Optimizing a Natural Language Processing pipeline for the automatic creation of RDF data

Dambowy, Nils

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Bachelor Thesis

Institute for Computer Science
Goethe University Frankfurt

supervised by:
Dr. Karsten Tolle

Contents

Erklärung zur Abschlussarbeit

gemäß § 25, Abs. 11 der Ordnung für den Bachelorstudiengang Informatik vom 06. Dezember 2010:

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe.

Frankfurt am Main, den 04. Mai 2023 Dambowy, Nils

Abbreviations

CN	Corpus Nummorum
RDF	Resource Description Framework
NLP	Natural Language Processing
NER	Named Entity Recognition
RE	Relationship Extraction

1 Introduction

Numismatics is the academic discipline focussing on the study of different forms of currency, including e.g. coins, paper money or medals. Numismatists made it their task to collect, preserve and categorize these archaeological findings and thereby play an important role in the enlightenment of our history. This is because the coins can give us an idea of the time period they were used in. Additionally, if we know where the coin was made and where it was found, we can also learn about the mobility of people during that time. Today, most numismatic institutions have to go through the challenge of finding an adequate way of managing their data and mostly[1] rely on databases. In this thesis, I am focussing on the **Corpus Nummorum** database, which will be further explained in a later chapter.

Currently working with this database is the **D4N[4]** (Data quality for Numismatics based on Natural language processing and Neural Networks) project, which goal it is to improve the assignment of existing data (descriptions or images) to the CN portal and nomisma.org[2]. The assignment is improved through two different ways. First, Natural Language Processing is used to assign the coins based on their description, and then image recognition is employed. Furthermore, the project aims to continue the development and implementation of tools useful for numismatic research portals. One of these tools is the subject of this thesis.

1.1 Problem description

In this thesis, I want to optimize a currently existing **Natural Language Processing** pipeline, which will be described in more detail in **chapter 3.1**. The pipeline consists out of two parts. First **Natural Language Processing** is used to extract named entities (see **Named Entity Relationship**) and to create word relationships (see **Relationship Extraction**) out of the **CN** database. Afterward, the results are written back again. Lastly, with the new data and the program **D2RQ Resource Description Format** data is being created with the aim to be published later. Currently, the whole process, from the execution of the notebooks to the setup of the D2RQ program, has to be done manually. To improve this, the goal of this thesis is to refine the state of current pipeline by automating the process of the NLP and creation of RDF data without having to on extra tools like D2RQ. At the end, the whole execution of the pipeline should require as little human input as possible. When given a coin description as an input it should return the results in the RDF format.

[1] http://nomisma.org/datasets
[2] http://nomisma.org/

1.2 Thesis structure

In the following chapter I am going to discuss the CN project and its successor, the D4N4 project. The MySQL database the pipeline is working with originates with coin data from this project. Furthermore, in **chapter 2** technical background information about the different technologies used is given to understand the functionality of the new and old pipeline. Afterwards, in **chapter 3**, I will explain in more detail the old version and the revised version of the pipeline. Furthermore, the implementation of the revised one is discussed. In **chapter 4** both pipelines will also be compared. **Chapter 5** leaves room for the conclusion and the outlook.

1.3 Related work

In the bachelor thesis ”Natural Language Processing to enable semantic search on numismatic descriptions”(Klinger, 2018) the first version of notebooks used in the pipeline was developed. These notebooks first allowed the processing of the data using NLP. Below you can see the version of the natural language processing pipeline, which was developed in the bachelor thesis. The pipeline will also be explained in more detail in a later chapter.

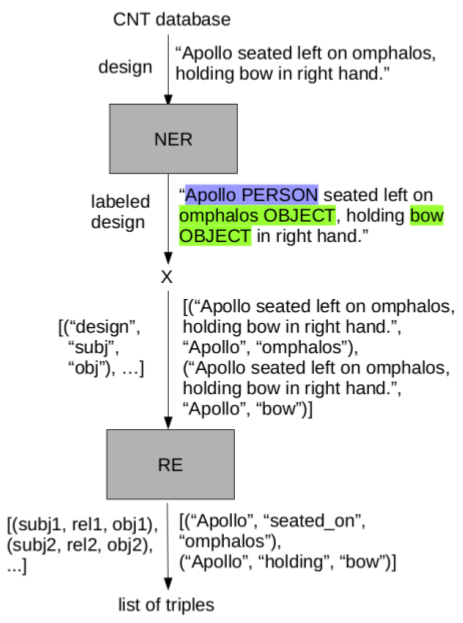

CNT database
design
“Apollo seated left on omphalos, holding bow in right hand.”
NER
labeled design
“Apollo PERSON seated left on omphalos OBJECT, holding bow OBJECT in right hand.”
X
[(“design”, “subj”, “obj”), ...]
[(“Apollo seated left on omphalos, holding bow in right hand.”, “Apollo”, “omphalos”), (“Apollo seated left on omphalos, holding bow in right hand.”, “Apollo”, “bow”)]
RE
[(subj1, rel1, obj1), (subj2, rel2, obj2), ...]
[(“Apollo”, “seated_on”, “omphalos”), (“Apollo”, “holding”, “bow”)]
list of triples

Figure 1: Scheme of the current pipeline version

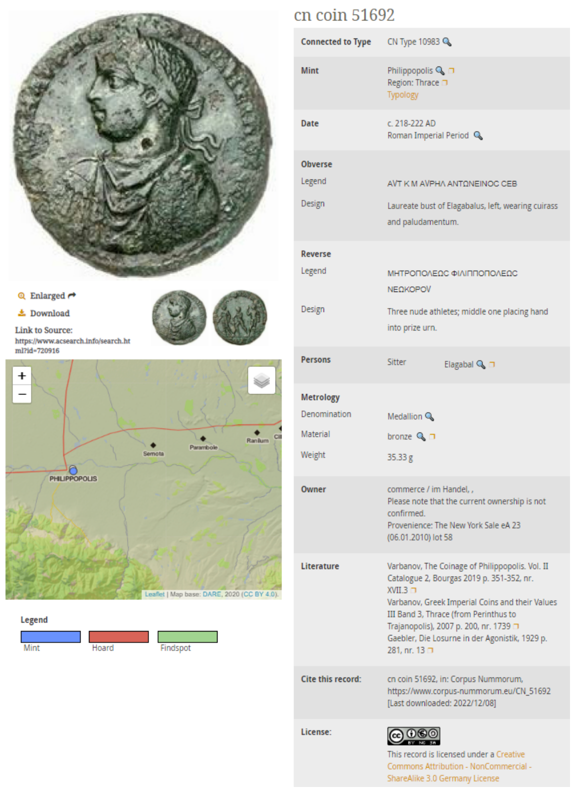
This version was later modified by (Deligio and Gencer, 2021) in their master thesis”Natural Language Processing auf mehrsprachigen Münzdatensätzen - Untersuchung der Qualität, Datenqualität und Übertragbarkeit auf andere Datensätze”. In their thesis they aimed to improve the current state of the notebooks by adding new entities and their relations to other entities to the NLP. Furthermore they added German translations to the previously existing entities and relations(Deligio and Gencer, 2021,p.3 Aufgabenstellung I.). To complete the overall pipeline and achieve the current state, Sebastian Gampe and Dr. Karsten Tolle from the Big Data Lab at Goethe University implemented the last step of the pipeline, which makes use of the D2RQ program in order to create RDF triples. Additionally, they also made adjustments and continued the development of the notebooks.

2 Background

2.1 Corpus Nummorum and the D4N4

Corpus Nummorum:

The Corpus Nummorum **(CN)** is a research database, which is the result of the joint work of the Münzkabinett Berlin, Berlin-Brandenburg Academy of Sciences and Humanities (BBAW) and the Big Data Lab of Goethe University, with the motivation to offer ancient Greek coinage for research purposes(*Corpus Nummorum site* n.d.). It contains information about coins with origin in the regions of Lower Moesia, Thrace, Mysia, and the Troad. Added together, the database contains information about approx. 28,000 publicly accessible coins, 17,000 coming from the area of Thrace and 11,000 from the remaining regions. It is also possible to contribute coins to the database yourself.



cn coin 51692

Connected to Type	CN Type 10983	
Mint	Philippopolis Region: Thrace Typology	
Date	c. 218-222 AD Roman Imperial Period	
Obverse		
Legend	ΑΥΤ Κ Μ ΑΥΡΗΛ ΑΝΤΩΝΕΙΝΟϹ ϹΕΒ	
Design	Laureate bust of Elagabalus, left, wearing cuirass and paludamentum.	
Reverse		
Legend	ΜΗΤΡΟΠΟΛΕΩϹ ΦΙΛΙΠΠΟΠΟΛΕΩϹ ΝΕΩΚΟΡΟΥ	
Design	Three nude athletes; middle one placing hand into prize urn.	
Persons	Sitter	Elagabal
Metrology		
Denomination	Medallion	
Material	bronze	
Weight	35.33 g	
Owner	commerce / im Handel, , Please note that the current ownership is not confirmed. Provenience: The New York Sale eA 23 (06.01.2010) lot 58	
Literature	Varbanov, The Coinage of Philippopolis. Vol. II Catalogue 2, Bourgas 2019 p. 351-352, nr. XVII.3 Varbanov, Greek Imperial Coins and their Values III Band 3, Thrace (from Perinthus to Trajanopolis), 2007 p. 200, nr. 1739 Gaebler, Die Losurne in der Agonistik, 1929 p. 281, nr. 13	
Cite this record:	cn coin 51692, in: Corpus Nummorum, https://www.corpus-nummorum.eu/CN_51692 [Last downloaded: 2022/12/08]	
License:	This record is licensed under a Creative Commons Attribution - NonCommercial - ShareAlike 3.0 Germany License	

Figure 2: An example of a coin in the database

All of this information is accessible through the CN Online website by using the provided search tool, which lets you filter coins by e.g. epoch, tribe, weight or material. Besides the mentioned characteristics, the CN offers textual descriptions of the depicted images on the front- and back side of the coins. To allow a better comparison between different coins, all properties have to be entered in accord to a standardized scheme.[1] This not only guarantees the previously mentioned upsides but also makes entering coins more accessible since e.g. volunteers do not have to worry about having to come up with a scheme themselves. The scheme offers guidelines for describing the obverse and reverse of the coin, figures or architecture, portraits, scenes or some general information on how properly describe a coin. After a submitting a coin, it has to be reviewed before it is published and added to the database.

[1]https://www.corpus-nummorum.eu/pdf/ExternalCoinEntry.pdf

D4N4:

The **D4N^4** is a research project and successor of the **Corpus Nummorum** project. The Name of the project is abbreviated from ”Data quality for Numismatics based on Natural language processing and Neural Networks”. The project aims to improve the usage of the many available images and descriptions of coins by classifying and assigning them to the different databases (*D4N4 project site* n.d.). The project was officially launched in July of 2021[1] and is the result of a collaboration between the Münzkabinett Berlin, Berlin-Brandenburg Academy of Sciences and Humanities (BBAW), and the Big Data Lab of Goethe University. The main objective of the project is to advance the development of tools needed for numismatics. By improving the quality of data through classification and assignment of coins, the D4N^4 project will provide researchers with tools for conducting analyses of numismatic data. The D4N^4 project utilizes technologies such as Natural Language Processing and Neural Networks to analyse large amounts of data in a more efficient way. The project's focus on developing and improving tools for numismatics is a critical step towards advancing the field and providing researchers with the necessary tools to make new discoveries.

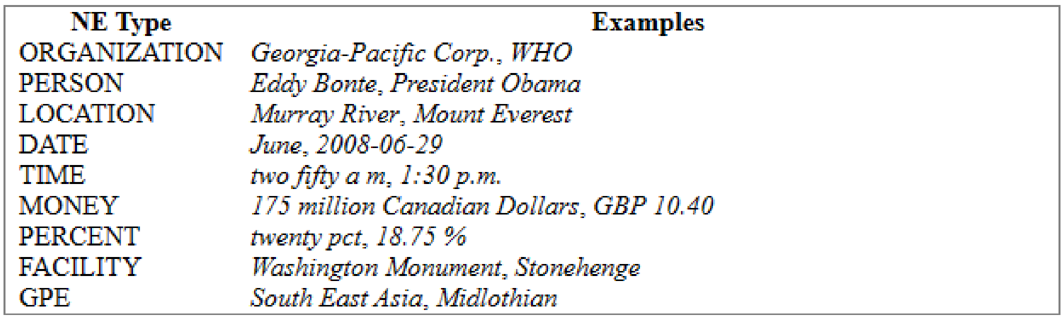
[1]https://www.corpus-nummorum.eu/news/1344?lg=en

2.2 Natural Language Processing

NLP deals with the processing of *natural* language. The term natural referring to the creation of the language which was natural e.g. through human conversation, unlike artificially created languages like programming languages. In Natural Language Processing, the language is given as input in text form for analyzation (Bird, Klein, Loper, 2009). To extract information such as entities, persons, places etc. out of natural text and process these, two different NLP-techniques are being used - Named Entity Recognition and afterwards Relationship Extraction.

2.2.1 Named Entity Recognition

In Named Entity Recognition(NER), natural text is being analysed for named entities. Named entities are words which give reference to a real world object using a proper noun, which identify only one object. Examples for that could be places, names, objects or organisations. (Bird, Klein, Loper, 2009) suggested the following types of named entities (Fig. 3):



NE Type	Examples
ORGANIZATION	*Georgia-Pacific Corp., WHO*
PERSON	*Eddy Bonte, President Obama*
LOCATION	*Murray River, Mount Everest*
DATE	*June, 2008-06-29*
TIME	*two fifty a m, 1:30 p.m.*
MONEY	*175 million Canadian Dollars, GBP 10.40*
PERCENT	*twenty pct, 18.75 %*
FACILITY	*Washington Monument, Stonehenge*
GPE	*South East Asia, Midlothian*

Figure 3: Types of named entities

Example - Recognition of named entities:

Yesterday in Frankfurt, John Smith met up with his friend Anna Mayer.

NER would then find the following entities in the sentence:

- *Frankfurt* as a *LOCATION*
- *John Smith* as a *PERSON*
- *Anna Mayer* as a *PERSON*

In the case of the NER pipeline used in the D4N4 project this could look like this:

Input:

Artemis standing right, wearing long garment, holding two arrows in right hand and bow in left hand.

Output:

Artemis PERSON standing right, wearing long garment OBJECT, holding two arrows OBJECT in right hand and bow OBJECT in left hand.

As shown in the example NER takes a description of a coin as input, finds the named entities and then labels them. Altogether there are four different labels, which are being assigned: *PERSON*, *OBJECT*, *ANIMAL* or *PLANT*.
It does that by making use of the spaCy package. spaCy is an open-source python libary [1] that offers different tools for processing natural language including the class *EntityRecognizer*[2] which annotates named entities in a given natural text (Klinger, 2018, chapter 4.1, p.14). The output then consists out of a list of triples. These triples represent the respective found named entity and contain the first position, *start*, and the last position of the NE, *end*. Lastly, the assigned *label* is saved in the triple. However, it was still necessary to train the named entity recognizer. For that P.Klinger implemented a four step NER workflow, which can be seen below:

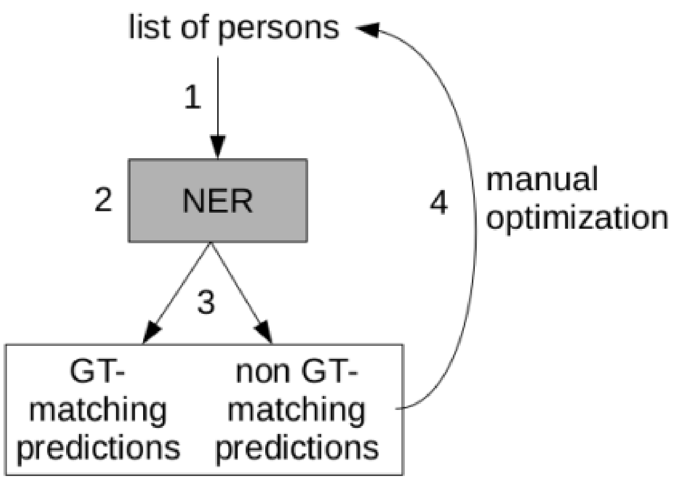


Figure 4: NER workflow

[1]https://spacy.io/
[2]https://spacy.io/api/entityrecognizer

After training (step 3 in the workflow), the NE recognizer is tested using the test data set. Afterwards the output is divided into two different groups - *ground truth-matching predictions* and *non ground truth-matching predictions*. Annotations that were predicted correctly were put in the first group and otherwise were put in the latter(Klinger, 2018, ch.4.2, p. 15). However, C. Deligio and K. Gencer noted the risk of *false positives* due to manual assignment of groups. ”This comes about because the annotation, i.e. the ground truth, is created through a manual process that only reflects the current state of knowledge about accepted entities.”(Deligio and Gencer, 2021, ch. 4.3.1, p.37, translated). As a solution, C. Deligio and K. Gencer suggested reviewing the latter list manually. During this process the false positives can be subsequently added as new entities. As indicated in figure 4, the workflow can be repeated until it produces satisfying results for all four entities.

2.2.2 Relationship Extraction

In text, words are connected over different relationships. With RE it is possible to identify and extract these relationships(Bird, Klein, Loper, 2009).

Example:

A car *is a* vehicle.

As seen above, the fragment '*is a*' is the relationship connecting the subject and the object and would be recognized by the Relationship Extraction.

In the current version of the pipeline, the RE takes as input a design plus a subject and an object. For one design the cross-product of the different found named-entities is created and passed as an input to the RE together with the design where the entities originated from. Afterwards it tries to establish a relationship between the subject and object using the input (Klinger, 2018, chapter 4). The output then consists out of the relations which are annotated in a triple form (NE1, α, NE2), where α is the sequence of words that connects the two entities NE1 and NE2 (Bird, Klein, Loper, 2009, ch7.6).

Input:

("Artemis standing right, wearing long garment,holding two arrows in right hand and bow in left hand.", "Artemis", "garment")

("Artemis standing right, wearing long garment,holding two arrows in right hand and bow in left hand.", "Artemis", "arrows")

("Artemis standing right, wearing long garment,holding two arrows in right hand and bow in left hand.", "Artemis", "bow")

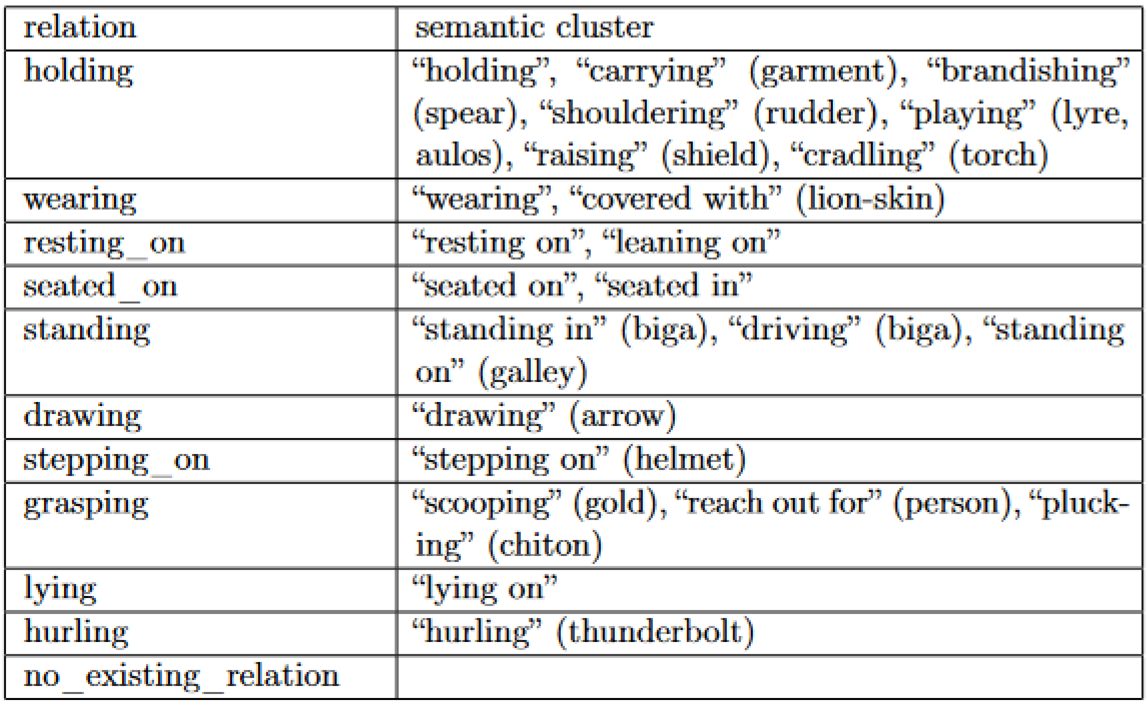
Output:

("Artemis", "wearing", "garment"")

("Artemis", "holding", "arrows")

("Artemis", "holding", "bow")

In the version, developed in her thesis, P. Klinger used an approach which can be understood as a multiclass problem. Due to the size of the design dataset being too small, different approaches did not seem viable(Klinger, 2018, chapter 4.2, p. 16). Since the types of relation were already known, P.Klinger decided to assign different classes for the respective relations.



relation	semantic cluster
holding	“holding”, “carrying” (garment), “brandishing” (spear), “shouldering” (rudder), “playing” (lyre, aulos), “raising” (shield), “cradling” (torch)
wearing	“wearing”, “covered with” (lion-skin)
resting_on	“resting on”, “leaning on”
seated_on	“seated on”, “seated in”
standing	“standing in” (biga), “driving” (biga), “standing on” (galley)
drawing	“drawing” (arrow)
stepping_on	“stepping on” (helmet)
grasping	“scooping” (gold), “reach out for” (person), “plucking” (chiton)
lying	“lying on”
hurling	“hurling” (thunderbolt)
no_existing_relation	

Figure 5: The proposed relations by P. Klinger

With the classes created, the task of extracting the relationship was reduced to asking if a given pair of named entities has a certain relation or not. Note that a named entity pair has the bottom class in figure 5 as the relation when there is no existing relation between the named entities. In the version developed by P. Klinger only named entity pairs, where one entity was labeled as a person (subject) and the other one was labeled as an object (object), were considered. As a result only those designs, that contained these named entities were annotated by the RE(Klinger, 2018, chapter 4.2, p.16). Another point P. Klinger had to take into consideration was that not all existing relations contained one of the created classes (see figure 5) which led to the implementation of semantic clusters, which are groups that contain relations that are semantically equivalent. With these groups created even edge cases like ”Nike in biga, right”, were correctly annotated as [”Nike”, ”standing”, ”biga”].

Continuing the work of P. Klinger, in their master thesis ”Natural Language Processing auf mehrsprachigen Münzdatensätzen - Untersuchung der Qualität, Datenqualität und Übertragbarkeit auf andere Datensätze” C. Deligio and K. Gencer introduced new labels (ANIMAL, PLANT) to the RE. This led to the creation of new named entity pairs, where the subject can exist as an *ANIMAL*, *PERSON* or an *OBJECT*. Since coin descriptions with the subject being labeled as *PLANT* were not common in the dataset, *PLANT* was not added to the possible subjects. Furthermore, they expanded the classes of relations to 18 different classes and implemented the RE for coin descriptions in the German language(Deligio and Gencer, 2021).

2.3 MySQL.connector

In order to retrieve the data that is stored in the MySQL database, the MySQL.connector library is used, which is offered[1] by MySQL themselves. The package allows the script to access the data by using SQL queries. In order to that, a connection to the database has to be established, and a cursor has to be initialized. Both can be seen in the example below.

Example:

```python
mydb = mysql.connector.connect(
    host = "localhost",
    user = "USERNAME",
    password = "PASSWORD",
    database="DB_NAME"
    )
cursor = mydb.cursor(buffered=True)
```

With a working database connection and cursor it is now possible to use the MySQL cursor to send SQL queries and retrieve data. Note that the cursor has the argument *buffered* set to True, in order to enable buffering[2] for this cursor. The cursor now automatically fetches the complete result from the server, which useful when iterating over the fetched results (*MySQL Connector/Python Developer Guide* n.d.). Implementing the same function with an unbuffered cursor would require fetching the result(using for example cursor.fetchall()) and then saving it in another variable. This means that by using a buffered cursor, those lines are saved when it comes to that special case. Below you can see an example for how data is retrieved from the MySQL database using the cursor:

Example:

```python
cursor.execute("Select id_design from d2r_coin_obv_design where id_coin =
  '3941';")
query_result = cursor.fetchall()
```

The result, with id = 3941, would be $[(10,)]$. In order to access the retrieved data the variable *query_result* has to be indexed accordingly. In this example, query_result[0][0] would yield the wanted result.

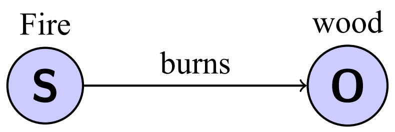
[1]https://dev.mysql.com/doc/connector-python/en/

[2]https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursorbuffered.html

2.4 Resource Description Framework

A **R**esource **D**escription **F**ramework, often abbreviated as **RDF**, is a standardized model which is most commonly used in the context of the Semantic Web. It is used to describe or exchange graph data. In an RDF model, data is represented as a directed graph, which consists out of triple statements (*Resource Description Framework (RDF): Concepts and Abstract Syntax* n.d.).
A triple graph statement is made out of the following components:

- A node for the *subject*
- A node for the *object*
- A *predicate* connecting the two nodes



For each of the triple statements exists such graph relationship, and altogether they make up the RDF model. Statements inside the RDF model that refer to the same subject or object are connected and form a semantic network.

Internally the components of the statement can be represented in different ways. A subject can be either a URI reference or a blank node, an object can be either a URI reference, blank node or a literal and a predicate can exist only as a URI reference(*Resource Description Framework (RDF): Concepts and Abstract Syntax* n.d.). A **Uniform Resource Identifier reference**, often shortened to **URI ref**, is a Unicode string that only is made up from characters out of the ASCII Alphabet(*IETF* n.d.). It is constructed out of five elements: scheme, authority, path, query and fragment.
Altogether, the generic URI syntax looks like this:

foo://example.com:8042/over/there?name=ferret#nose[1]

This shows the similarity to the more popular *URL*, which is a special case of an URI. A *blank node* is a node that is neither a URI ref nor a literal, that's why it is also called an anonymous resource. It contains no information. A *literal* is used to represent values such as numbers, text strings or dates. Literals can also have a data type in order to further specify their value. For example, the value ”42”, could be given the data type xsd.string to be represented as a string or xsd.integer to be seen as a number. It is also possible to add a language tag to a literal(*RDF 1.1 Concepts and Abstract Syntax* 2014).

[1]Example taken from: https://en.wikipedia.org/wiki/Uniform_Resource_Identifier

2.5 RDFLib

RDFLib is an open-source[1] python package created for working with RDF. It allows the user to conveniently add information to a RDF graph and offers different parsers/serializers to work with. It was initially created in 2002 and is still being maintained/updated with the latest major version being published in 2021.

Example:[2]

```python
from rdflib import Graph, URIRef, Literal, FOAF
from rdflib.namespace._XSD import XSD

g = Graph()

g.add((
URIRef("http://example.com/person/nick"),
FOAF.givenName ,
Literal("Nick", datatype=XSD.string)
))

g.serialize(destination="output.nt", format="nt", encoding="utf-8")
```

In this example the structure *Graph* is being introduced, it functions as the primary interface/container when working with RDFLib. It contains all of our triples and allows to perform common set-operations (like add(), to add triples). In this example we are adding one triple to the graph. The triple is made out of a *URI-reference*, the property *FOAF.givenName* and a *Literal*, which was explained in chapter 2.4. FOAF stands for *Friend Of A Friend* and is a vocabulary commonly used in the semantic web to represent people and their relationships. FOAF.givenName is a property in the FOAF vocabulary and used to represent the given name of a person. After adding the triple to the graph, the information can be serialized using the different serializers. In the example N triples was chosen as the serializer which converts the RDF data to the N-Triples format[3]. Each line of an N-Triples file represents a single RDF triple, with each part of the triple separated by a whitespace and terminated with a period.

Output:

```
<http://example.com/person/nick>
<http://xmlns.com/foaf/0.1/givenName>"Nick"^^
<http://www.w3.org/2001/XMLSchema#string> .
```

Note that due to the width could not be displayed as one line.

[1]https://github.com/RDFLib/rdflib
[2]Example taken from the RDFLib documentation
[3]https://www.w3.org/TR/n-triples/

2.6 D2RQ

To create a RDF graph from the database used in the pipeline, D2RQ is used. D2RQ is an abbreviation for *Database to RDF Query* and is an open source mapping tool which allows one to access their data as a RDF graph without having to store it in an RDF format in the database. It consists out of:

- the D2RQ mapping language,
- the D2RQ server and
- the D2RQ engine.

In order to create RDF graphs with D2RQ, firstly a mapping file has to be created (*D2RQ - Getting started* n.d.). The D2RQ mapping language will be further discussed in the following chapter. With the mapping file created, it is now possible for us to map the database content to RDF.

2.6.1 D2R Mapping Language

The D2RQ mapping language is a declarative language used to map the content from the **D4N**[4] database to RDFS vocabularies(*D2RQ mapping language documentation* n.d.) and is thereby responsible for how the virtual RDF graph is structured. When revising the current pipeline it was crucial to have an understanding of the mapping language since its functionality had to be replicated using **RDFLib**. In order to give a better understanding of the pipeline, one needs to discuss how the D2RQ mapping language is used to create the RDF graph. An explanation about the complete RDF graph with images can be found in chapter 4. In the mapping file the most commonly used functions, that are provided by D2RQ, are the *PropertyBridge* and *ClassMap*.

Example - ClassMap:

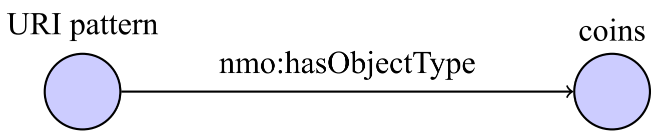
```
map:coins a d2rq:ClassMap;
  d2rq:dataStorage map:database;
  d2rq:uriPattern "https://www.corpus-nummorum.eu/coins/@@data_coins.id@@";
  d2rq:class nmo:NumismaticObject;
  d2rq:condition "data_coins.publication_state = 1";
      .
```

In the example we are creating the ’d2rq:ClassMap’ *coins*, which represents one or multiple classes in the D2RQ mapping language. Instances of the class are being handled in accord to the definition in ’d2rq:ClassMap’ by the D2RQ mapping language (*D2RQ mapping language* *documentation* n.d.). In the next line reference to where the data is stored is given by mapping the database to 'd2rq:dataStorage'. In this case *database* is a variable defined earlier which represents a connection to the database. After this a reference to the resource is given by passing an URI pattern and an RDFS Class is being created. Every instance created from the ClassMap will also be an instance of this class. Lastly we are limiting the mapped coins to those with the value set accordingly, it functions as a SQL WHERE statement. When looking at the mapping language visually, creating this ClassMap means that every entry of the database will be represented in the graph as a node with a specific URI reference. However, only the coins which are ment to published will be mapped, which is the job of the last line.

Example - PropertyBridge:

```
map:individualcoins_coin a d2rq:PropertyBridge;
  d2rq:belongsToClassMap map:Coins;
  d2rq:property nmo:hasObjectType;
  d2rq:uriPattern "http://nomisma.org/id/coin";
  .
```

This example deals with the other common function - *PropertyBridge*. With these functions it is possible to attach information to the classes/resources (*D2RQ mapping language documentation* n.d.). In order to do that a reference to the associated class has to be given. This can be achieved, as seen in the second line, by using *d2rq:belongsToClassMap*. In the following row we can specify the RDF property which should connect the information to the resource. In the last line we can pass the value of the property bridge. Visually the property bridge looks like this:



3 Assignment

3.1 Current state of the pipeline

At the current state the steps of executing the pipeline are associated with a lot of manual work, starting from the manual execution of the NLP notebooks to the running of the D2RQ application. At the beginning of the pipeline the NER is being executed. This is crucial since the RE relies on the output of the NER, which builds the foundation of the RE. Both NER and RE are located in separate python notebooks and have to be started separately. Every design saved in the database is loaded by the notebooks to be worked with and NER is being applied to each of them. The resulting output consists out of a tuples packed in a list. These tuples are made out of the respective design and the found named-entities. Afterwards, RE follows, taking the previous output as an input and attempting to establish a relationship between the found entities of the respective description. With completion of the RE the output takes the form of triples (Subject, Predicate, Object). These are then uploaded to the MySQL database. The last step of the pipeline is the one, which requires the most amount of manual work and thereby offers the most room for improvement. In order to create the RDF data, the program D2RQ is used. As mentioned, D2RQ lets us create the RDF data through a connection between the database and the D2RQ program. It allows us to apply a mapping via the command line, which will map the contents of our database to RDF.

3.2 Process and Challenges

When I first started the planning for this thesis, I faced the challenge of becoming familiar with the D2RQ mapping language and gaining an understanding of the database itself. In order to identify areas for improvement, it was necessary to understand how the different components of the pipeline interacted with each other. However, the large size of the mapping file and database created initial obstacles. By referring to the D2RQ documentation and comparing the output of the pipeline to the mapping file, I gained a better understanding of how the mapping works. Due to the size of the database, the initial output was limited to only one coin, which allowed me to directly identify the origin of RDF triples in the output. Additionally, I divided the output file into different sections based on their subject and then assigned each section the corresponding mapping, which provided a blueprint for the structure of the output. Since the D2RQ mapping file had comments for each mapping, I could use these as an indicator for the mapping.

Example:

```
# Coins identifier
<https://www.corpus-nummorum.eu/CN_3941> <http://purl.org/dc/terms/
    identifier> "coin_id=3941" .
# Coins property bridge
<https://www.corpus-nummorum.eu/CN_3941> <http://nomisma.org/ontology#
    hasObjectType> <http://nomisma.org/id/coin> .
# Coins classmap
<https://www.corpus-nummorum.eu/CN_3941> <http://www.w3.org/1999/02/22-rdf-
    syntax-ns#type> <http://nomisma.org/ontology#NumismaticObject> .

# Coins --> Reverse (map coin_reverse)
<https://www.corpus-nummorum.eu/CN_3941> <http://nomisma.org/ontology#
    hasReverse> <https://www.corpus-nummorum.eu/CN_3941#reverse> .

# Coins --> Obverse (map coin_obverse)
<https://www.corpus-nummorum.eu/CN_3941> <http://nomisma.org/ontology#
    hasObverse> <https://www.corpus-nummorum.eu/CN_3941#obverse> .
```

This knowledge enabled me to take the first step in developing a new version of the pipeline, starting with replicating the functionality of the D2RQ program, with the rest to be programmed later on. This part of the pipeline, called *create_rdf_graph*, will be discussed in more detail in chapter 3.3. While developing the script, I encountered issues with the RDFLib output not being in the same order as the output produced by the D2RQ program, resulting in the tedious process of rearranging the lines. When rearranged I could then compare my output to the expected output line for line to look for any differences. Another obstacle was the fact that some triples in output file that was created by the D2RQ program were doubled and had to be found and deleted manually, making the process more time-consuming. After implementing a major part of the mapping file, I repeated this process and compared the output to ensure that every triple was created by my script. This process, along with the time it took to understand the mapping, took the most time during the initial stages of development. Since I only used basic functions from the RDFLib package, this part of the implementation process ran smoothly and I encountered no major issues. After replicating the D2RQ program, I continued to test the script on other coins. Initially, I only used one coin when developing the script, but when testing it on other coins, I encountered multiple issues with the SQL queries that I was not aware of when dealing with only one coin since not every SQL query from the script was executed or discrepancies in the database were found. This was for example because of a spelling error in the query or the data was not accessed the right way. Furthermore, running the script for all coins allowed me to find formatting errors like ’YearOfEmperor’ instead of ’YearOfTheEmperor’ in the database which resulted in SQL queries not working properly. After reporting these kinds of errors to Sebastian Gampe I could make the necessary changes in the database. With the RDF part of the new pipeline working, I continued the development of a Python notebook that would handle the execution of the NLP notebooks and my script. For this task, I was able to work with the existing code from the NLP notebooks. However, I needed to make some changes. As the user only provides the pipeline with an id, I created a function that loads the coin description based on their id. In the notebooks developed by P. Klinger, every design in the database was loaded, which is not efficient because not every design is needed when one is only working with a subset of coins. This led to the development of the function *load_designs_with_id*, which allows loading designs using the id. Since the previous function was located in the cnt-package, developed by P. Klinger, I implemented the new function in the same location. If the function receives the parameter *all* instead of an array of ids, it used the original function developed by P.Klinger which simply loads all existing designs. When testing all coins in the database the modified pipeline took approximately 2 hours to run with most of the time being spent on *create_rdf_graph* script.

3.3 Implementation of the revised pipeline

To optimize the pipeline, at first the different areas of improvement had to be identified. In order to reduce the required input by the user the pipeline should be automated and require as little as possible human input. Additionally the pipeline should run efficiently and use the minimum amount of resources. To achieve this the functionality of both notebooks was replicated and merged into one notebook - called *start_pipeline*. Furthermore through a new python script *create_rdf_ graph* the function of the D2RQ program was replicated. This python script, like the scripts used in the NLP notebook, was imported into the main notebook and is executed after the NLP. To execute the revised pipeline the user simply has to specify the ids of the coins (either by listing them or entering ”all” as the parameter) and run the notebook.
start_pipeline:
This script handles the whole execution of the pipeline and is the place where the user needs to specify the coins he wants to pass into the pipeline. This is done by either listing the respective ids or by passing ’all’ as a parameter, which then loads all coin descriptions of the database. Firstly a database connection is created using the my-sql.connector package and then the necessary designs are loaded from that database. Since this functionality was only implemented for *all* coins in the previous version a new function, called *load_design_with_id* had to be added into file *io.py*. This function is built out of the existing function *load_designs_fromd_db*, however it now lets the user specify the coin or coins from which the designs will then be loaded. Instead of simply loading all designs, the function now iterates over the given array of coin ids and selects the respective design ids. With these design ids the description can now be selected

and the concatenated dataframe, that contains the descriptions, is returned. Since the output of both function is identical both function can be used for the same thing. Afterwards the models used for the NLP are specified. In the following section the models that are used to apply the NLP to different designs are loaded and the NLP is applied to the loaded designs. Lastly the function *create_graph* is executed which has its origin in the file *create_rdf_graph*.

create_rdf_graph:

The file *create_rdf_graph.py* consists out of the multiple function which all together create the RDF output. Similar to the D2RQ mapping language it takes the data, which was uploaded to the database, and create RDF triples by adding them to a graph. After the process of adding triples is finished, the graph is serialized and with that the output file is created. In the context of the pipeline the function *create_graph* is used as an entry point. This function creates the variables used through out the different functions, like the RDF graph or the MySQL cursors for example. Afterwards the function iterates over the given array of ids and executes the respective functions. First the general information of the coin and the first connections to the obverse/reverse designs are mapped with the function *map_coin*. Secondly, the function *map_designs* is called. This function maps the ids and the descriptions of the obverse and reverse designs to their respective sides. Afterwards the two NLP functions are called which task it is to map the result of the NER and RE - *map_reverse_nlp* and *map_obverse_nlp*. Although they access different tables, both function in the same way. In theory it would have been possible to merge the two functions into one, however this would have come with at the cost that the code would have been less readable since every SQL query would have to be formatted accordingly. Both functions consist of a NER and a RE part. They use the data that was previously uploaded into the tables *cnt_pipeline_ner_url* and *cnt_pipeline_url* to retrieve the data for the triples. The process and the functionality of the functions can be seen well at the following example:

```python
##########################
# Coin general information
##########################
cursor.execute("Select id from data_coins where id = {};".format(int(id)))
query_result = cursor.fetchall()
coin_id = check_for_none(query_result,"Select id from data_coins where id = {};.format(int(id))")
pattern = "https://www.corpus-nummorum.eu/CN_"+ str(coin_id[0][0])

#coin property bridges
g.add((URIRef(pattern), URIRef(prefix_dict["nmo"]+"hasObjectType"), URIRef(prefix_dict["nm"]+"coin")))
g.add((URIRef(pattern), URIRef(prefix_dict["dcterms"]+"identifier"), Literal("coin_id="+str(id))))
g.add((URIRef(pattern), URIRef(prefix_dict["rdf"]+"type"), URIRef(prefix_dict["nmo"]+"NumismaticObject")))
# Coin -> obverse_coin
g.add((URIRef(pattern), URIRef(prefix_dict["nmo"]+"hasObverse"), URIRef("https://www.corpus-nummorum.eu/CN_{}#obverse".
  format(str(id)))))
# Coin -> reverse_coin
g.add((URIRef(pattern), URIRef(prefix_dict["nmo"]+"hasReverse"), URIRef("https://www.corpus-nummorum.eu/CN_{}#reverse".
  format(str(id)))))
```

In the example you can see a snippet of the *map_coin* function. In the first section the necessary informations are retrieved using the MySQL.connector package and its cursor. With the cursor we are able to retrieve the data from the database using SQL Querys. Afterwards the data has to be retrieved from the cursors using the function *.fetchall()*. The output is then saved to a variable. Above we are retrieving the coin id from the database, which makes sure that the given id actually exists as a coin the database. With the id saved to the variable *coin_id*, the URI-pattern used in the triples is defined. Lastly the triples are created using the information we retrieved before. The comments display the mapping which was originally used in the D2RQ mapping file. For example ”Coin -> obverse_coin” refers to the connection between the *coin node* and the *coin_obverse node*. Due to that the user should be able to easily identsify what entities are being mapped in the lines following the comment. After completion the iteration process the last triples are added with the functions *create_hierarchy* and *create_prop_class*. All the triples created by these two functions exist independently from the input IDs and are always part of the output. Since they do not depend on the input, it is not necessary for the functions to be included into the iteration process. The last step of the script and thus the last step of the pipeline is to serialize the graph. After that the output file can be found in the folder where the *start_pipeline* script is included.

How to change the mapping:
Depending on what should be changed in the mapping, either **(a)** one of the existing functions has to be modified or **(b)** a new function containing the desired mapping has to be created. At the moment the respective mapping functions are executed after each other in the main function and any new mapping function has to be added in order to be executed by the script. It is also possible to comment out functions in order to avoid a part of the mapping. When it comes to (a), the dividing of the mapping process into multiple functions and adding in-depth comments improves the readability of the code and allowing the user to easily identify where what part of the mapping is located. After locating the part that should be changed, it is possible to change the mapping in two different ways - either modify the SQL query or modify the RDFLib statement. Modifying the SQL query changes what data is retrieved from the database and modifying the RDFLib statement changes how the retrieved data is accessed. When it comes to (b), the new function has to be passed certain arguments (for example the RDF graph or the MySQL cursor). The mapping process then functions in the same manner as explained in (a). Depending on what information from the database should be mapped, a SQL query retrieving that information has to be created. Afterwards, the information can be processed and added as a triple to the graph by using the RDFLib library.

4 Results

4.1 Output

In this chapter I would like to describe the resulting RDF graph created by the pipeline. Initially, each coin is assigned to the class Coin, which means that the coin is represented as a node in the graph. Furthermore, each instance, i.e. each coin, now has a URI pattern that refers to the corresponding page in the Corpus Nummorum Online. This URI takes the form 'https://www.corpus-nummorum.eu/CN_id', where *id* is the respective id of the coin. The URIs appended in the RDF are specific to the CN and have their origin there. Additional information, such as the respective id, is also assigned to the coins. In the next step, the respective designs, meaning the descriptions of the images, are assigned. A distinction is made between the obverse and the reverse, which is also reflected in the RDF graph. There is a node in the graph for both sides. These two nodes are connected to the coin via the properties 'hasObverse' and 'hasReverse', which assign the respective side to the coin. Further information that is appended now either relates to the entire coin or to the respective side. As with the coin in general, information such as the actual description (German/English), id, publisher, and title of the design is also attached to the designs.

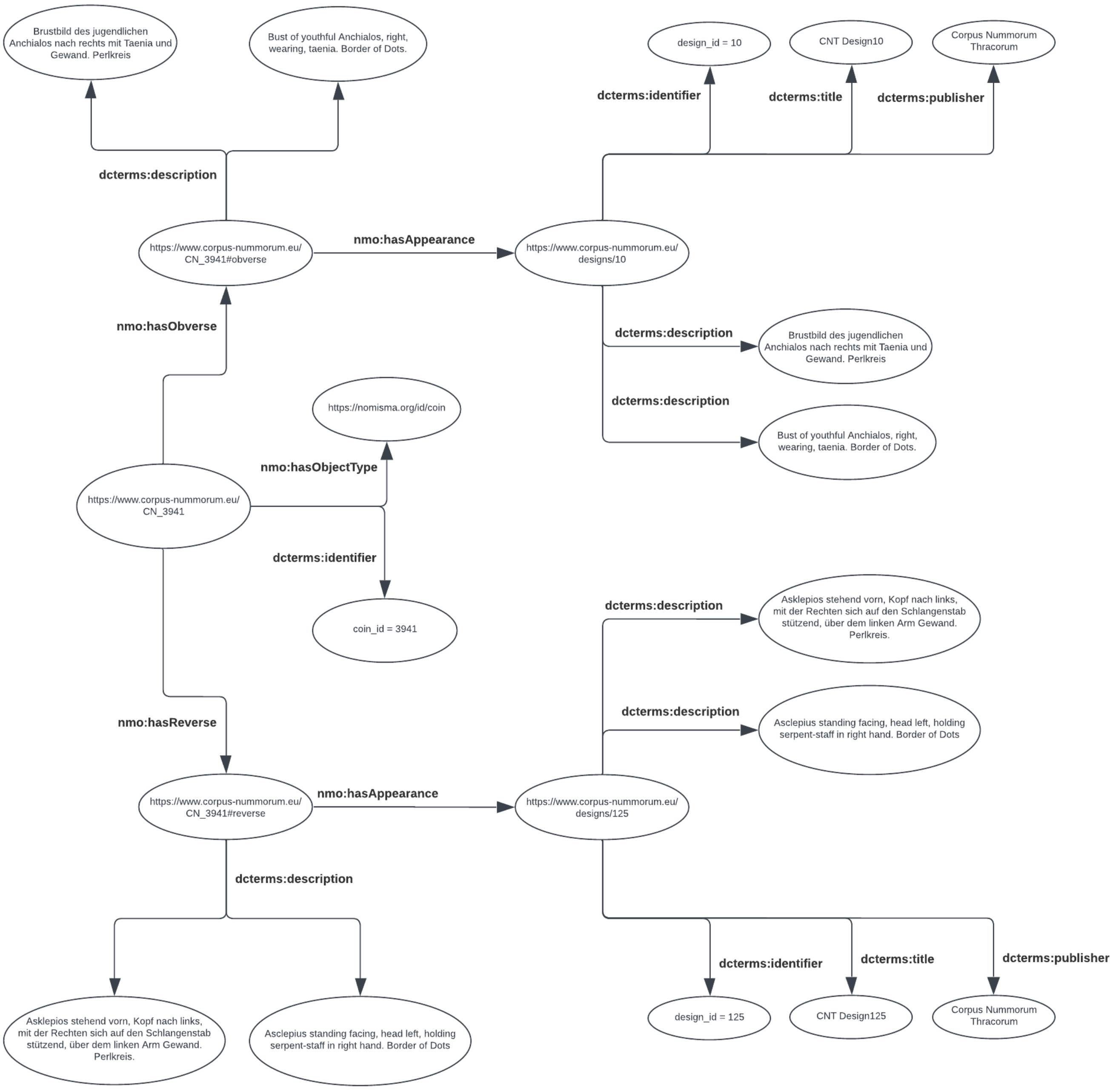
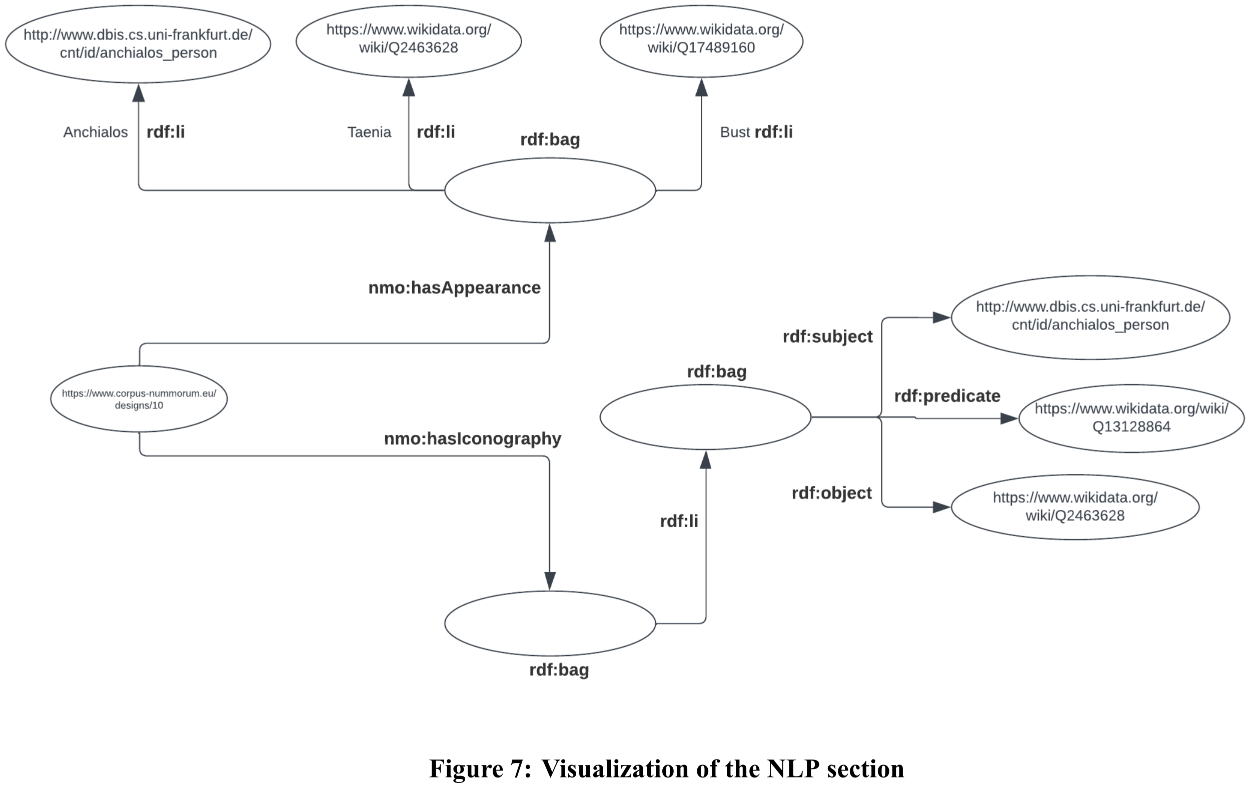

Brustbild des jugendlichen Anchialos nach rechts mit Taenia und Gewand. Perlkreis
Bust of youthful Anchialos, right, wearing, taenia. Border of Dots.
design_id = 10
CNT Design10
Corpus Nummorum Thracorum
dcterms:identifier
dcterms:title
dcterms:publisher
dcterms:description
https://www.corpus-nummorum.eu/CN_3941#obverse
nmo:hasAppearance
https://www.corpus-nummorum.eu/designs/10
nmo:hasObverse
dcterms:description
Brustbild des jugendlichen Anchialos nach rechts mit Taenia und Gewand. Perlkreis
dcterms:description
Bust of youthful Anchialos, right, wearing, taenia. Border of Dots.
https://nomisma.org/id/coin
nmo:hasObjectType
https://www.corpus-nummorum.eu/CN_3941
dcterms:identifier
coin_id = 3941
dcterms:description
Asklepios stehend vorn, Kopf nach links, mit der Rechten sich auf den Schlangenstab stützend, über dem linken Arm Gewand. Perlkreis.
dcterms:description
Asclepius standing facing, head left, holding serpent-staff in right hand. Border of Dots
nmo:hasReverse
https://www.corpus-nummorum.eu/CN_3941#reverse
nmo:hasAppearance
https://www.corpus-nummorum.eu/designs/125
dcterms:description
dcterms:identifier
dcterms:title
dcterms:publisher
design_id = 125
CNT Design125
Corpus Nummorum Thracorum
Asklepios stehend vorn, Kopf nach links, mit der Rechten sich auf den Schlangenstab stützend, über dem linken Arm Gewand. Perlkreis.
Asclepius standing facing, head left, holding serpent-staff in right hand. Border of Dots

Figure 6: Visualization of the general part.

Now, the results of the NLP are added to the RDF graph. For each side of the coin, two nodes are attached to the respective design, which function as containers for the NLP results. They have the class *rdf:bag*[3], one of three possible container classes in RDF. In the context of the pipeline, the bags themselves consist of a so-called blank node and do not carry any information. They are used to establish a connection between the NLP results and the coin sides. The connection is made once through the property 'hasIconography' and once through 'hasAppearance'. Additionally, it is the task of these bags to store all the entries that were uploaded to the database by the NLP. Starting from the blank node, which can be reached via 'nmo:hasIconography', the various *entries* are attached. The entries themselves consist of three different nodes, representing subject, predicate, and object and containing the results from the RE. In the case of coin 3941, the appropriate URI for the entities 'Anchialos', 'wearing', and 'taenia' are now attached to it. Furthermore, the URIs for the entities 'Asklepios', 'holding', 'serpent' and 'Asklepios', 'holding', 'staff' are attached to the other design. The results from the NER are held by the bag connected by 'nmo:hasAppearance'. Each found named entity gets its own node, which is connected to the bag via 'rdf:li'.



http://www.dbis.cs.uni-frankfurt.de/cnt/id/anchialos_person
https://www.wikidata.org/wiki/Q2463628
https://www.wikidata.org/wiki/Q17489160
Anchialos
rdf:li
Taenia
rdf:li
rdf:bag
Bust rdf:li
nmo:hasAppearance
http://www.dbis.cs.uni-frankfurt.de/cnt/id/anchialos_person
rdf:subject
https://www.corpus-nummorum.eu/designs/10
rdf:bag
rdf:predicate
https://www.wikidata.org/wiki/Q13128864
nmo:hasIconography
rdf:object
https://www.wikidata.org/wiki/Q2463628
rdf:li
rdf:bag

Figure 7: Visualization of the NLP section

[3]https://www.w3.org/TR/rdf-schema/#ch_bag

4.2 Comparison to the previous pipeline

In comparison to the old version of the pipeline, the new version improves the old version by automating the whole execution process of the NLP notebooks and the RDF creation part. Thereby, the required human input is minimized. The D2RQ mapping has now been replaced by an alternative, which is better maintainable and expandable due to its implementation in python and the simplicity of the script. In cases where the RDF output for a specific subset of coins is required: It can now be accomplished more conveniently through the use of the *create_rdf_graph* script. By passing the corresponding coin id's to the script, the output can be generated. This functionality streamlines the data creation process, eliminating the need for adjusting the mapping file and searching of entity id's. Additionally, the NLP part of the previous pipeline version was integrated into the *start_pipeline* script, which automatically executes the NLP for the respective coin descriptions. Regarding the execution time, the D2RQ program required 3 hours and 33 minutes to create the RDF data for all coins in the database on a system equipped with an Intel Core i5-2500K @ 3.30 GHz and 16 GB of DDR3 RAM. In comparison, the updated version accomplished the same task in 1 hour and 42 minutes, which includes the execution of the NLP notebooks.

For an easier comparison, I converted the output of both pipelines from N-Triples to Graphviz using this[4] website. Afterwards, I loaded both graphs into the program Gephi[5] to get an overview of both graphs. In Gephi I then used the 'Yifan Hu' layout on both graphs. Below you can find the main part of both RDF graphs.

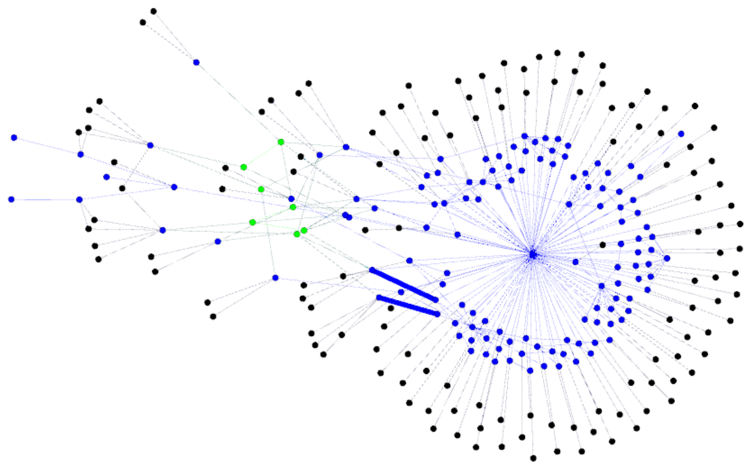


Figure 8: Visualization of the D2RQ output for coin 3941

[4]https://www.easyrdf.org/converter
[5]https://gephi.org/

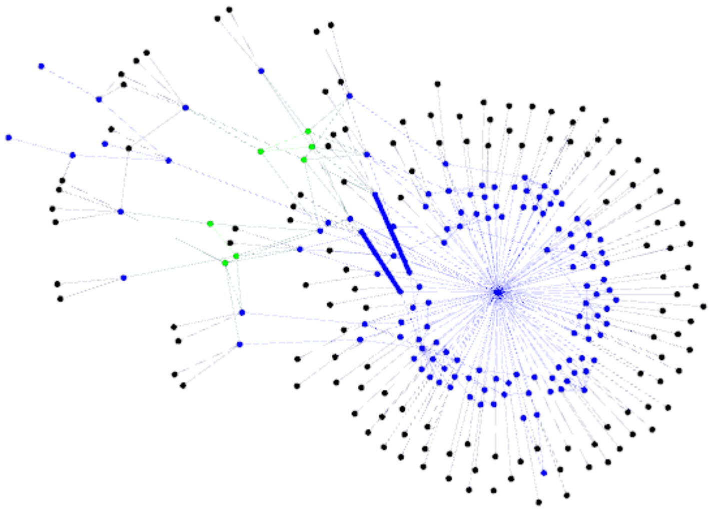


Figure 9: Visualization of the create_rdf_graph output for coin 3941

As seen in both images, the graphs are made out of three different types of coloured nodes. Overall, there are 282 nodes in both graphs with 7 being green, 144 being blue and 131 being black. The green nodes represent blank nodes. In the mapping these nodes, act as containers for the result of the NLP. The black nodes are used to highlight literals. Lastly, the blue nodes represent URI references. Most of the blue nodes are centred around a single node '*rdfs:Class*' which, by using the connection '*rdf:type*', gives the different classes their types. These different classes can be seen as the 'blue ring' that surrounds the centre node. Around the blue nodes, and forming a 'black ring' are the nodes with their origin in the *nlp_hierarchy* table. These nodes are mostly connected to the blue nodes via '*skos:prefLabel*'.

Branching out on the top left of both graphs one can see the nodes related to the design of both reverse and obverse of the coin. Also the general information of the coin is located, and the blank nodes are located there. The nodes which contain the URI reference to the designs can be seen in both nodes as the two nodes in the top left. Afterwards the side (obverse/reverse) is being assigned to them, along with further information. Indicated as rectangular shapes are *parallel edges*. Parallel edges occur if there is more than one edge between two nodes. These can carry different weights in Gephi, which results in the edge being displayed thicker. As an example: In our case this happens once, because the word 'staff', with the URI reference: `https://www.wikidata.org/wiki/Q10971443`, is connected to the class 'Tools' multiple times.

Once because the word ’staff’ is recognized by the NER and thereby it’s category(’Tools’) is added to graph.

```
<https://www.wikidata.org/wiki/Q10971443>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<https://www.wikidata.org/wiki/Q39546> .
```

And once again, because ’Staff’ is part of the *nlp_hierarchy* table and a subclass of ’Tools’:

```
<https://www.wikidata.org/wiki/Q10971443>
<http://www.w3.org/2004/02/skos/core#prefLabel>
"Staff"^^<http://www.w3.org/2001/XMLSchema#string> .

<https://www.wikidata.org/wiki/Q10971443>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/2000/01/rdf-schema#Class> .

<https://www.wikidata.org/wiki/Q10971443>
<http://www.w3.org/2000/01/rdf-schema#subClassOf>
<https://www.wikidata.org/wiki/Q39546> .
```

5 Conclusion and outlook

At the beginning, the objective of this bachelor thesis was defined as optimizing the existing version of the pipeline. The aspects requiring improvement were identified as the manual execution of the pipeline and the utilization of the D2RQ program for the generation of RDF data. Thereby, the task of this thesis was (a) to minimize the required human input and (b) to replicate the functionality of the D2RQ program in such a way that it is easier maintainable in the future and no complex mapping file is required. (a) was achieved through the merging of both NLP notebooks, which were initially developed by P. Klinger in her thesis, into the script ”start_pipeline” and the development of a new function - *load_design_with_id* that allowed to load the design for the specified ids. This reduced the required input to entering the desired coin ids and starting the script.
When it comes to point (b), this thesis introduced the script ”create_rdf_graph” which in the revised version of the pipeline handles the creation of the RDF data. It makes use of the libraries *mysql.connector*and *RDFLib* to do so. With the first package the script retrieves the data from the MySQL database using SQL queries, afterwards the data is processed and then added to the RDF using the second library.

With completion of the mapping process, the script then creates the RDF output. In chapter 4.2 the differences between the two pipeline version were highlighted, showing the improvement of the RDF creation process due to the new pipeline. Overall, improvements were not only made in the previously mentioned areas, but also in terms of the execution time. In the future, the pipeline can be improved in different ways. Since most of the execution time is spent on the ”create_rdf_graph” script, cutting down the execution time of this script would yield the most improvement. Additionally, making the script previously mentioned script even more readable and reduce amount of lines would improve the maintainability.

Appendix

A USB stick containing the implementations of the pipeline discussed in this thesis is attached. The *start_pipeline* script can be found in the ’code’ folder and the *create_rdf_graph* script and the function *load_design_with_id*, located in the file io.py, can both be found in the ’cnt’ folder. Furthermore, a SQL file containing the database, the Gephi files for the visualization and the D2RQ mapping file are included.

6 List of figures

7 Literature

Bird, Klein, Loper (2009). *Analyzing Text with the Natural Language Toolkit*. https://www.nltk.org/book/, Accessed: 01.04.2023.

Corpus Nummorum site (n.d.). `https://www.corpus-nummorum.eu/about`. Accessed: 19-03-23.

D2RQ - Getting started (n.d.). `http://d2rq.org/getting-started`. Accessed: 03-04-23.

D2RQ mapping language documentation (n.d.). `http://d2rq.org/d2rq-language`. Accessed: 03-04-23.

D4N4 project site (n.d.). `http://www.bigdata.uni-frankfurt.de/d4n4/`. Accessed: 18-03-23.

Deligio, Chrisowalandis and Kerim Gencer (2021). *Natural Language Processing auf mehrsprachigen Münzdatensätzen*. Master Thesis.

IETF (n.d.). `https://www.ietf.org/rfc/rfc2396.txt`. Accessed: 03-04-23.

Klinger, Patricia (2018). *Natural Language Processing to enable semantic search on numismatic description*. Bachelor Thesis.

MySQL Connector/Python Developer Guide (n.d.). `https://dev.mysql.com/doc/connector-python/en/`. Accessed: 22-02-23.

RDF 1.1 Concepts and Abstract Syntax (2014).

Resource Description Framework (RDF): Concepts and Abstract Syntax (n.d.). `https://www.w3.org/TR/rdf-concepts`. Accessed: 03-04-23.