of one sentence. One sentence in this part of text contains a coding annotation (highlighted red). A total of four sentences are within the sliding window.

the first matching rule, the search for further matches is canceled.

For the example sentence shown in Figure 2.2, a rule that identifies the subject using the `nsubj` and the object using the `nmod` tag is suitable. This results in the dependency

$$\text{OSS} \xrightarrow{\texttt{depends\_on}} \text{contributor}$$

by using the preposition "on" in combination with the verb as description of the relationship.

We have identified several more patterns of dependencies frequently correlating with meaningful relationships within a semantic graph. Each pattern starts at the root of a dependency tree. The patterns can be separated into to two groups: Patterns with a verb as their root node and a pattern to match sentences with an adverb as their root node.

In the following section, names of individual dependencies follow the *Universal Dependencies* standard as described by Silveira et al. (2014).

We have identified the following patterns:

Nsubj-Dobj-Rule The *nsubj* dependency points to the subject of the clause. The *dobj* is the direct object of the sentence. Therefore, this is the most simple statement-type of sentences.

Nsubj-Nmod-Rule This rule links a subject using the *nsubj* relationship to a noun phrase that extends the meaning of a verb using the *nmod*. This noun phrase is not an object of the sentence, but a *n*oun *mod*ifier. As the verb links the subject to this noun phrase, we treat the noun phrase as object for our relationships, although this is grammatically no object relation.

Nmod-Nsubjpass-Rule In passive statements, the order of the noun phrases is reversed: *nsubjpass* points to the logical object of the verbal clause, *nmod* is used for its subject.

Nsubj-Dobj-Linked-Verbs-Rule This rule is used to extract relationships from sentences that connect multiple verbal clauses with conjugations. As an example, consider the following sentence:

> Hence, most modern corporations of today heavily utilize and often develop software systems or their customizations.

Here, the root of the semantic graph *utilize* is connected to the verb *develop* with a conjugation. To gather the objects of the relationship, this rule follows conjugations determined by the *conj* dependency.

Ccomp-Sentence-Rule Some statements themselves contain a dependent clause with its own subject or object. From the root of the semantic graph – the verb of the statement – a *ccomp* dependency is used to identify the dependent clause. Consider the following sentence:

> Studies show that contributors develop Open Source systems in a joint effort.

Here, the real information is not the statement about studies, but the message about contributors in Open Source software.

AdverbialRoot-Nsub-Nmod-Rule This rule matches sentences that have an adverb as root node of the dependency tree, contrary to the default verb root.

If both object and subject of such a relation are nouns, we add this relationship to the set of relationship candidates. If at least one of the entities participating in the relationship is not a noun, we attempt to resolve this instance to a noun. For this purpose, we use the statistical coreference resolution system contained in Stanford CoreNLP as described by Clark and Manning (2015).

Relationships generated in such a way are described by the verb connecting the subject and object in the semantic graph.

The rule-based approach used here works well for finding relationships that are encoded in explicit statements. Additionally, it allows to find a description for a relationship by using the verb used explicitly in the statement. A limitation of this approach is however the use of complicated sentence structures or implicit meaning in sentences.

Matching of Dependency Tree Relationships to Codes

The dependency tree rules described in the previous section do not extract relationships between two codes directly. Instead, a relationship instance is identified using the nouns of the text occurrence it was taken from, which we will refer to as *linguistic entitites*. Therefore, an additional step is necessary to infer relationships between the codes of the code system.

In the software we implemented, this step is based on a look-up table that matches linguistic entities to the corresponding codes. The table is populated before the dependency tree run. Each key, representing a linguistic entity, maps to multiple possible codes that may be used for this entity.

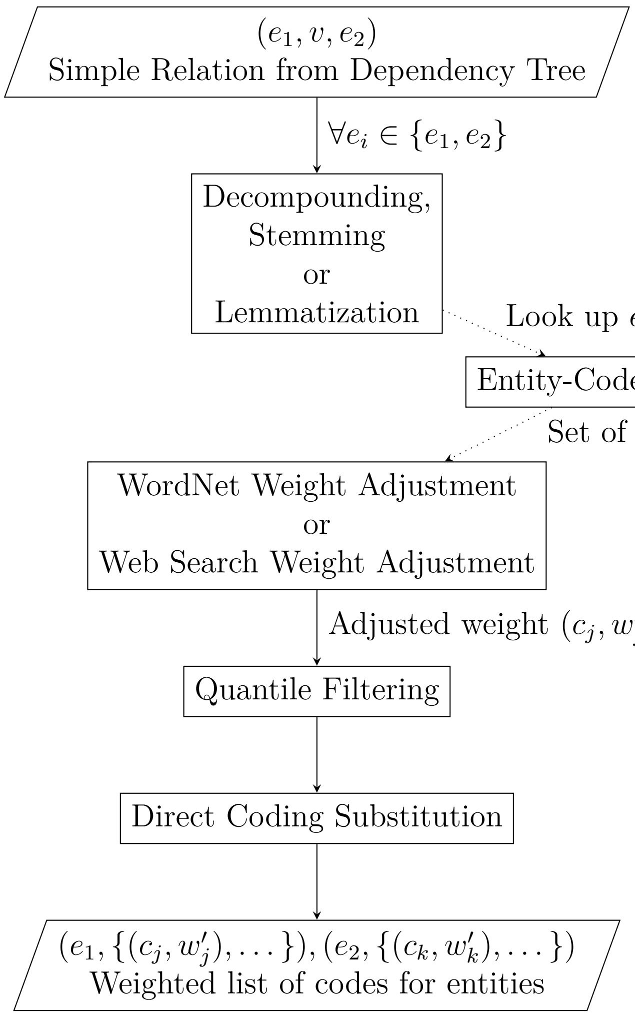
Building the Entity-Code Table To generate the table, we identify linguistic entities in the text corpus by their Penn Treebank Tag (Marcus, Marcinkiewicz, & Santorini, 1993). We identify nouns using a generalized PoS tag, taking only the first two characters of the tag into consideration. This way, all possible noun tags (NN, NNS, NNP, ...) are reduced to NN. Compound nouns are identified by using the dependency tree parse in combination with the dependency type `compound`.

If the noun in question is annotated with a coding, it is added to the look-up table with the corresponding code. To give an indication of the applicability of a certain code for a linguistic entity, a frequency counter is stored for every pair of entity and code.

To expand our knowledge of entities matching a code we use the so called *doubly-anchored pattern* (Kozareva, Riloff, & Hovy, 2008). It uses a web search engine to find additional instances of a concept, using the internet as a corpus. To find these new entities, a web query containing both the concept (in our case the code) and a seed instance (in our case the entity extracted from the text) are used as search term. New instances are then extracted from the snippets of the search results by pattern matching.

Look-Up Process Within our application, the look-up process is configurable. It can be performed in varying grades of detail. The look-up expects a linguistic entity e as input. It produces a list of codes, each code c having an attached weight that corresponds to the frequency count of e and c during the creation of the look-up table.

Configuration options for the look-up process contain exact matching of entity names, stemming or lemmatizing them. Each of the options can be combined with the right-hand-head-rule as explained by Williams (1981). This rule allows to split



(e_1, v, e_2)
Simple Relation from Dependency Tree
$\forall e_i \in \{e_1, e_2\}$
Decompounding, Stemming or Lemmatization
Look up e_i
Entity-Code Table
Set of weighted codes (c_j, w_j)
WordNet Weight Adjustment or Web Search Weight Adjustment
Adjusted weight (c_j, w'_j)
Quantile Filtering
Direct Coding Substitution
$(e_1, \{(c_j, w'_j), \dots\}), (e_2, \{(c_k, w'_k), \dots\})$
Weighted list of codes for entities

Figure 2.4: Look-up process for a linguistic entity e leading to a set of codes C.

compound nouns to generalize the meaning of the noun phrase. Table 2.1 shows the effect of the options applied to the example entity "software development processes" (note the plural form of "processes").

After the look-up, the weight of each code for an entity is adjusted by their distance in the WordNet database. To calculate this distance we experimented with two different measures: We use a strict hypernym/hyponym[1] distance or the Shortest Ancestral Path, the path distance of two words over a common ancestor in the WordNet tree.

In addition to WordNet, we offer the possibility of using a Web Search engine for weight adjustments. As we retrieve candidate codes for each entity, we send

[1]A hypernym is a more generic term for a specific term (the hyponym), such as "house" to "castle".

	Without RHHR	With RHHR
Lemmatization	“software development process”	“software”␣“development” ␣“process”
Stemming	“softwar develop proc”	“softwar”␣“develop” ␣“proc”
No transformation	“software development processes”	“software”␣“development” ␣“processes”

Table 2.1: Example of looking up the entity “software development processes” with different methods. ␣ signifies the location of possible split of a compound noun.

a search query for each pairing of entity and code. We then use the result count of the search engine to adjust the weights. Pairings with a low result count are rated lower compared to entity-code pairs with a high result count. To adjust for unlikely entity words (which are not often found on the web), we calculate the weights locally, and compare them only in regards to the specific entity.

In addition to linguistic modifications to the search term, we allow a quantile-based filtering. It restricts the result set of codes for a certain entity to those codes which are in the upper n percent of all entity-code mappings. Using this step, spurious correlations of codes for certain entities are eliminated.

If no code could be found by the entity-code table, we substitute the entity with the code attached to the text that is currently overlapped by the sliding window. If multiple codes have been found in the sliding window, we add all those codes, as they are all valid candidates for relationships. This information is retrieved by the sliding window algorithm as described in the previous section.

Coding Clusters

The dependency tree rule approach is suitable for semantically well-formed statements in natural language. Some documents, however, do not use such language. A notable example would be the use of presentation slides. Within our data sets, several slide sets contained information on specific topics which could not be found elsewhere. Within these slides, all textual information is encoded in bullet point lists. However, we do not have structured information about the hierarchy of the bullet point items. This is caused by using PDF as document format, which is mainly a presentation data format and focused on transporting layout information. Structure information is optional in PDFs (Withington, 2011).

It is however apparent that the location of codings in the document may be used to infer relationships in the affected code words: Code words in close neighborhood are related. In contrast to the usage of dependency tree rules and their ability to parse the verbs in sentences, the name of relationships based only on locality information cannot be determined automatically. Regardless, this information can still be useful in further refining a model in an additional step after the automatic extraction of relationships by dependency trees.

The problem of finding coded text segments which are close to each other breaks down to a one-dimensional clustering problem on a set of integers. To form this set of integers, the starting offset of each coding in the document is used as identifier for coding. A neighboring group of codings in this set can then be seen as a cluster.

Traditional clustering algorithms, such as the k-means algorithm, are often optimized for their use in multiple dimensions and require the number of clusters to be found as parameter. Especially the latter is not the case for our problem: The number of coding clusters in a document cannot be determined a priori.

Therefore, a custom algorithm based on hierarchical descend is used. The algorithm exploits the fact that a total ordering of the input array can be achieved due to it being composed exclusively of integers. A termination condition based on the local density of a cluster is used to cancel the recursive descend.

For each document, a set of clusters of codings is generated. This cluster information must then be incorporated into the relationship model. Relationships can be added either as being directed or as being undirected. We experimented with three different modes of adding relationships from one cluster, influencing both direction and number of relationships:

First-to-Children In this mode, a relationship is added between the first item of the cluster (e.g. the item with the smallest offset in the document) and each of its successors. This way, a hierarchical relation between the first mentioned code and its successors can be modeled: The first code in the cluster is related to all its successors.

Children-to-First This mode adds relationships to the first item in the cluster from all its successors. It only differs from the *First-to-Children* mode for directed relationships.

Cartesian In this mode, a relationship is added between each of the items in the cluster. Using the cartesian product of all codewords used in the cluster stresses the relatedness of items in a cluster.

2.4.5 Inference of Relations in the Code System

Code System Ancestors

The code system itself represents a hierarchy of concepts. This hierarchy is created by the coder and contains a high information density.

The structure of the code system depends on the coding standard used by the coder. However, children often represent a specialization of their parent node, e.g. a "is-a" relationship exists between a child and its parent node.

To harvest this information, a step was included in our NLP pipeline that adds relationships based on the code system hierarchy. Such relationships are assigned an origin tag of CODE_SYSTEM_HIERARCHY. In our prototype, relationship candidates from this source are weighted particularly high, because they are created by the coder and therefore influenced by their a-priori knowledge.

2.4.6 Code System Clean-Up

After potential relationships between codes have been added gathered, we remove codes that are not suitable for use in the model. A code may not be suitable if it describes an action instead of an entity or if it was not intended to be used in the model by the coder. In the latter case, a naming pattern or the depth of the code in the code system tree is used. If a code is removed, relationships that this code is a part of are redirected to the parent code in the code system hierarchy. This way, we ensure that no relationship information gets lost when removing codes.

The need for such an additional processing step depends on the structure of the code system that is used. As there is no default structure for code systems, the applicability of the methods presented in this section varies between different projects.

Removal of "Action Codes"

Some codes in the code system do not represent entities, but instead represent an action. This action may, in turn, relate to another entity in the code system. Such codes are easily identifiable by having a leading verb in their name and are called "action codes" in this thesis.

An example for such a code is the code `find Stakeholders` in the Inner Source data set by Salow (2016). This code does not represent an entity. Instead, it describes a relationship between its parent code and another entity "stakeholders".

To eliminate such codes from the code system, a depth-first traversal of the code system hierarchy is performed. If a code name does match the pattern `<verb> <object>`, it is eligible for removal. If another code can be found that corresponds to the object due to name equality, a relationship is added in the model between the parent of the code to be removed and the found object. This relationship is named after the verb of the code.

Relationships whose edges are incident to a removed code is redirected to the parent code word. The depth-first traversal algorithm also ensures that a cascading removal of codes starting from the bottom of the code system hierarchy is possible.

Removal of Codes Deep in the Code System Hierarchy

Code systems are trees of codes. We found, that in our data set codes with a high depth in the code system are specializations of their parent codes[2]. Code systems with a fine-grained code system often have only a small number of coded text segments attached to each individual code. This means that meaningful relationships cannot be extracted from them, as too few relationship candidates corresponding to them are found.

In such cases, codes very deep in the code system hierarchy are removed and their relationships are moved to their parent codes. These parent codes represent more general concepts and gather more relationship candidates by using this method.

The heuristic of using the depth of codes is superseded by the method described in the next section. Its advantage is that it can be used regardless of the naming conventions of codes.

Removal of Codes by Pattern

In some cases the set of codes that should be removed to create the final relationship model cannot be filtered finely enough by the methods in the previous sections. Therefore, we propose an additional filter in this section. This method allows to remove codes that do not match a pattern provided by a regular expression. The applicability of this method is however restricted by the facts that

(a) a suitable filter condition must be derivable from the naming of the codes, and

(b) the regular expression used for filtering must be provided by the user.

[2]This rule is only an approximation and can not be generalized to other projects. Ideally, these relationships should be encoded in the coding guidelines before of annotating the text corpus.

As an example, our evaluation data set uses a numerical identifier in the code name for all the entities that can be used in the model. By constructing a suitable regular expression, invalid codes can be removed efficiently from the model.

Redirection of relationships from and to codes that need to be removed is performed in as described in subsection 2.4.6.

2.4.7 Output

Reduction of Model to Graph

The in-memory representation of the data model that has been created by the information extraction steps of the program cannot be used as a graphical representation. It consists of a mixed multigraph with weighted edges, where every edge is a potential relationship. In the final relationship model, only at maximum one edge should exist between two codes. This improves legibility and allows the model to be understood by the end-user.

Therefore, the internal relationship model must be transformed from a mixed multigraph to a mixed graph. The transformation should be targeted at optimizing two main criteria:

1. *Suitable edges* should be selected. In the internal data model, each relationship candidate is weighted. Only the most likely relationships should be chosen for the final model.
2. The *graph density* of the result should be kept relatively low. Too dense graphs are prone to poor graphical representations and may be illegible to the end user.

For undirected edges that can be created by the coding cluster analysis method, we sort all undirected edges in the model by descending order by their weight. From this list we then take the top m edges, as these are the edges with the greatest confidence.

For directed edges, the algorithm minimizes the problem of “shadowing” directed edges. By regarding every possible directed edge (with its name and its weight) on its own, and selecting the top n edges by weight, some important edges will not be identified. A relationship candidate e_1 between two nodes, where no other parallel candidate relationships have been found, can be chosen instead of a relationship e_2 with a lower weight, but with multiple parallel relationships candidates between its adjacent nodes. The sum of the weights of all parallel relationship candidates similar to e_2 is higher than that of only e_1, so that a relationship between its nodes is more likely.

Therefore, the algorithm uses the sum of all possible edges between two adjacent nodes to determine which codes need to be related to each other in the model. The name of the relationship is determined by the single highest weighted relationship candidate between the two nodes.

Additionally, the model to graph algorithm for the mixed multi-graph model prevents overlapping directed relationships in the opposite direction between two nodes. Instead, only the edge with the largest weight regardless of direction is chosen.

Generation of Graph Figure

The output of the program is a directed graph which allows only single edges between nodes. The graph is serialized to the GraphVIZ format (Gansner & North, 2000) and a PDF graphic with layout is created with an automatic graph layouting algorithm.

2.5 Used Data Sources

The data source used to evaluate and develop the relation extraction was a the QDA project "Inner Source". Salow (2016) created a finished domain model in the form of a UML diagram that was used as the gold standard to evaluate our work.

We used WordNet as described in (Fellbaum, 1998) to measure the relatedness of words. WordNet is a lexical database of that links words by their meaning.

In addition to WordNet, we also queried the Microsoft Cognitive Services Bing Web Search API as described at ("Microsoft Cognitive Services – Bing Web Search API," n.d.).

2.6 Research Results

Our prototype is highly configurable. In this section, we examine the individual parts of the algorithm and their configuration options and then present general results for the accuracy of our method.

One limitation of our method is the need to specify the number of relationships that should be extracted from the QDA project as a parameter. Therefore, precision and recall scores often used in benchmarking information retrieval methods are the same. For this reason, we give the accuracy of our method as the number of correctly identified directed relationships. A total of 46 such directed relationships were present in the gold standard.

2.6.1 Clustering Algorithm

The clustering method allows us to create relationships based on the location of codings in the documents. We implemented three modes to add relationships to the internal data model. Their results are shown in Figure 2.5. Adding relationships from the first code in the cluster to its successors was least successful, as two thirds of those relationships were reversely orientated, while 13 % of relationships could be identified. Adding relationships from the successors to the first code was more successful, with only one sixth of the relationships being reversed while the number of relationships stayed the same. Adding the Cartesian Product of the code set of a cluster as relationships improved accuracy minimally by one additional relationship being found to a total of 15 % correctly identified relationships.

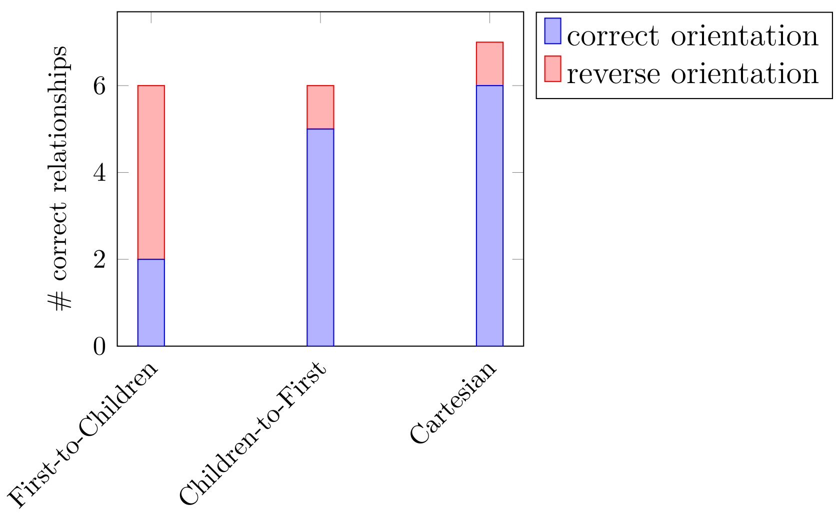


Figure 2.5: Relationships extracted by the clustering algorithm using different modes for relationship creation.

2.6.2 Dependency Trees Rules

Dependency Tree Rule Algorithm

We implemented two algorithms to generate sets of sentences for the dependency tree rules. The *basic* algorithm scans every sentence in the text corpus, whereas the *sliding window* algorithm considers only sentences with an attached coding.

To evaluate the effectiveness of these algorithms, we measured the accuracy of each of these algorithms individually without clustering or code system relationships. However, codes that were not intended to be used in the domain model were deleted by the code filter. For these basic measurements, improvements to the sliding window algorithm (besides look-ahead/look-behind parameters) were disabled.

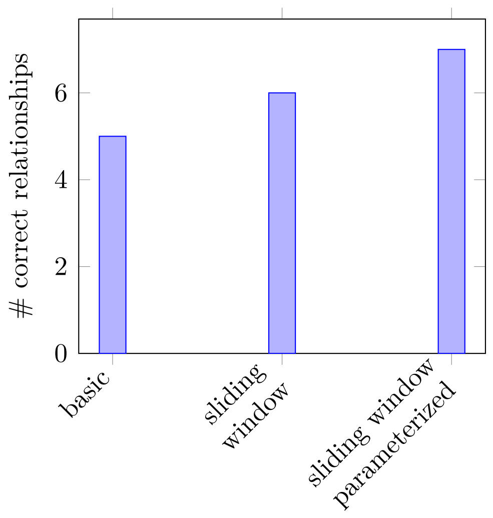


Figure 2.6: Effectiveness of the dependency tree algorithms.

Figure 2.6 shows the effectiveness of the algorithm configurations. Using the sliding window algorithm in its basic form yields a small benefit compared to the basic algorithm. We tried various parameters for the look-ahead/look-behind of the sliding window and found that parameterizing the sliding window with a look-ahead of eight sentences and no look-behind further improved the number of extracted correct relationships to a total of seven correctly identified sentences.

The sliding window algorithm contains two improvements: The code overlaying the current window can be weighted stronger than others when finding codes for relationships (*current code boosting*), and the current code can be used as a substitute for codes participating in relationships. The effect of these options is shown in Figure 2.7. We could not find evidence, that the code substitution aids in finding additional relationships. Preferring the code that is attached to the current sliding window however helped in finding two additional relationships (a 29 % improvement), although the orientation of these relationships was reversed.

In Figure 2.8, we experimented with various values for the current code boosts and found a weight boost of 8 to produce the best results. Lower values were not enough to surface all relationships, higher values did not produce any extra relationships.

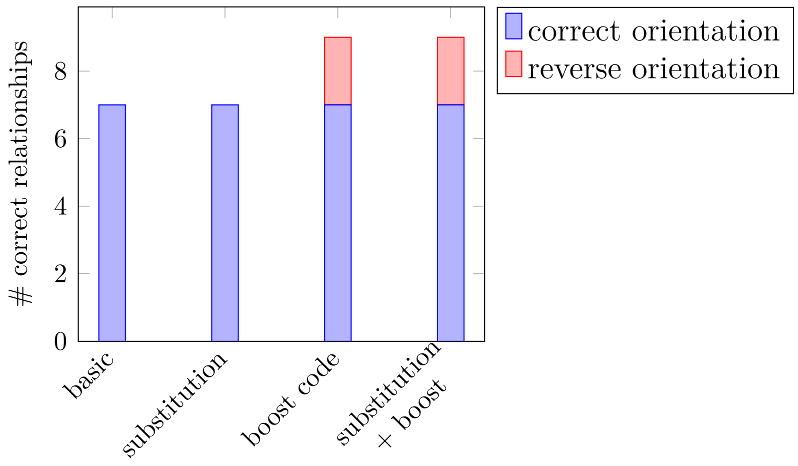


Figure 2.7: Effectiveness of improvements to the sliding window algorithm for dependency trees.

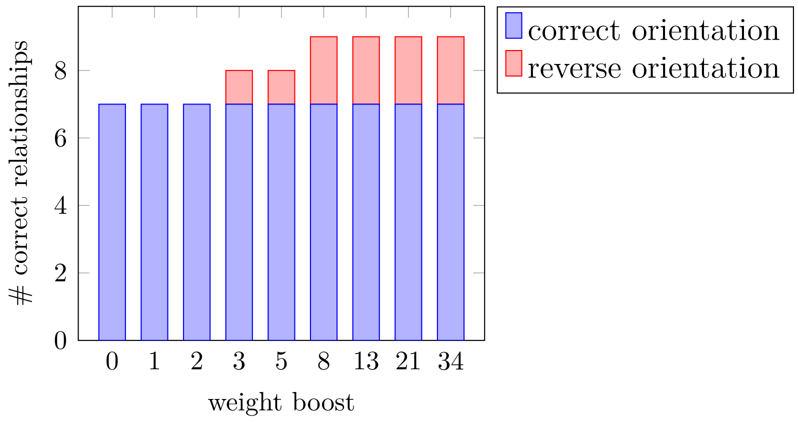


Figure 2.8: Effectiveness of various values for the sliding window boost.

Entity-Code Matching

Relationships identified by the dependency tree rules tree rules need to be transferred to the code system. For this purpose, we implemented a entity-code lookup table in our prototypical relationship extraction system.

To investigate possible improvements to the look-up process, we added techniques such as stemming and deconstruction of compound nouns for the keys of the table. The effectiveness of these methods is shown in Figure 2.9. Decompounding nouns using the RHHR rule resulted in a 14 % lower accuracy. Stemming decreased the accuracy of dependency tree relationships by 29 %, same as the combination of stemming and decompounding.

We added two additional knowledge bases to improve our results. WordNet is

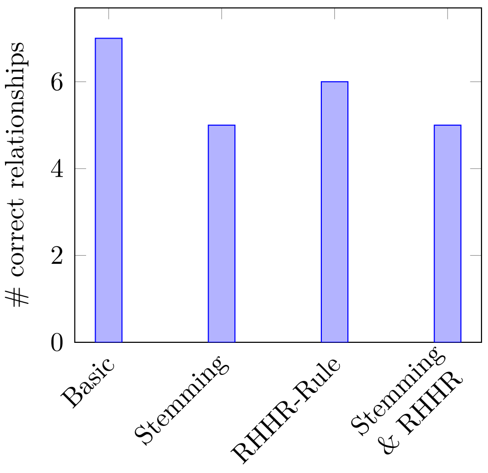


Figure 2.9: Effectiveness of entity-code table lookup improvements.

used to rate the accuracy of entity-code pairings returned from the table. Web search results can also be used in the same way. Additionally, we use the web search to generate more entries for our entity-code table via usage of the doubly-anchored pattern.

Figure 2.10 shows how these techniques influenced the results from the dependency tree results. By rating the quality of found code-entity pairs using WordNet, the total number of found relationships stayed the same, however no incorrectly orientated relationships are found any more. Adding additional entities to the entity-code table using a web search engine as knowledge base yielded no additional results. We also found that using the web query technique for weighting found pairs of code and entity decreased the number of correct results by 14 %.

2.6.3 Code System Relationships

Generating relationships from the code system alone was highly successful. Figure 2.11 shows the number of relationships generated by the code system analysis and its improvements through WordNet application. Just by running relationship extraction on the code system, we successfully identified 31 out of 46 directed relationships (67 %).

Using WordNet to rate the quality of the code system relationships was unsuccessful when we chose to only use strict hypernym/hyponym relationships in WordNet. A small improvement of one additional relationship was found when using the Shortest Ancestral Path algorithm on the WordNet database. We found that combining both the Shortest Ancestral Path as well as the hypernym/hyponym measure reduced the number of correctly identified relationships to 30.

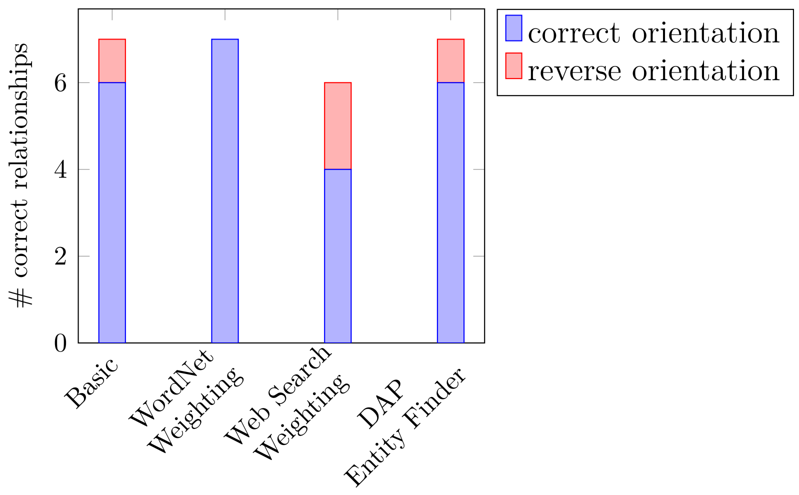


Figure 2.10: Effectiveness of adding additional knowledge bases to the entity-code table.

2.6.4 All Methods Combined

Our prototypical relationship extraction system combines the three data sources which we have evaluated in the previous sections. Figure 2.12 shows the result of bundling relationships from these data sources. Combining all three methods, the system identifies 16 relationships (15 correctly orientated, one reversed). Disabling the code system relationships reduces the number of relationships by 53 % to seven. Disabling only the clustering algorithm lead to a minimal decrease of one relationship. However, disabling the relationships from the dependency tree algorithm doubled the number of identified correct relationships up to 70 %.

2.7 Results Discussion

2.7.1 Results and Limitations

We were able to show that it is possible to extract a limited subset of relationships between the codes by using NLP techniques. However, relying on relationships encoded in the text alone proved not to be satisfactory.

Instead, we referred to additional data sources in the QDA project such as the code system hierarchy. Our final results in subsection 2.6.4 show that the quality of relationships generated from the dependency tree rules is poor; by disabling the dependency tree rules, the code system relationships surfaced. This leads to the

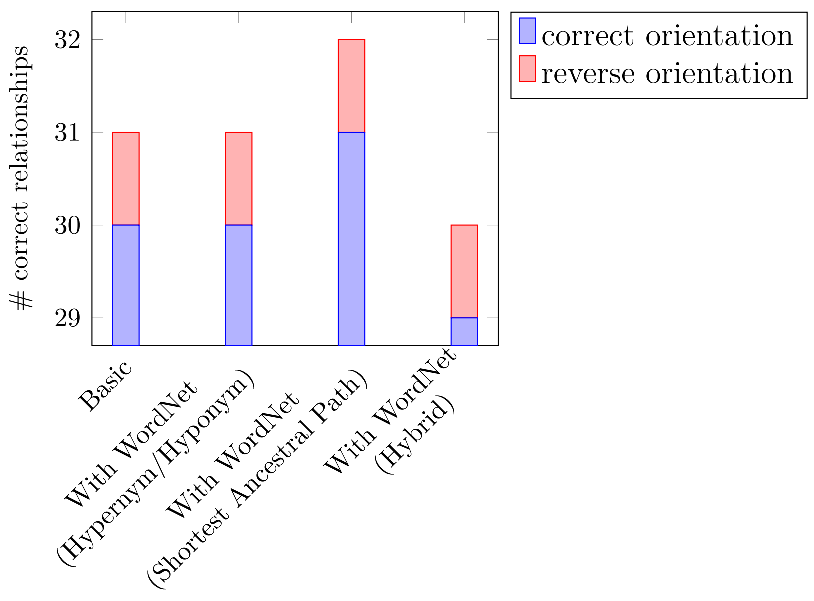


Figure 2.11: Number of relationships extracted purely from the code system hierarchy (y-axis adjusted for readability).

conclusion that the dependency tree relationships are of lower quality compared to the code system relationships.

Relying too strongly on relationships from the ocde systems contains the risk of relying on implicit assumptions that were encoded by the creator of the code system. Ideally, all relationships between codes should be visible straight from the content of the text corpus.

Similar criticism may be applied to the application of WordNet and Internet search engines. By using these data sources, we effectively incorporate knowledge beyond the contents of the QDA project into our application. However, when manually creating a domain model and during the coding process, the coders themselves are subject to a similar bias. Therefore, the closed world assumption is not feasible for the task at hand.

Regarding the creation of the domain model, we rely on a simple “boxes-and-lines” approach, due to the difficulty of extracting any valid relationships at all. It would be more appropriate to model the relationships with types and descriptions similar to Unified Modeling Language (UML) class diagrams (UML, 2015).

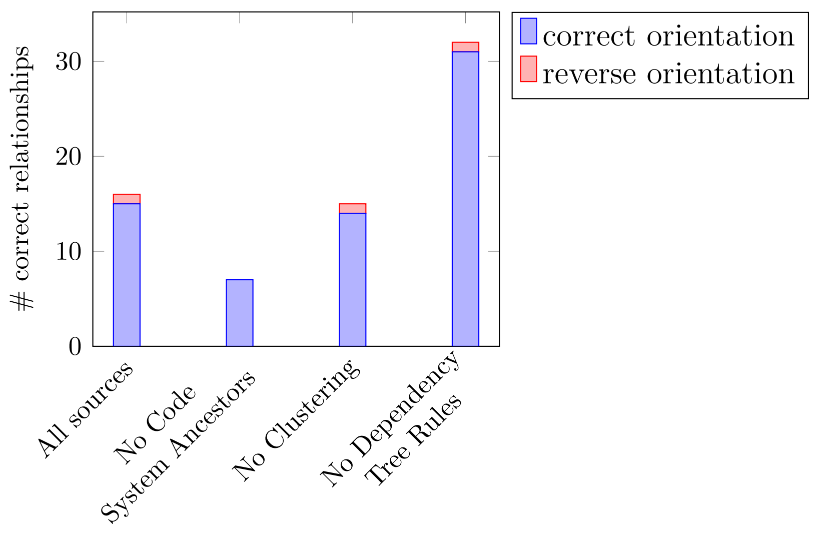


Figure 2.12: Results of combining all three sources of relationships in the QDA project.

2.7.2 Further Work

We suggest that two main aspects need to be addressed to improve relationship extraction from QDA projects:

1. A specialized coding guideline that makes QDA data more accessible to NLP methods should to be developed.
2. Other data sources should be taken into consideration.

Regarding the first aspect, several points should be addressed in such a coding guideline: For example, the codes in the code system that should be used in the domain model should be labelled. Additionally, only codes that stand for concepts and entities should be present in the code system. During the coding process, only short text segments – ideally only subjects and objects of statements – should be annotated with a code instead of whole paragraphs. This would greatly simplify identification of codes belonging to a specific relationship that is mentioned in the text.

Referring to the second aspect, additional data sources that could be used for finding relationships include ontologies or additional text corpora such as the Wikipedia. These data sources allow to simplify the task of matching entities to codes, similar to the way we already used WordNet in our work. Using larger knowledge bases would accurately reflect the existing knowledge a human coder creating a domain model brings into the process.

2.8 Conclusion

We show that it is possible to extract relationships from the contents of a QDA project. To do so, we identify three different data sources for relationships in the QDA project: The text corpus, locality of codings, and the code system. For text relationships we use a rule-based approach focused on dependency tree parses. Coding locality is regarded as one-dimensional clustering problem. The code system analysis is based on parent-child relationships in the code system tree.

While the code system relationships are mostly accurate with an accuracy of 67 %, relationships from the text corpus with 19 % accuracy and relationships from locality of codings with 15 % accuracy are less reliable. Especially the transfer of relationships from statements in the texts onto codes in the code system has been difficult, although we incorporated additional knowledge bases such as WordNet and web search into this step.

In conclusion, we identify two main areas for further research in this work: A stricter coding guideline adapted to the use of NLP in QDA should be developed and evaluated to improve the accuracy of found relationships, and additional knowledge bases should be investigated in regards to the transfer of text relationships to code system relationships.

3 Elaboration Chapter

3.1 Finding Relationship Candidates

3.1.1 Sentence Parsing

Adoption of Dependency Trees for Other Languages

Dependency trees show the grammatical relations between words in a sentence. These relations are stored in a tree data structure that links the words using the appropriate grammatical role. The grammatical relations are named in accordance to the Universal Dependencies standard. In our work, we use the dependency tree to extract knowledge using a rule-based approach.

By using Universal Dependencies, these rules are applicable to texts in multiple languages, as language-agnostic parsing is an aim of the Universal Dependencies initiative (Nivre et al., 2016). However, applications of Universal Dependencies have shown a tendency of lower accuracy in other languages such as German (Vulic & Korhonen, 2016).

While our approach fared reasonably well on our corpus consisting mainly of English research papers, QDA data sources often consists of spoken speech, for example in interviews. In research, spoken language has been found to differ from written language in regards to the Universal Dependencies annotations (Dobrovoljc & Nivre, 2016).

In our work, we investigated the applicability of our dependency parsing approach to other QDA data sets that were available to us. Of special interest was the data set from Kunz (2015) that investigates the creation of domain models in the area of human resources management.

We encountered several difficulties analyzing the sentences using Dependency Tree parses. Consider the following sentence:

Interviewer: Also das Gehalt ist an die Jobcluster gebunden?

> Frau ——: An die Kompetenzprofile... ganz stark gebunden in den meisten Unternehmen. [1]

This section of the interview links “Gehalt” (salary) and “Kompetenzprofil” (competence profile) in the domain model. However, this is not immediately obvious in spoken language:

- The question from the interviewer and the interviewee’s answer relate to the same topic (salary), however there is no explicit reference in the interviewee’s answer.
- The interviewee’s answer shows the use of ellipsis in natural language, which we frequently encountered in interviews.

Especially the use of ellipsis, and the related changes of topics within one sentence where problematic for dependency tree parses. Due to these problems, we decided to apply our approach only to written language.

Coreference Resolution

In a natural language text, not all entities of a sentence are identified by their name. Instead, in some cases, the subject or object of a statement are referred to via pronouns. To use our dependency tree algorithm, we need to find representative noun phrases for each pronoun, so that we can look up the codes that may be referred to in the statement via our entity-code table. The process of resolving pronouns in texts is called coreference resolution in NLP terminology.

The Stanford CoreNLP framework offers multiple distinct coreference systems, ranging from a *deterministic* rule-based approach over a *statistical* system up to a coreference system based on a *neural* net. The neural-network-based coreference resolution system offers the highest accuracy, followed by the statistical system. The deterministic resolution follows last in terms of accuracy.

Due to run-time considerations, we decided to use the statistical coreference resolution system as described by Clark and Manning (2015). It requires Stanford CoreNLP’s dependency parse information, which is already contained in our pipeline configuration to enable dependency tree rule information extraction.

Each occurrence of a reference to a specific entity in a text is called a *mention* of the entity. All mentions of this entity are linked in a so called coreference chain. When encountering an entity in a statement that is matched by a dependency tree rule, we resolve the entity as follows:

[1] Interviewer: So the salary is bound to a job cluster?
Mrs. ——: To the competence profile... strongly bound in most companies.

1. If the entity is a noun, it does not need to be resolved as we can feed it directly to the entity-code table. Instead, we return the noun directly. This is also the case for compound nouns where we return all words making up the noun.
2. If the entity is the preposition “we”, it too is not resolved. The rationale behind this rule is rooted in the fact that this preposition mostly stands for the speaker or author of the paper and tends to confuse the Stanford CoreNLP coreference system, so that it cannot be resolved correctly anyway.
3. Get the coreference chain for the preposition. If no coreference chain exists (e.g. the coreference system was unable to find another mention for this specific preposition), the algorithm fails.
4. Get the so called *representative mention* for the preposition. This is the best mention, and it should ideally be a noun. Similar to point 1, this noun is then expanded to include all its terms if it is a compound noun.

Resolution of compound nouns as mentioned above is necessary because the coreference chain only points to one part of a compound noun, not the whole entity. It is implemented using the dependency tree annotation `compound`. To resolve a compound noun, we follow this annotations in a chain starting from the noun identified by the coreference system.

3.1.2 Clustering Algorithm

Finding clusters of codings in a document breaks down to a one-dimensional clustering problem. The position of the coding in the document is given by its character offset. These positions can then be used as input to the clustering algorithm.

The problem at hand differs from other clustering problems in computer science which are often set in more than one dimension. Additionally, for these problems the numbers of clusters to identify by the algorithm has to be known a priori.

Both of these aspects are not present in our case. We can exploit the fact that coding offsets can be sorted, as they are simply a set of integers. However, we need to be able to find an arbitrary number of clusters in the document.

To solve the task of clusters of codings we use the one-dimensional clustering algorithm as shown in Algorithm 1. The algorithm is based on hierarchical descend with a local criterion as termination condition.

It expects a sorted array of integers as input. The array is then divided into two clusters. To do so, two pointers advance from the left and right border of the array until they meet. The distance between an element and its successor is used

Algorithm 1 Finding an arbitrary number of clusters in one dimension.

Require: x is a list of positive integers, sorted in ascending order; left and right are borders of the current array slice.

```
function Cluster($x$, left, right)
    $n \leftarrow \text{size}(x)$
    clusters $\leftarrow \emptyset$
    while left < right do
        leftDist $\leftarrow x[\text{left}+1] - x[\text{left}]$
        rightDist $\leftarrow x[\text{right}] - x[\text{right}-1]$
        if leftDist < rightDist then
            left $\leftarrow$ left + 1
        else
            right $\leftarrow$ right − 1
    leftCluster $\leftarrow x[0..\text{left}]$
    rightCluster $\leftarrow x[\text{right}..n]$
    if ClusterDensity(leftCluster) $\leq 0.6$ then
        clusters $\leftarrow$ clusters $\cup$ Cluster(leftCluster)
    else
        clusters $\leftarrow$ clusters $\cup$ leftCluster
    if ClusterDensity(rightCluster) $\leq 0.6$ then
        clusters $\leftarrow$ clusters $\cup$ Cluster(rightCluster)
    else
        clusters $\leftarrow$ clusters $\cup$ rightCluster
    return clusters

function ClusterDensity($y$)
    return $\frac{\text{AvgDistBetween}(y)}{\text{MaxDistBetween}(y)}$

function AvgDistBetween($y$)
    sum $\leftarrow 0$
    for $i \leftarrow 1$ to size($y$) do
        sum $\leftarrow sum + y[i] - y[i-1]$
    return $\frac{\text{sum}}{\text{size}(y)}$

function MaxDistBetweeen($y$)
    $k \leftarrow 0$
    for $i \leftarrow 1$ to size($y$) do
        if $k < y[i] - y[i-1]$ then
            $k \leftarrow y[i] - y[i-1]$
    return $k$
```

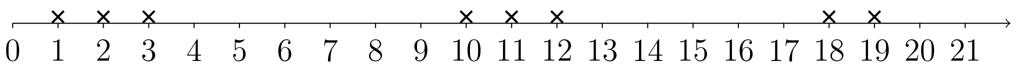
to decide whether the left or the right pointer should be advanced. The distance is given by the difference between the values of the array.

After two clusters have been found, a *density metric* of each cluster is used to decide whether the algorithm should be applied to the cluster. The density of a cluster is calculated as the ratio of the average distance and the maximum distance between values in the cluster. If the density is lower than a threshold, the cluster is divided again by recursively applying the algorithm again. Else, it is simply added to the set of clusters. For our work, a density threshold of 0.6 has yielded good results.

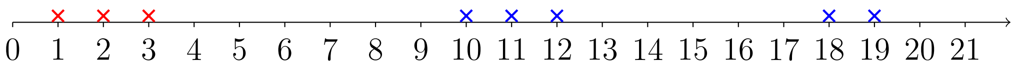
An example of clusters found by the algorithm is shown in Figure 3.1. Figure 3.1a shows the grouping of the numbers $1, 2, 4, 10, 11, 12, 18, 19$. For the human reader, it is readily apparent that a total of three clusters forms a pleasing subdivision of the set of numbers.

In Figure 3.1b, the first recursion step has been applied and the integers have been divided into two sets: $\{1, 2, 3\}$ form one cluster, whereas $\{10, 11, 12, 18, 19\}$ form the other. The latter cluster is too sparse according to the condition formulated above.

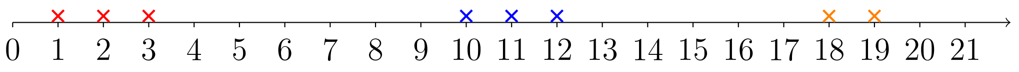
Accordingly, the algorithm recurses a second time on the set $\{10, 11, 12, 18, 19\}$. This time, the resulting sets are of unequal size: $\{10, 11, 12\}$ form the left cluster, $\{18, 19\}$ form the right cluster. Both clusters fulfill the density condition and no further recursion steps are necessary. Therefore, the algorithm returns the three clusters $\{\{1, 2, 3\}, \{10, 11, 12\}, \{18, 19\}\}$.



(a) Unclustered set of integers.



(b) First recursion: One appropriate cluster has been found (red), the other cluster must be divided again as it is not dense enough (blue).



(c) Second recursion: A total of three clusters (red, blue, orange) have been found.

Figure 3.1: Sequence of the clustering steps performed on a set of eight integers by recursively dividing.

With this algorithm, the local termination condition for the recursion leads to the clusters not to be of equal size. However, this is not required for our task.

3.2 Matching Codes to Linguistic Entities

3.2.1 Look-Up Table Approach

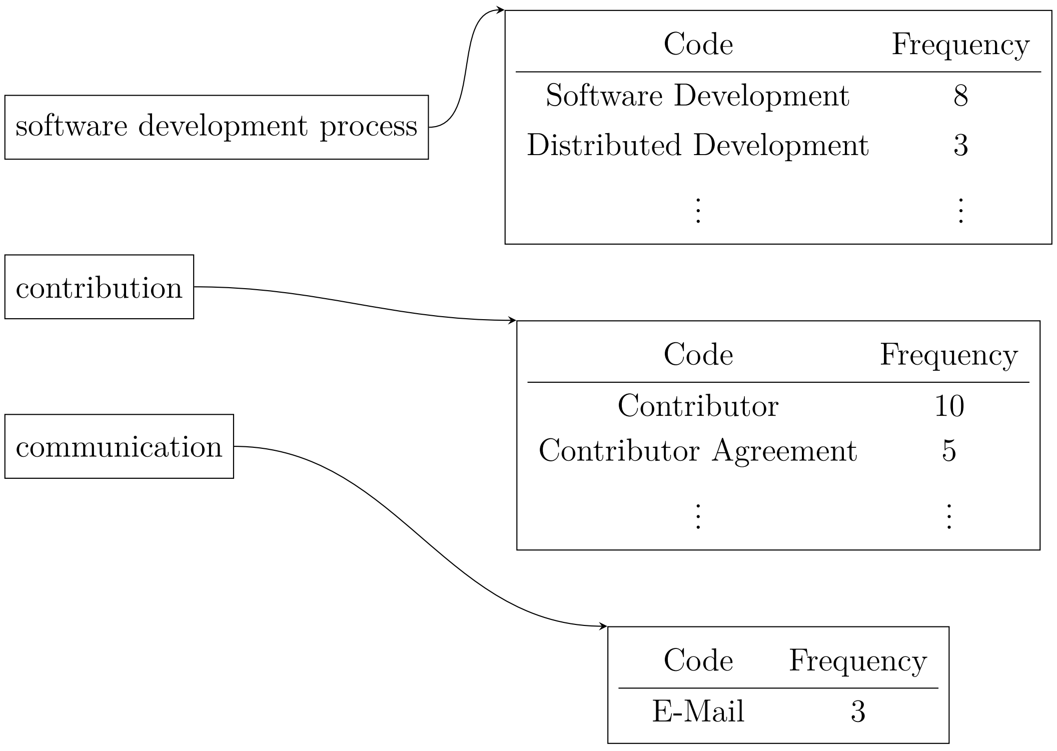
Dependency trees describe relationships between two entities found in a sentence and using the nouns from this sentence. To create a model containing only codes, these relationships must be transferred to the code system. In our software, this is done using a look-up table that contains multiple fitting codes for each linguistic entity.

The look-up table is populated before running the dependency tree algorithm. To do so, a scan over the text corpus is executed. For each sentence, noun phrases and their coding annotations are extracted. These pairings of noun phrases and codes are then added to the table. To store information about the likeliness that a pairing of a noun phrase to a code is correct, a frequency counter is incremented for each occurrence of the pairing in the texts. The structure of the look-up table can be seen in Figure 3.2.

In its simplest form, look-up is performed using the strings of linguistic entities as keys. To improve the extraction rate, lemmatization is performed on the linguistic entity. That means that instead of matching “software development processes”, the lemmatized form “software development process” is used as look-up term. Thereby, plural forms, possessive terms and other inflections are removed and do not hinder the look-up process.

Lemmatization of linguistic entities can be substituted by stemming. Stemming removes word endings to reduce similar words to a common stem. This word stem does not necessarily have to be an English word itself. Using this method, similar words can be traced back to the same code in the look-up table. Consider for example the words “developer”, “development”, and “developing”. Using stemming, all of these words can be transformed into “develop”.

A special case is the treatment of compounds. Compound nouns provide a more detailed description of a general concept. With the method described above, matching codings for compound nouns can only be found if exactly the same compound noun is already present as a key of the look-up table. Too fine-grained compound nouns such as “software development process” do however not result in a matching code if only a more general term such as “development process” is available.



Code	Frequency
Software Development	8
Distributed Development	3
⋮	⋮

Code	Frequency
Contributor	10
Contributor Agreement	5
⋮	⋮

Code	Frequency
E-Mail	3

Figure 3.2: The entity–code look-up table: Entities can be substituted by multiple codes. Each code has an attached weight.

For this purpose, an option to decompound nouns was implemented. Our approach follows the right-hand-head-rule as described by (Williams, 1981). The rule states that the rightmost word of a compound is also its head, meaning the word with the most general description. Additional specifying terms are added on the left side.

Therefore, if no match is found for a complex compound noun $w_0, w_1..w_n$, we remove the leftmost term and try again to find a match using the term $w_1..w_n$. This step is repeated until a match is found or only the head w_n remains and the result set is empty.

Application of the right-hand-head-rule, or lack thereof, can be combined with both lemmatization and stemming, giving a total combination of four different modes for matching entities to codes.

3.2.2 Weighting with WordNet

WordNet by (Fellbaum, 1998) is a database of English words. It contains information about meanings and relatedness of both nouns and verbs. WordNet contains a variety of information regarding a word. For our work, the concepts of synonyms and hypernyms are especially relevant.

A *synonym* of a word is another word with the same meaning. In WordNet, synonyms for one concept are grouped into so called synsets. One synset contains all words with the same meaning. As an example, the words {castle, palace, manor, stately home} form a synset.

A *hypernym* is a more general term for a word, i.e. its superordinate. The subordinate word (that has a more refined meaning) is called a hyponym. Therefore, the relationship between a hypernym and its hyponym forms a "is a" relationship. In WordNet, the hypernym/hyponym relationships form a tree. The root node of this tree is the concept of an *entity* as the most abstract concept.

The hypernym of a word depends on the synset that is chosen for a word. For example, the word *castle* has – besides others – the meaning of "a large and stately mansion" as well as "a large building formerly occupied by a ruler and fortified against attack". The first meaning has the direct hypernym *mansion*, further generalizing its meaning as a representative housing. The latter meaning has the direct hypernym *fortification*.

Using the example word *castle* also shows that not only abstract concepts enter into the dictionary of WordNet. The synset for *castle* in the meaning of "fortification" also contains the Maginot Line, a concrete instance of a fortification.

For our thesis, we are interested in determining the relatedness of two words. This information can then be used to adjust the matching of codes to linguistic entities. Thus, the methods presented in the following sections aim to help us in finding out whether two given nouns, originating from the linguistic entity and a code from code system, fit together.

Strict Hypernym/Hyponym Relationship

The strict hypernym/hyponym relationship is a strong indicator that the words used in the linguistic entity and the code that should be correlated are related. Because the WordNet tree for nouns is built from hypernym/hyponym relationships, we can use WordNet directly to identify hypernyms.

The hypernym/hyponym relationship does not have to be direct. Instead, we can also find grandparents or even more distant parents for a given noun. Within the WordNet tree, we find a path between the words in question. The length of this path is then used as the distance between the words. This distance can then be incorporated into the NLP pipeline to determine whether an entity and a code fit together.

There are two special cases to consider. First, if two words are equal, the distance between those two is zero. Second, if the two words are not in a hypernym/hyponym relationship, we consider the distance to be infinite.

The fact that one noun can be found in different synsets that do have different hyponyms makes identifying the correct hypernym of a noun difficult. We do not have semantic information that could help us identify the exact synset for a given noun. Therefore, we use a simple heuristic to mitigate this problem: Of all the synsets for a given word, we calculate all possible hypernym distances. We then select the smallest distance as result.

Shortest Ancestral Path

How can the relatedness of two terms be determined if they are not hyponym and hypernym? As WordNet forms a tree, two nodes in the tree can always be traced back to a common ancestor. In the worst case, this common ancestor is the root of the WordNet tree.

To measure the relatedness of two terms t_1, t_2, we use the path distance between t_1 and t_2 over a common ancestor a. An example for the application of the shortest ancestral path algorithm to the words *software* and *product* is shown in Figure 3.3.

Note that there are different synsets that can be used as a starting point of navigating the WordNet tree to a common ancestor. For the word *product*, the most suitable synsets appear to be “commodities offered for sale” or “an artifact that has been created by someone or some process”. Depending on the context, both these meanings can be applied to *software*. The definitions of *product* as a term in the domain of mathematics or chemistry does not fit the context of software development.

To deal with the various synsets, we take the minimal shortest ancestral path over all possible ancestral paths of all synsets. This means that we do not differentiate between different meanings of the synsets, but take an optimistic approach. In some cases, such as the example in Figure 3.3, this approach leads to the algorithm using the wrong synset. Here, the synset of *product* in the meaning of “result of a process” is used as its shortest ancestral path is smaller than the one that uses the meaning of *good, merchandise* that would be more fitting.

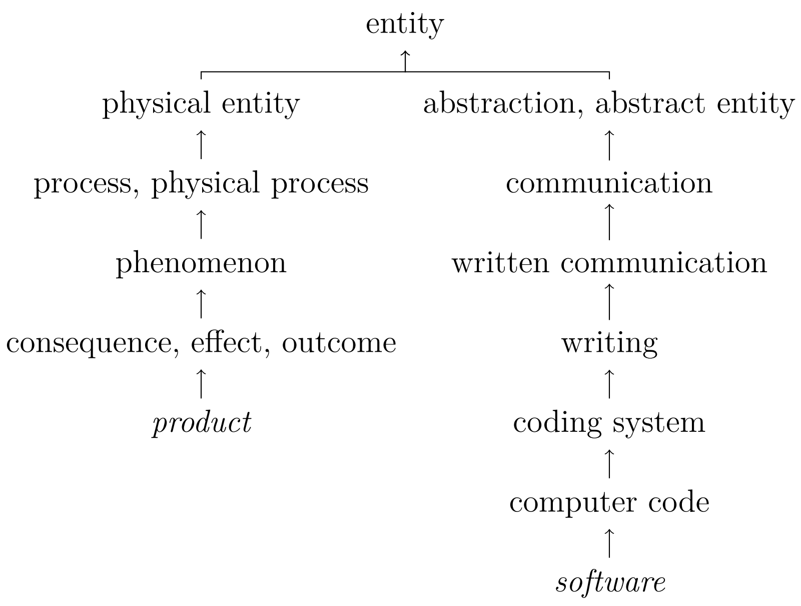


Figure 3.3: Shortest Ancestral Path algorithm in WordNet applied to the words *software* and *product*. Arrows show a hypernym-to-hyponym relationships. The total number of all edges (and therefore the shortest ancestral path) is 12.

3.2.3 Web Search Engines as Knowledge Bases

Introduction

As noted in the previous section, WordNet mixes concepts and instantiations. It also in some cases produces results that do not match the expectations of a domain expert. For example, the Shortest Ancestral Path between the words *software* and *product* is 10, although in reality they are closely related.

Additionally, the resolution of synsets is difficult. It is often unclear, which synset should be chosen for a specific word. Therefore, an additional knowledge base is needed.

Prior work has been done on the topic of relationship extraction from the Internet as information source. Kozareva et al. (2008) propose using a search engine as data source for learning instances of classes of things. In their work, they incrementally expand classes of concepts by using the search engine to find new instances of the class.

To retrieve instances for a certain class, they use the class name and an initial seed instance of the class. With these values, the so called doubly-anchored pattern "<class_name> such as <class_instance> and *" is used to find new instances.

The use of the doubly-anchored pattern allows to differentiate between homonyms that belong to different classes. (Kozareva et al., 2008) name the use of the word *Ford* as an example. It can both stand for a president of the United States of America and a car manufacturer. Using other methods to find related entities, such as WordNet, makes it difficult to decide which meaning is suitable. By using the class name in the query string, this problem is eliminated. For example, applying the DAP to the class "car" using the instance "Ford" results in a list of additional auto manufacturers.

Algorithm 2 *Reckless Bootstrapping* algorithm for the doubly-anchored pattern (Kozareva, Riloff, & Hovy, 2008).

```
 1: Members ← {Seed}
 2: $P_0$ ← "*Class* such as *Seed* and *"
 3: $P \leftarrow \{P_0\}$
 4: iter ← 0
 5: while iter < Max_Iters and $P \neq \{\}$ do
 6:     iter ← iter+1
 7:     for each $P_i \in P$ do
 8:         Snippets ← WEB_QUERY($P_i$)
 9:         Candidates ← EXTRACT_WORDS(Snippets, $P_i$)
10:         $P_{\text{new}} \leftarrow \{\}$
11:         for each Candidate$_k$ ∈ Candidates do
12:             if Candidate ∉ Members then
13:                 Members ← Members ∪ {Candidate$_k$}
14:                 $P_k$ ← "*Class* such as *Candidate$_k$* and *"
15:                 $P_{\text{new}} \leftarrow P_{\text{new}} \cup \{P_k\}$
16:     $P \leftarrow P_{\text{new}}$
```

The algorithm used to find new instances is shown in Algorithm 2. It iteratively builds a set of members starting from the initial seed term. Two methods are central to the algorithm. WEB_QUERY() retrieves preview snippets from the search engine. EXTRACT_WORDS() then uses these preview snippets to find new instances by matching the search pattern to the text of each snippet. Words that are found in the place of the wild chard "*" are added as new class members. To restrict the number of searches for each class, a maximum number of iterations can be configured. A higher number means that more instances can be found, however executing more searches also poses the risk of finding instances that do not actually belong to the class.

Adaption

Differing from the original goal of the doubly-anchored pattern, the goal of our application is not to build a catalog of semantic classes. However, one part of the NLP pipeline is similar to such a catalog: The entity-code look-up table. Therefore, we decided to use the doubly-anchored pattern during the initial population phase of the table.

During this phase, noun entities that are annotated with a code are linked to this code in the table. Multiple entities can link to the same code. Therefore, we use the doubly-anchored pattern to find entities similar to those found in the text corpus on the web. These entities are then added to the table as an instance for the code they were found with.

The pattern need two parameters: A class name and an instance. In our case, the code found by the coding annotation was used as class name. The entity that was annotated with this code in the text corpus is used as seed instance.

Finding additional instances of the code was achieved by following the algorithm named *Reckless Bootstrapping* by (Kozareva et al., 2008).

Due to the sheer amount of search request we would use to populate the entity look-up table with the pattern, we decided to reduce the maximum iteration number for the *Reckless Bootstrapping* algorithm to two. Nevertheless, roughly 7200 search requests were needed when analyzing the "Inner Source" data set.

Search Engine Access

The authors of the *doubly-anchored pattern* used the Google search engine as data source. However, the time of this writing, the Google search API has been discontinued and its successor, the Google Custom Search API is rate-limited in a way that makes it unusable for our purpose.

We therefore resorted to using the Microsoft Cognitive Services Bing Search API preview as data source. It can be queried via a simple REST interface. Default values for the query have been chosen as visible in Table 3.1.

Parameter	Value	Description
mkt	en-US	Country settings
safesearch	Off	Restrict results by family-friendliness
q	`<query>`	The query string

Table 3.1: Query options used for the Bing API.

Search results could not be easily replicated by the search engine. This means, that the number of results would change when the same query was executed several times. This was the case even for queries that were handled in the short timespan of five minutes.

3.3 The Code System

Some properties of the code system, such as using a code system meta model, provide further indications for relationship extraction.

3.3.1 A Code System Meta Model

Information that can be gained from QDA projects relies on the way the coding is performed and the code system is created. For example, generating hierarchy information from the code system tree is only possible if the tree is well-formed and the information represented in the tree structure is correct.

The “Inner Source” project that we used as data source for evaluating the methods investigated in this thesis is created using a meta model. This model was developed by Salow (2016). This meta model aims at deriving both structural as well as behavioral models from the same coding. To do so, every code is assigned meta data in the form of a memo that is attached to it within the QDA software.

The memo is a text document that contains well-structured information. While the memo also contains relationship information that was gained during the coding process by the human coder, we are interested in information about the entity described by the code.

This information is encoded into two different axes. The *Label* describes the structure of a code. The *Aspect* of a code encodes semantic information about the role of the code. It may be either one of the following values: *Object*, *Place*, *Actor*, *Activity*, or *Process*.

Relationships are then created according to a set of rules. However, not all rules are strict. Instead, some optional rules are applied by the coder according to their judgment. Therefore, automatic application of the rules to generate all relationships is implausible. We can nevertheless use the information to enrich our own data with relationship candidates.

We were especially interested in the semantic information encoded in the *Label* property of the memos. For this purpose, we investigated the correlation between the label type and the number of relationships in the “Inner Source” QDA project.

Aspects	Relationship Count	Cumulated
Aspect/Aspect	10	
Process/Process	8	homogeneous: 24
Object/Object	8	
Object/Process	8	
Process/Object	4	
Actor/Process	4	heterogeneous: 20
Actor/Object	3	
Object/Actor	1	

Table 3.2: Undirected relationships and their correlation with the *Label* meta data annotation of the used codes.

The domain model used for the analysis was not the model generated by our algorithm, but the "gold standard" model provided by Salow (2016). The results can be seen in Table 3.2.

Apparently, there exists a slight bias towards relationships between two codes of the same *Label* type. However, a ratio of 24 homogeneous relationships to 20 heterogeneous relationships is not strong enough for us to deduce a rule that would increase the weight of homogeneous relationship candidates in our algorithm. We therefore decided not to follow up on this idea.

3.3.2 Code Naming Conventions

The codes in the code system encode concepts, entities and in some cases actions. To enable automatic analysis of codes, they should be well-formed.

In the "Inner Source" project by (Salow, 2016), codes that should be used in the domain model are enumerated. Thus, they can be identified by matching code names to the pattern `#<number> <actual code name>` where `<number>` stands for the number of the code, followed by the actual code word.

Code Enumeration

Prefixing codes with numbers or other meta-information poses a challenge for automated analysis using NLP techniques. Arbitrary tokens in the code name interfere with attempts to extract PoS information. Therefore, code names have to be normalized by removing numeral prefixes and special characters used in such phrases.

However, code numbers are not only a nuisance. On their positive side, they allow identification of codes that should be used for the domain model. In the case of the "Inner Source" project, codes that should be used for the domain system are identifiable by their numeric prefix.

Codes encoding Actions

Instead of encoding concepts, some codes encode actions instead. This method allows the coder to find evidence for behavior in the source material they are using. In our research it became evident that such codes must not only be removed, but can also used as a source of information themselves.

For the purpose of extracting a domain model automatically, these codes can not be used. In the "Inner Source" project, one such code is `find Stakeholders`. It describes the act of identifying all parties with claims in a software project. However, being an action, it cannot be used as an entity in the domain model. Instead, it can be used as a directed relationship between the *actor* that performs the search and the code *stakeholder*.

To implement identification and removal of such codes, we implemented a simple mechanism based on and underlying pattern in the code name. We extract PoS annotations of the code name (that has been cleaned as described in subsection 3.3.2). Codes which have a first PoS tag that identifies a verb are accepted as code describing an action.

A drawback of this method is the difficulty of correctly assigning PoS tags. This process is error-prone for codes consisting of compound nouns. As the code names are very short fragments of text and not complete sentences, the PoS tagger stumbles in some cases and assigns wrong tags to words. One such example is the compound noun "Open Source Software". As can be seen in Figure 3.4, "Open" is interpreted as the verb "to open".

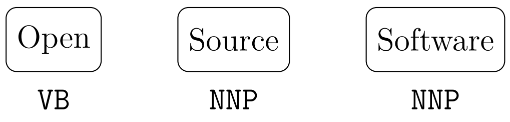


Figure 3.4: The PoS tag of "Open" is erroneously identified as verb instead as a part of the compound noun.

To deal with this problem, a blacklist of words that mislead the PoS tagger has been included in our software. These words, such as "Open Source Software" are accepted as valid noun phrases and action identification is suppressed.

Of course, this approach is highly domain specific and requires user intervention. Further research in the area of code system information extraction should probably resort to more robust methods such as using a dictionary to identify valid nouns.

3.3.3 Impulses for Future Research Work

Our work was highly experimental, as we undertook to our knowledge the first attempt to create relationship models from QDA data. While we evaluated basic approaches to the problem, we identified two main areas of work that would benefit from further inquiry.

Adaption of Machine Learning

In accordance with the recent trend towards machine learning, learning approaches are brought into relationship extraction. Neural networks have been applied to relationship extraction (Zheng et al., 2016), yielding good results. Similarly, Nguyen, Matsuo, and Ishizuka (2007) use a Support Vector Machine (SVM) to extract relationships from the Wikipedia corpus. While these papers are – as with our approach – based on Dependency trees, they do not use rules to extract the relationships. Instead, the dependency trees are, in reduced form, fed into the learning algorithm. Airola et al. (2008) propose a kernel-based approach for this task.

Unfortunately, due to the restricted data set available for our work, we could not experiment with adapting machine learning-based relationships extraction to QDA. For this to work, a larger data set that allows the separation in training and evaluation data would have been necessary.

We hope that with the ongoing work on QDA within the QDAcity project[2], such a data set will become available.

To facilitate the development of machine learning-aided approaches to relationship extraction, we propose a *strict coding guideline*. It should adhere to the following principles:

1. Codings should be applied to small text segments. This allows automated approaches to identify the exact statement that is referred to by the coding. Unfortunately, this was not always possible in our data set, where some codings covered a span of half a page of text.
2. The code system should ideally contain only codes relevant for the domain model. In our prototype, we had to include a function to remove unwanted codes.

[2]See qdacity.com.

Code Matching Problem

One major problem we encountered in our work was the matching of entities in the texts, that were referred to by the dependency tree rules, to codes in the code system. We tried to mitigate the problem by relying on co-occurence of nouns and codes in our corpus and WordNet as additional knowledge base.

For further research in this are, we propose adding additional knowledge bases. Such knowledge bases may include the Wikipedia or ready-made ontologies. This additional information can then be used to find more matching codes and nouns accurately.

This suggestion may be criticized for not relying only on the QDA project data for finding relations. Ideally, one might argue, a closed-world assumption should be valid for the task of relationship extraction. However, as we have found high accuracy of relationships generated based on the coding system. The coding system itself is created by the coder. As the coder brings a-priori knowledge into the creation of the coding system as well as in the final creation step of the domain model, we believe that this closed-world assumption is not valid for traditional, human-based approaches.

Appendix A Overview over the Software Artifact

A.1 General Information

The prototypical implementation of the relationship extraction system we developed in the course of this thesis is written in Java.

The program expects a single command line parameter (`-config`), which points to a configuration file. To facilitate experimentation with parameter values, a helper program was created that creates permutations of `.properties` files based on a job description. It is provided on GitHub[3].

A.2 Architecture

The program is structured around a central `workflow` that determines the sequence of the program starting at input file reading over various NLP methods up to the output file generation. These methods have all been moved to their own sub-packages within the `algorithm` package: These are packages `codesystem`, `codingcluster`, `coreferences`, `dependencytrees`, `entitycodematching`, `model`, `webquery`, `wordnet`, and `workflow`.

The system employs dependency injection using Google Guice as supporting library. This pattern helps to to bootstrap the application from a single entry point and to make implementations interchangeable. The latter aspect proved to be especially useful, as it allowed us to exchange modules by adjusting the central Guice binding definition. Runtime parameters are also injected via dependency injection by parsing the configuration file in the Guice module definition and injecting the parameters as `@Named()` constructor parameters into the objects where they are needed.

A.3 Scripts

A set of scripts is provided in the directory `scripts/`. Four bash shell scripts provide easy deployment to a remote server and enable remote debugging using a SSH tunnel.

Two Python scripts aid in the evaluation of the software. `run_batch.py` expects a directory as parameter and starts the program for each configuration file found.

[3]https://github.com/m-hofmann/properties-permutator

`read_resultfile.py` parses and sorts result files according to their accuracy.

A.4 Parameters

In the course of this thesis, we evaluated many approaches to relationship extraction from QDA data. All these approaches have been built into the software prototype, and have been made configurable to test them against each other. The list of configuration parameters with their data type and explanation is shown in Table 3.3.

Table 3.3: Parameters in the configuration file of the prototypical relationship extraction system.

Parameter	Datatype	Comment
cacheDirectory	String	Path to cache directory.
buildDocumentCache	Boolean	Specify whether documents should be parsed by CoreNLP or the if the cache should be used.
buildEntityCodeTableCache	Boolean	Specify whether the entity-code table should be rebuilt or loaded from cache.
idealGraphFile	String	Path to file containing gold-standard graph representation for evaluation.
stemLinguisticEntites	Boolean	Whether stemming should be used for entity-code table lookup.
rhhrSplitting	Boolean	Whether noun decompounding should be used for entity-code table lookup.
useDAPEntityFinder	Boolean	Whether the doubly-anchored pattern should be used to build the entity-code table.
strictNounOnlyRelationship	Boolean	Whether only dependencies where subject and object are strict noun phrases should be accepted.
clusterRelationshipType	Enum	One of: DIRECTED, UNDIRECTED
clusterRelationshipMode	Enum	One of: NONE, FIRST_TO_CHILDREN, CHILDREN_TO_FIRST, CARTESIAN
clusterRelationshipWeight	Integer	Weight of relationships created from clusters.

Continued on next page

Table 3.3 – continued from previous page

Parameter	Datatype	Comment
clusterOnlySlides	Boolean	Whether the clustering algorithm should only be applied to slides.
undirectedSelfLoops	Boolean	Whether self loops should be allowed in the undirected part of the graph.
dependencyAlgorithm	Enum	One of: nop, basic, slidingWindow
dependencyRules	Enum List	One or more of: NSUBJ_DOBJ, NSUBJ_NOBJ, NSUBJ_DOBJ_LINKED_VERBS, CCOMP, ADVERBIAL_ROOT, NMOD_NSUBJPASS, NSUBJ_NMOD
slideWindowSentencesBefore	Integer	Look-behind of the sliding window algorithm.
slideWindowSentencesAfter	Integer	Look-ahead of the sliding window algorithm.
slideWindowBoostCurrentCoding	Boolean	Whether codes within the current window should be weighted higher.
slideWindowBoostValue	Integer	Additional weight for codes in the sliding window.
slideWindowSubstituteCode	Boolean	Whether the slide window code should be used for entity-code matching.
useWordnetRelatednessForDirected	Boolean	Whether WordNet should be used in entity-code matching of directed relationships.

Continued on next page

Table 3.3 – continued from previous page

Parameter	Datatype	Comment
useWordnetRelatednessForUndirected	Boolean	Whether WordNet should be used in entity-code matching for undirected relationships.
wordNetRelatednessPower	Integer	Power-of-n boost for entity-code matchings found in WordNet.
useWebSearchRelatednessForDirected	Boolean	Whether web search should be used in entity-code matching of directed relationships.
useWebSearchRelatednessForUndirected	Boolean	Whether web search should be used in entity-code matching of undirected relationships.
webSearchRelatednessPower	Integer	Power-of-n boost for entity-code matchings found with web search.
useEntityCodeTableQuantiles	Boolean	Whether quantile-based filtering should be used for entity-code table results.
entityCodeTableQuantileThreshold	Integer	Threshold for entity-code table results.
codeSystemAncestorsEnabled	Boolean	Whether the code system hierarchy should be used to generate relationships.
codeSystemAncestorsWeight	Integer	Weight used for relationships generated from code system hierarchy.
codeSystemAncestorsUseWordNet	Boolean	Whether WordNet should be used to influence the weight of code system ancestor relationships.

Continued on next page

Table 3.3 – continued from previous page

Parameter	Datatype	Comment
codeSystemAncestorsWordNetMode	Enum	One of: SHORTEST_ANCESTRAL_PATH, HYPERNYM_HYPONYM, HYPERNYM_OR_SAP
codewordFilterRegex	String	Regular expression for code name-based filtering.
ignoreSelfLoops	Boolean	Whether self loops should be ignored when creating the graph result from the model.
plotEdgeSumDistribution	Boolean	Option to output edge weights distribution for development.
avoidOppositeEdges	Boolean	Whether edges opposite to each other should be merged.
ratioCodeSystemDirectedEdges	Float	Ratio of edges from the code system (assumed high validity) vs other relationship sources.
codeSystemEdgeWeighting	Enum	One of: SINGULAR, TOTAL_SUM

Appendix B Bill of Materials

Table 3.4: Dependencies of the prototype implemented in the course of this thesis.

GroupId	ArtifactId	Version	License
com.beust	jcommander	1.69	Apache 2.0
com.google.code.gson	gson	2.8.0	Apache 2.0
com.google.guave	guava	21.0	Apache 2.0
com.google.inject	guice	4.1.0	Apache 2.0
com.google.inject.extensions	guice-assistedinject	4.1.0	Apache 2.0
com.google.inject.extensions	guice-multibindings	4.1.0	Apache 2.0
commons-io	commons-io	2.5	Apache 2.0
de.ruedigermoeller	fst	2.48	Apache 2.0
edu.stanford.nlp	stanford-corenlp	3.7.0	GNU GPL v3
io.dropwizard.metrics	metrics-core	3.2.2	Apache 2.0
junit	junit	4.12	Eclipse Public License 1.0
net.sf.extjwnl	extjwnl	1.9.2	BSD
net.sf.extjwnl	extjwnl-data-wn31	1.2	BSD
org.apache.logging.log4j	log4j-api	2.7	Apache 2.0
org.apache.logging.log4j	log4j-core	2.7	Apache 2.0
org.apache.tika	tika-parsers	1.7	Apache 2.0
org.freemarker	freemarker	2.3.25-incubating	Apache 2.0
org.glassfish.jersey.core	jersey-client	2.25.1	CDDL+GPL
org.glassfish.jersey.media	jersey-media-jaxb	2.25.1	CDDL+GPL
org.mockito	mockito-core	2.6.3	MIT
org.xerial	sqlite-jdbc	3.14.2.1	Apache 2.0

Continued on next page

Table 3.4 – continued from previous page

GroupId	ArtifactId	Version	License
org.zalando.phrs	jersey-media-json-gson	0.1	Apache 2.0

Glossary

coreference resolution The process of identifying all mentions of the same entity in a text, regardless whether they are referred to explicitly or by pronouns. 28

GraphVIZ Software package for graph layout and drawing. 18

MaxQDA Software to support Qualitative Data Analysis. 2, 4, 7

Stanford CoreNLP Java framework that implements various NLP algorithms. 4, 7, 10, 28, 29

Acronyms

ER entity-relationship. 3

NLP Natural Language Processing. 2, 3, 4, 5, 6, 7, 15, 23, 25, 26, 28, 34, 37, 40, 53

PoS part of speech. 7, 11, 40, 41

QDA Qualitative Data Analysis. vii, 2, 4, 5, 6, 18, 23, 24, 22, 24, 25, 26, 27, 39, 42, 43, 46

SVM Support Vector Machine. 42

UML Unified Modeling Language. 24

References

Airola, A., Pyysalo, S., Björne, J., Pahikkala, T., Ginter, F., & Salakoski, T. (2008). A graph kernel for protein-protein interaction extraction. In *Proceedings of the workshop on current trends in biomedical natural language processing* (pp. 1–9). Association for Computational Linguistics.

Ambriola, V. & Gervasi, V. (1997). Processing natural language requirements. In *Automated software engineering, 1997. Proceedings., 12th IEEE International Conference* (pp. 36–45). IEEE.

Black, W. J. (1987). Acquisition of conceptual data models from natural language descriptions. In *Proceedings of the third conference on european chapter of the Association for Computational Linguistics* (pp. 241–248). Association for Computational Linguistics.

Clark, K. & Manning, C. D. [Christopher D.]. (2015). Entity-centric coreference resolution with model stacking. In *Association for Computational Linguistics (ACL).*

Dobrovoljc, K. & Nivre, J. (2016). The universal dependencies treebank of spoken slovenian. In *Proceedings of the ninth international conference on language resources and evaluation (LREC'16)* (pp. 1566–73).

Du, S. & Metzler, D. P. (2006). An automated multi-component approach to extracting entity relationships from database requirement specification documents. In *International conference on application of natural language to information systems* (pp. 1–11). Springer.

Fellbaum, C. (1998). *WordNet.* Wiley Online Library.

Gansner, E. R. & North, S. C. (2000). An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, *30*(11), 1203–1233.

Garten, Y. & Altman, R. B. (2009). Pharmspresso: a text mining tool for extraction of pharmacogenomic concepts and relationships from full text. *BMC bioinformatics*, *10*(2), S6.

Kozareva, Z., Riloff, E., & Hovy, E. H. (2008). Semantic class learning from the web with hyponym pattern linkage graphs. In *ACL* (Vol. 8, pp. 1048–1056).

Kunz, K. (2015). *Developing a domain analysis procedure based on grounded theory method* (Master's thesis, FAU Erlangen-Nuernberg).

Marcus, M. P., Marcinkiewicz, M. A., & Santorini, B. (1993). Building a large annotated corpus of English: the Penn Treebank. *Computational linguistics*, *19*(2), 313–330.

Nguyen, D. P., Matsuo, Y., & Ishizuka, M. (2007). Relation extraction from Wikipedia using subtree mining. In *Proceedings of the national conference on artificial intelligence* (Vol. 22, *2*, p. 1414). Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Nivre, J., de Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajic, J., Manning, C. D. [Christopher D], ..., Silveira, N., et al. (2016). Universal dependencies v1: a multilingual treebank collection. In *Proceedings of the 10th international conference on language resources and evaluation (LREC 2016)* (pp. 1659–1666).

Omar, N., Hanna, J., & McKevitt, P. (2004). Heuristic-based entity-relationship modelling through natural language processing. In *Artificial intelligence and cognitive science conference (aics)* (pp. 302–313). Artificial Intelligence Association of Ireland (AIAI).

Microsoft Cognitive Services – Bing Web Search API. (n.d.). Retrieved April 10, 2017, from https://www.microsoft.com/cognitive-services/en-us/bing-web-search-api

Quan, C., Wang, M., & Ren, F. (2014). An unsupervised text mining method for relation extraction from biomedical literature. *PloS one*, *9*(7), e102039.

Robeer, M., Lucassen, G., van der Werf, J. M. E., Dalpiaz, F., & Brinkkemper, S. (2016). Automated extraction of conceptual models from user stories via nlp. In *Requirements engineering conference (RE), 2016 IEEE 24th international* (pp. 196–205). IEEE.

Roberts, A., Gaizauskas, R., & Hepple, M. (2008). Extracting clinical relationships from patient narratives. In *Proceedings of the workshop on current trends in biomedical natural language processing* (pp. 10–18). Association for Computational Linguistics.

Salow, S. (2016). *A metamodel for code systems* (Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg).

Schuster, S. & Manning, C. D. [Christopher D]. (2016). Enhanced English universal dependencies: An improved representation for natural language understanding tasks. In *Proceedings of the tenth international conference on language resources and evaluation (lrec 2016).*

Silveira, N., Dozat, T., de Marneffe, M.-C., Bowman, S., Connor, M., Bauer, J., & Manning, C. D. (2014). A gold standard dependency corpus for English. In *Proceedings of the ninth international conference on language resources and evaluation (LREC 2014).*

UML, O. (2015). Unified modeling languagetm (uml®) version 2.5.

Vulic, I. & Korhonen, A. (2016). Is "universal syntax" universally useful for learning distributed word representations? In *The 54th annual meeting of the association for computational linguistics* (p. 518).

Williams, E. (1981). On the notions “Lexically Related” and “Head of a Word”. *Linguistic Inquiry*, *12*(2), 245–274.

Withington, J. (2011). *PDF Explained.* O’Reilly Media, Inc.

Zheng, S., Xu, J., Bao, H., Qi, Z., Zhang, J., Hao, H., & Xu, B. (2016). Joint learning of entity semantics and relation pattern for relation extraction. In *Joint european conference on machine learning and knowledge discovery in databases* (pp. 443–458). Springer.