POZNAN UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTING AND TELECOMMUNICATIONS
Institute of Computing Science

Doctoral Dissertation

AUTOMATING COMPETENCY QUESTIONS HANDLING IN ONTOLOGY DEVELOPMENT PROCESSES

Dawid Wiśniewski

Supervisor
Agnieszka Ławrynowicz, Ph.D., Dr. Habil.
Supporting Supervisor
Jędrzej Potoniec, Ph.D.

POZNAŃ 2022

Abstract

Ontologies are widely used to describe a given domain of interest in terms of formally specified concepts and relationships between them. They help solve tasks such as question answering, information integration, information extraction, or text disambiguation. Nowadays, engineers frequently model them using Web Ontology Language (OWL) – a language based on description logics, which allows inferring new knowledge that is implicitly defined in a given ontology. This ability makes them powerful tools that mimic human-like reasoning. However, it makes ontologies hard to construct.

People struggle with constructing ontologies. As the domains they represent are frequently big, involving hundreds or thousands of concepts and relationships, it becomes crucial to help engineers assess the completeness of the modeled knowledge. Moreover, the inferring abilities make it possible that the logical consequences of the constructed ontology will be incorrect, and engineers often find it hard to predict these consequences by themselves. For these reasons, it is beneficial to provide engineers with ways of controlling the quality of the models they build.

Many ontology engineering methodologies use the notion of competency questions (CQs) to refer to questions stated in natural language that a complete and correct ontology should be competent to answer correctly. Competency questions serve as a source of vocabulary that engineers should represent in an ontology. Later, during actual development, engineers formalize them using an appropriate query language (e.g., SPARQL-OWL) to check the quality of their ontology. They observe which questions cannot be answered or return wrong answers to see which parts of ontology they should improve.

As the CQs handling has been mainly a manual process so far, in this dissertation, we address the problems of automating competency questions handling in ontology development processes. We propose two areas of automatization. The first one relates to vocabulary extraction that aims to provide a list of concepts and relationships an ontology should define as classes, instances, and properties. We describe two methods providing automatic suggestions for terms to be modeled extracted from a set of CQs. The first method is based on a machine learning model, and the second one is based on handcrafted rules expressed using sequences of POS tags. We show that the rule-based approach works better and generalizes to requirements stated as statements.

The second area of automatization refers to the automatic translation of CQs into SPARQL-OWL. In this dissertation, we motivate the need for SPARQL-OWL and show why popularly used SPARQL is insufficient. Then, we propose a method that recommends SPARQL-OWL queries for CQs based on the knowledge modeled in an ontology. Finally, we show how this tool can be integrated with an existing test-driven development ontology engineering approach. We evaluate the method and show that it works on unseen datasets.

We decided to build the translator in a data-driven approach. We provide a pipeline-based method, motivated by the analysis of pairs of CQs translated into SPARQL-OWL queries. As no such dataset was available before our work, we collected a set of 234 CQs stated for five different ontologies and constructed queries manually for 131 of them. In this dissertation, we analyze the dataset and search for regularities among the collected questions and queries.

However, the dataset we collected is rather small and does not cover many possible forms of CQs and queries. For this reason, we provide a method for generating pairs of CQ patterns and SPARQL-OWL templates automatically based on a set of axiom patterns. We use frequent axiom patterns extracted from BioPortal to construct a synthetic dataset of almost 78,000 CQ patterns mapped to 575 query templates. Since the dataset is based on frequently recurring axioms, it is expected to cover the most popular modeling decisions. The pairs of patterns and templates can be filled with IRIs and labels to create large sets of CQs and SPARQL-OWL queries.

We hope that the datasets and methods introduced in this dissertation will make the usage of CQs more prevalent since the datasets we propose may guide authors on how to formulate CQs, and the introduced tools may ease the ontology development.

Streszczenie

Niniejsza rozprawa dotyczy problemów automatyzacji obsługi pytań kompetencyjnych w procesach wytwarzania ontologii. Ontologie są szeroko stosowane do opisu zadanej dziedziny w formie zbioru formalnie określonych pojęć i relacji między nimi. Pomagają one rozwiązywać zadania, wśród których wyróżnić można: automatyczne odpowiadanie na pytania, integrację informacji, ekstrakcję informacji czy ujednoznacznianie tekstu. Obecnie inżynierowie najczęściej tworzą je przy użyciu języka Web Ontology Language (OWL), który bazuje na logikach deskrypcyjnych. Wykorzystanie języków modelowania bazujących na logice formalnej pozwala na zastosowanie mechanizmów wnioskowania, dzięki którym istnieje możliwość odkrycia nowej, niejawnie zdefiniowanej w danej ontologii, wiedzy. Ta zdolność czyni z ontologii potężne narzędzia, które naśladują ludzki sposób rozumowania.

Wytwarzanie ontologii stanowi wyzwanie dla inżynierów. Dziedziny, które są przez nich opisywane, nierzadko są obszerne i obejmują setki lub tysiące klas i relacji wymagających sformalizowania. Z tego powodu, kluczowym wydaje się być wsparcie inżynierów w procesie identyfikacji bytów do zamodelowania i pomoc w ocenie kompletności formalizowanego słownictwa.

Co więcej, zdolność ontologii do wykorzystania mechanizmów wnioskowania powoduje, że bardzo istotną kwestią staje się zapewnienie, aby logiczne konsekwencje skonstruowanej ontologii były poprawne. Z tego powodu ważnym aspektem staje się zapewnienie inżynierom sposobów kontrolowania jakości budowanej reprezentacji, ponieważ często mają oni trudności z przewidywaniem logicznych konsekwencji modelowanej wiedzy.

Wiele metodyk z obszaru inżynierii ontologii wykorzystuje pojęcie pytań kompetencyjnych, które formułowane są w języku naturalnym i na które kompletna i poprawna ontologia powinna być w stanie poprawnie odpowiedzieć. Pytania kompetencyjne służą jako źródło słownictwa, które inżynierowie powinni zamodelować w ontologii. Podczas właściwego modelowania, inżynierowie formalizują pytania kompetencyjne przy użyciu odpowiedniego języka zapytań (np. SPARQL-OWL), aby zweryfikować jakość ontologii. Poprzez obserwację, na które z pytań nie można udzielić odpowiedzi z uwagi na niekompletne słownictwo bądź na które pytania ontologia zwraca niepoprawne odpowiedzi, inżynierowie dowiadują się, które obszary modelowanej wiedzy wymagają poprawy.

Dotychczas obsługa pytań kompetencyjnych była głównie procesem manualnym, dlatego w niniejszej rozprawie zaproponowano dwa obszary automatyzacji ich wykorzystania. Pierwszy z nich stanowi zadanie wykorzystania pytań kompetencyjnych jako źródła słownictwa, które powinno zostać zamodelowane w ontologii. Przedstawiono dwie zautomatyzowane metody dostarczające sugestie klas, instancji i właściwości do zamodelowania na podstawie zadanego zestawu pytań kompetencyjnych. Pierwsza z metod oparta jest na uczeniu maszynowym i stanowi nadzorowany model warunkowych pól losowych. Druga z nich oparta jest na ręcznie przygotowanych regułach wyrażonych w formie sekwencji znaczników części mowy. W niniejszej pracy porównano oba podejścia i pokazano przewagę systemu regułowego, który generuje wyniki wyższej jakości i uogólnia się również do scenariusza wykrywania słownictwa ze zdań oznajmujących.

Drugi z obszarów automatyzacji obejmuje proces formalizowania pytań kompetencyjnych do postaci języka zapytań SPARQL-OWL. W niniejszej rozprawie przedstawiono potrzebę wykorzystania języka SPARQL-OWL, a następnie zaproponowano metodę, która rekomenduje formy zapytań SPARQL-OWL dla zestawu pytań kompetencyjnych przy udziale wytwarzanej ontologii. Przedstawiono również sposób integracji tej metody z istniejącym podejściem do rozwijania ontologii wykorzystującym testy. W rozprawie tej przeprowadzono analizę jakości narzędzia implementującego metodę na niewidzianym wcześniej zestawie pytań kompetencyjnych i ontologii.

Zaproponowana metoda translacji pytań kompetencyjnych do formy zapytań SPARQL-OWL skonstruowana jest na podstawie wniosków zebranych podczas analizy zbioru przykładów pytań kompetencyjncyh sformalizowanych w postaci zapytań SPARQL-OWL. Ponieważ nie istniał dotychczas żaden zbiór danych zawierający tego typu translacje, w ramach rozprawy zebrano zbiór 234 pytań kompetencyjnych zdefiniowanych dla pięciu ontologii i ręcznie przygotowano zapytania dla 131 pytań. W rozprawie przeanalizowano właściwości tego zbioru i zidentyfikowano regularności wśród pytań, zapytań i relacji pomiędzy obiema formami. Regularności te pozwoliły opracować metodę rekomendującą zapytania dla zadanych pytań.

Ponieważ zebrany zbiór danych cechował się niewielkimi rozmiarami i nie zawierał reprezentacji wielu możliwych form pytań kompetencyjnych i zapytań, w niniejszej pracy zaproponowano również metodę automatycznego generowania par wzorców pytań kompetencyjnych i szablonów SPARQL-OWL ze zbioru wzorców aksjomatów. Wykorzystano częste wzorce aksjomatów wyodrębnione z serwisu BioPortal, aby skonstruować syntetyczny zbiór danych składający się z niespełna 78 tysięcy wzorców pytań kompetencyjnych w relacji do 575 szablonów zapytań. Ponieważ wzorce aksjomatów reprezentują częste formy wykorzystywane do budowania aksjomatów, efekt działania metody pokrywa najpopularniejsze decyzje dotyczące zarówno sposobów modelowania wiedzy jak i formy pytań.

Żywimy nadzieję, że badania przeprowadzone w tej rozprawie sprawią, że wykorzystanie pytań kompetencyjnych będzie bardziej rozpowszechnione, gdyż zebrane zbiory danych mogą pomóc autorom ontologii w formułowaniu pytań kompetencyjnych i ich wykorzystaniu, a proponowane narzędzia mogą pomóc w zwiększeniu produktywności inżynierów wytwarzania ontologii.

Podziękowania

Niniejsza rozprawa nie mogłaby powstać bez ogromu pomocy bliskich mi osób. Chciałbym w tym miejscu serdecznie podziękować moim promotorom: dr hab. inż. Agnieszce Ławrynowicz, prof. PP i dr. inż. Jędrzejowi Potońcowi za niezliczone ilości godzin wspólnie spędzonych nad planowaniem, motywowaniem, doradzaniem i weryfikowaniem postępów. Dzięki waszej życzliwości nauczyłem się, że warto walczyć do samego końca, nawet jeśli cel wydaje się nieosiągalny.

Chciałbym również podziękować rodzicom, Eli i Andrzejowi, za życzliwość i zrozumienie podczas przygotowywania niniejszej pracy.

Specjalne podziękowania kieruję także ku żonie – Sandrze, za miłość, wyrozumiałość, wsparcie i obecność przy mnie w każdym momencie, a także moim dzieciom: Kornelce, za ciągłe przypominanie o tym, jak ciekawy jest świat i Piotrusiowi, za uśmiech i radość czerpaną z codzienności.

Sandrze, Kornelii i Piotrowi
Gieni, w 100 rocznicę urodzin

Contents

TABLE 1: Table of symbols

Symbol	Meaning	Page
$\{\cdot\}$	Set	19
$\|\cdot\|$	Norm	36
$\lvert\cdot\rvert$	Absolute value	22
$\times$	Cartesian product	8
Σ	Sum	27
Π	Product	27
∇	Gradient	20
$\mathbf{H}$	Hessian	20
∂	Partial derivative	20
$\emptyset$	Empty set	109
$\subseteq$	Subset	34
$argmax$	Argument of the maxima	109
$log(\cdot)$	Logarithm	21
$exp(\cdot)$	Exponential	27
$cos(\cdot)$	Cosine	36
$Jaccard(\cdot)$	Jaccard similarity	107
$\in$	Set membership	8
$\mapsto$	Mapping	60
$\forall$	Universal quantifier	70
$\exists$	Existential quantifier	109
$\cap$	Intersection of sets	107
$\cup$	Union of sets	107
I_{RDF}	Set of all IRIs in RDF graph	8
L_{RDF}	Set of all literals in RDF graph	8
B_{RDF}	Set of all blank nodes in RDF graph	8
T_{RDF}	Set of all terms in RDF graph	8
V_{RDF}	Set of query variables	14
G_{RDF}	RDF graph	8
L	Entity labels set	68
ph	Competency Question placeholder	68
$m(\cdot)$	Mapping function	70
CE	Class expression	88
LHS	Left-hand side	88
RHS	Right-hand side	88
$VERB$	Main verb	88
$\mathbb{O}$	Ontology	68
cq	Competency Question	68
D	Dataset	19
D_{docs}	Documents	31
$\mathbf{x}$	Feature vector	19
y	Object label	19
$\hat{y}$	Prediction	19
$\mathbf{w}$	Parameters vector	20

$\mathbf{h}$	Hidden state vector	25
$\mathbf{U}, \mathbf{V}, \mathbf{W}$	Parameters matrices	25
$h(\cdot)$	Hypothesis	20
$g(\cdot)$	Activation function	25
$\ell(\cdot)$	Loss function	20
$J(\cdot)$	Cost function	20
d	Document (sequence of tokens)	31
$\mathbf{d}$	Vector representation of document d	31
t	Token	31
V	Vocabulary	31
η	Learning rate	20
$\mathbf{y}_n$	Vector of gradient changes	21
$\mathbf{s}_n$	Vector of parameter changes	21
ρ_n	Product of gradient and parameter changes	21
λ	Regularization weight	22
$f_k(\cdot)$	CRF k-th feature function	27
P	Precision	29
R	Recall	29
F_1, F_β	Harmonic averages of precision and recall	29
β	Precision-Recall trade-off score in F_β	29
$\mathbf{a}, \mathbf{b}$	Real-valued vectors	36
A, B	Sets	107
Q_{RE}	Finite set of states	34
Σ_{RE}	Alphabet	34
δ_{RE}	Transition function	34
q^0_{RE}	Initial state $q^0_{RE} \in Q_{RE}$	34
F_{RE}	Accept states set $F_{RE} \subseteq Q_{RE}$	34
l	Entity label	68
cl	Entity label in context	110
S	SPARQL-OWL query signature	60
S_Q	SPARQL-OWL query signature created from the query Q	60
$\$PPx\{NUM\}\$$	SPARQL-OWL query placeholder	64
e	Ontological entity	109
L_{nthing}	Set of normalized class and individual labels	108
L_{nprop}	Set of normalized object and data property labels	108
$lhs(\cdot)$	Function returning left-hand sides of assertions	109
$rhs(\cdot)$	Function returning right-hand sides of assertions	109
p	Phrase extracted using ReqTagger	108
$\mathbf{p}$	Phrase vector of p	108
γ	SeeQuery‘s threshold parameter	108
$prop$	Property	108
C, D	Classes	108
i	Individual	109
CT	Set of triples each representing subject, property, and object	109
$domain(\cdot)$	Domain of a given property	108
$range(\cdot)$	Range of a given property	108

$super(\cdot)$	Superclasses	108
$\tau(prop)$	Set of pairs of classes seen with *prop*	109
θ	SeeQuery trade-off parameter	109
$embedding(\cdot)$	Function generating phrase vectors from text	109
$label(\cdot)$	Function extracting ontology labels for given IRI	109
$translate(\cdot)$	Function extracting possible translations for argument	108
Q	SPARQL-OWL query	115
PQ	Presupposition query	115
PQ^+	Positive presupposition query	115
PQ^-	Negative presupposition query	115
$\mu(\cdot)$	Function returning query answer	115

Table 2: Table of abbreviations and proper names

Abbreviation/Proper name	Explanation
ACE	Attempto Controlled English
AI	Artificial intelligence
AWO	African Wildlife Ontology
BERT	Bidirectional Encoder Representations from Transformers
BigCQ	Dataset of CQ patterns formalized into SPARQL-OWL templates
CE	Class expression
CORAL	Dataset of ontological requirements
CQ	Competency question
CQ2SPARQLOWL	Dataset of CQs translated into SPARQL-OWL queries
CRFs	Conditional Random Fields
CWA	Closed-world assumption
Dem@Care	Dementia Ambient Care Ontology
DFA	Deterministic finite automaton
EC	Entity chunk
FSA	Finite-state automaton
IRI	Internationalized Resource Identifier
L-BFGS	Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm
LHS	Left-hand side
ML	Machine learning
NFA	Nondeterministic finite automaton
NLP	Natural language processing
NLTK	Natural Language Toolkit (`https://www.nltk.org`)
ODP	Ontology Design Pattern
OntoDT	Ontology of Datatypes
OWA	Open-world assumption
OWL	Web Ontology Language
PC	Predicate chunk
Pizza	Pizza Ontology
POS	Part of speech
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
ReqTagger	Tagger for automatic glossary of terms suggestions extraction
RHS	Right-hand side
SeeQuery	Automatic method for translating CQs into SPARQL-OWL
spaCy	Library for Natural Language Processing (`https://spacy.io`)
SPARQL	SPARQL Protocol and RDF Query Language
SPARQL-OWL	SPARQL with OWL Direct Semantics Entailment Regime
SPO	Subject-Predicate-Object
SS	Subclass-Superclass
Stuff	Stuff Ontology
SWO	Software Ontology
TDD	Test-Driven Development
TrhOnt	Ontology assisting rehabilitation processes
UNA	Unique name assumption
W3C	World Wide Web Consortium

Chapter 1

Introduction

1.1 Historical context

The term ontology is known to humankind since the early 17th century. It was first used in Latin texts of two German philosophers: Jacob Lorhard, who in his work entitled *Ogdoas Scholastica* [103] used the word ontology as a synonym to metaphysics – the study of being introduced by Aristotle almost 2500 years ago [5], and Rudolphus Goclenius, who in *Lexicon philosophicum, quo tanquam clave philosophiae fores aperiuntur* [62] defined ontology as the philosophy of being.

During the subsequent centuries, philosophers tried to answer questions like *What does exist?*, *How do things relate to each other?* to understand reality better.

In the 1950s, these problems came into focus again due to the influential publications of Willard Van Orman Quine [139], who discussed the criterion of ontological commitment and made the famous statement that *To be is to be the value of a bound variable.*

Soon after the works of Quine, in 1956, Dartmouth Summer Research Project on Artificial Intelligence gave rise to the field of Artificial Intelligence (AI), which focuses on giving machines the ability to perform tasks associated with intelligent beings [122]. One of the paradigms proposed to solve problems of AI is the use of formal logic to represent knowledge so that logical conclusions can be drawn automatically. This kind of approach is named logic-based AI, with John McCarthy being its most famous advocate. His computer program called Advice Taker [111] used commonsense and deduced the consequences of already known facts. McCarthy believed that intelligent systems that are based on logic should use ontology to list things that exist [112].

During the subsequent decades, the knowledge engineering field emerged, and the term ontology became widespread in AI [163]. In 1993, Thomas Gruber attempted to formalize the definition of ontology tailored to the field of AI [64] as a formal specification of conceptualization. In 2001, the idea of the Semantic Web was coined by Sir Tim Berners-Lee [14]. The Semantic Web, an extension of the World Wide Web, focuses on modeling content meaningful to computers. Here, the notion of ontology is one of the key concepts as domain-related ontologies define concepts and relations that can be understood, shared, and processed automatically by computers. Ontologies are also adapted to problems of data integration from multiple sources or question answering, among others.

1.2 Motivation

Nowadays, ontologies are often expressed in the Web Ontology Language (OWL), which can be used to define knowledge with the use of capabilities given by Description Logics [7]. Most types of Description Logics are decidable fragments of the First-Order Logic.

However, it is not easy to construct ontologies as they often need to cover large domains with thousands of concepts interlinked with multiple relations between them. Moreover, the logic-based representation language enables the ability to reason, but at the same time, engineers have to ensure that the logical consequences of the knowledge modeled are foreseen to make sure the entailed knowledge is correct [145].

For these reasons, many ontology development methodologies provide tools and processes that help to control and supervise ontology development. Among them, a popular approach is to use so-called competency questions (CQs) [175] – a set of questions stated in natural language that a complete ontology should be able to answer correctly. A sample CQ defined for the software ontology may be Which software can export data to CSV files?. As CQs define the scope of knowledge a given ontology should represent, domain experts use them as requirements, and state them upfront before starting the process of ontology construction.

With a comprehensive set of CQs, we can achieve two goals:

- Define the vocabulary to be modeled – The ontology has to represent all concepts and relations mentioned in each CQ to be able to answer them. For example, to answer the following question Which software can export data to CSV files?, the ontology must include the notion of `software`, `CSV files` and know what it means to `export`. Using CQs, we can track how much of the required vocabulary is already modeled and what is still to be added. Some of the ontology development methodologies expect engineers to build a glossary of terms that list all vocabulary to be modeled in the ontology.
- Control the completeness and correctness of the ontology – With a predefined list of CQs, during ontology development, engineers can formalize each CQ in an appropriate query language (e.g., SPARQL-OWL [89]) to check if the ontology can provide the expected answers that may be listed together with the CQs. If the ontology provides a right answer to each of the CQs, one can assume it is complete (as it models the required vocabulary) and correct (as the knowledge and its logical consequences lead to correct answers).

The second goal is similar to the use of unit tests in software engineering. There, a set of unit tests is provided upfront, and during software development, those tests are run periodically to control the quality of the software. With a comprehensive set of tests, engineers may assume that the software is correct and complete if all tests pass.

Nowadays, competency questions are handled manually. It is especially problematic in the context of the second goal since:

- Engineers, apart from the ontology modeling language and the modeled domain, have to be proficient in the semantics of the query language.
- Engineers have to find classes and properties related to phrases stated in the CQ to retrieve their unique identifiers called IRIs that are used to construct queries. This task is time-consuming, especially in the context of large ontologies.

In addition, considering the glossary of terms construction mentioned in the first goal, it may not be interesting for engineers to iterate over CQs and extract domain-related phrases manually.

We expect that automated CQs handling will make ontology development easier and quicker. We hope that with appropriate methods helping to use CQs, more engineers will use them in their work.

1.3 Aim and scope of the dissertation

In this dissertation, we focus on the automatization of competency question handling in the context of ontology development. In particular, we would like to understand:

- How do engineers construct CQs and how phrases in CQs relate to labels of entities modeled in the ontology,
- How grammatical constructs used in CQs relate to forms of SPARQL-OWL queries,
- If it is possible to extract candidates to be included in the glossary of terms automatically, based only on the forms of CQs,
- If CQs and their SPARQL-OWL formalizations can be generated automatically to build synthetic datasets of translation examples,
- If it is possible to translate CQs into SPARQL-OWL queries automatically,
- If automatic translators of CQs into SPARQL-OWL queries can be integrated with Test-Driven Development of ontologies.

Based on the list presented above, we define the following research questions to be addressed in this dissertation:

- **RQ1** – Are there recurring patterns among CQs, SPARQL-OWL queries, and between CQs and SPARQL-OWL queries?
- **RQ2** – How to automate the glossary of terms extraction?
- **RQ3** – How to construct pairs of CQs and SPARQL-OWL queries automatically based on ontology axioms?
- **RQ4** – How to construct SPARQL-OWL query recommendations from CQs automatically?
- **RQ5** – How to integrate the automatic translation of CQs into SPARQL-OWL with Test-Driven Development of ontologies?

The main contributions provided in this dissertation are:

1. The dataset of CQs translated into SPARQL-OWL queries and its analysis introduced in Chapter 6. This contribution relates to **RQ1** and is publicly available on GitHub[1].
2. Two automatic glossary of terms extractors introduced in Chapter 7. This contribution relates to **RQ2**. Both extractors are available online[2,3].
3. The automatic method for recommending translations of ontology competency questions into SPARQL-OWL described in Chapter 9. This contribution relates to **RQ4** and its implementation is publicly available on GitHub[4].
4. The method of generating synthetic pairs of CQs and SPARQL-OWL queries described in Chapter 8. This contribution relates to **RQ3**. We made the implementation of the method and a dataset created with it publicly available on Github[5].

[1]`https://github.com/CQ2SPARQLOWL/Dataset`
[2]`https://github.com/dwisniewski/CRFBasedGlossaryOfTermsExtraction`
[3]`https://github.com/reqtagger/ReqTagger`
[4]`https://github.com/dwisniewski/SeeQuery`
[5]`https://github.com/dwisniewski/BigCQ`

5. The integration of the automatic translator into Test-Driven Development (TDD) of ontologies introduced in Chapter 10. This contribution relates to **RQ5**.

We listed the answers to all research questions in Chapter 11.

1.4 Organization of the dissertation

The dissertation is structured as follows: In Chapters 2, 3, and 4, we introduce some basic concepts related to Semantic Web, machine learning, and Natural Language Processing, respectively. Chapter 2 presents the need to choose SPARQL-OWL as a formalization of competency questions. In Chapter 5, we discuss related works. In Chapter 6, we introduce and analyze the dataset of real-world CQs translated into SPARQL-OWL. Chapter 7 discusses the automatization of the glossary of terms extraction from textual requirements. In Chapter 8, we introduce a large synthetic dataset of CQs formalized as SPARQL-OWL. We introduce SeeQuery, an automatic recommender of SPARQL-OWL queries from CQs in Chapter 9 and show how it can be integrated with test-driven development of ontologies in Chapter 10. Finally, we summarize the dissertation and discuss the future work in Chapter 11.

1.5 Author's publications

In this section, we list all the author's publications. The following list of symbols is used to describe papers:

- (J) or (C) – indicates whether a paper was published in a Journal or Conference, respectively.
- GS – citation number according to Google Scholar, visited on the 16th of December 2021.
- SCOPUS – citation number according to Scopus, visited on the 16th of December 2021.
- MEiN – the number of points assigned by the Polish Ministry of Education and Science as of 2021.
- CORE – The conference rank assigned by the Computing Research and Education Association of Australasia.
- IF5 – the 5-year impact factor calculated according to Journal Citations Report (JCR).

1.5.1 Papers related to the dissertation

[P1] *Automatic translation of competency questions into SPARQL-OWL queries*, Dawid Wiśniewski, In: Companion of the The Web Conference 2018 on **The Web Conference 2018, WWW 2018**, Lyon, France, April 23-27, 2018, pages 855–859. ACM, 2018.
(C), CORE: **A***, GS: **4**, SCOPUS: **2**, MEiN: **conference: 200, publication: unknown**

[P2] *Analysis of Ontology Competency Questions and their formalizations in SPARQL-OWL*, Dawid Wiśniewski, Jędrzej Potoniec, Agnieszka Ławrynowicz, C. Maria Keet, **Journal of Web Semantics** 59, 100534, 2019
(J), IF5: **3.524**, GS: **23**, SCOPUS: **16**, MEiN: **100**

[P3] *Dataset of ontology competency questions to SPARQL-OWL queries translations*, Jędrzej Potoniec, Dawid Wiśniewski, Agnieszka Ławrynowicz, C. Maria Keet, **Data in Brief** 29, 105098, 2020
(J), IF5: **-**, GS: **11**, SCOPUS: **7**, MEiN: **40**

[P4] *A tagger for glossary of terms extraction from ontology competency questions*, Dawid Wiśniewski, Agnieszka Ławrynowicz, **The Semantic Web: ESWC 2019 Satellite Events - ESWC 2019** Satellite Events, Portoroz, Slovenia, June 2-6, 2019, Revised Selected Papers, volume 11762 of LNCS, pages 181–185. Springer, 2019
(C), CORE: **A**, GS: **5**, SCOPUS: **4**, MEiN: **conference: 140, publication: unknown**

[P5] *ReqTagger: A rule-based tagger for automatic glossary of terms extraction from ontology requirements*, Dawid Wiśniewski, Jędrzej Potoniec, Agnieszka Ławrynowicz
(J) Accepted for publication in **Foundations of Computing and Decision Sciences**, IF5: **-**, MEiN: **40**

[P6] *SeeQuery: An Automatic Method for Recommending Translations of Ontology CQs into SPARQL-OWL*, Dawid Wiśniewski, Jędrzej Potoniec, Agnieszka Ławrynowicz, In Proceedings of the 30th **ACM International Conference on Information & Knowledge Management, CIKM '21**, pages 2119–2128, New York, NY, USA, 2021. Association for Computing Machinery
(C), CORE: **A**, GS: **-**, SCOPUS: **-**, MEiN: **140**

[P7] *Incorporating Presuppositions of Competency Questions into Test-Driven Development of Ontologies*, Jędrzej Potoniec, Dawid Wiśniewski, Agnieszka Ławrynowicz, **SEKE 2021**: Proceedings of the 33rd **International Conference on Software Engineering and Knowledge Engineering**, 437-440, 2021
(C), CORE: **B**, GS: **-**, SCOPUS: **-**, MEiN: **70**

[P8] *BigCQ: Generating a Synthetic Set of Competency Questions Formalized into SPARQL-OWL*, Dawid Wiśniewski, Jędrzej Potoniec, Agnieszka Ławrynowicz, **AAAI 2022**, accepted for publication
(C), CORE: **A***, GS: **-**, SCOPUS: **-**, MEiN: **conference: 200, publication: possibly 200**

The relation between these publications and the chapters introduced in this dissertation is as follows:

- Section 2.6 introduces the ideas stated in [P1],
- Chapter 6 summarizes the content of [P2] and [P3],
- Chapter 7 provides methods presented in [P4] and [P5],
- Chapter 8 relates to the dataset presented in [P8],
- Chapter 9 relates to [P6],
- Chapter 10 introduces concepts defined in [P7].

1.5.2 Other papers coauthored by the author

Apart from the publications related to this dissertation, the author coauthored the following peer-reviewed papers:

[P9] *Contract Discovery: Dataset and a Few-shot Semantic Retrieval Challenge with Competitive Baselines*, Łukasz Borchmann, Dawid Wisniewski, Andrzej Gretkowski, Izabela Kosmala, Dawid Jurkiewicz, Łukasz Szałkiewicz, Gabriela Pałka, Karol Kaczmarek, Agnieszka Kaliska, Filip Graliński, **Findings of the Association for Computational Linguistics: EMNLP 2020**, Online Event, 16-20 November 2020, 4254-4268.
(C), CORE: **A**, GS: **3**, SCOPUS: **-**, MEiN: **140**

[P10] *RecipeNLG: A Cooking Recipes Dataset for Semi-Structured Text Generation*, Michał Bień, Michał Gilski, Martyna Maciejewska, Wojciech Taisner, Dawid Wiśniewski, Agnieszka Ławrynowicz, Proceedings of the 13th **International Conference on Natural Language Generation, INLG 2020**, Dublin, Ireland, December 15-18, 2020, 22-28.
(C), CORE: **B**, GS: **3**, SCOPUS: **-**, MEiN: **70**

The papers with unknown number of MEiN points are published in companion volumes, the role of which is not specified by the MEiN yet.

- Publication [P1] was presented during the Ph.D. symposium organized during the Web Conference and published in the companion volume of the conference proceedings.
- Publication [P4] was presented as a poster. It was published in a *Satellite Events* proceedings volume.
- Publication [P8] will be published as a poster/student abstract at a conference early 2022.

1.6 Research grants participation

This dissertation is the result of research carried out in the following grants in which the author has participated:

- Grant No 2014/13/D/ST6/02076 (ARISTOTELES: Metodologia i algorytmy automatycznej aktualizacji ontologii w scenariuszach zadaniowych) funded by the Polish National Science Center. Principal Investigator: Agnieszka Ławrynowicz, Ph.D. Dr. Habil.
- Applied Research Programme under the EEA and Norway Grants 2014-2021. Project registration number: NOR/SGS/TAISTI/0323/2020 (TAISTI: Development of a Technology based on Artificial Intelligence for inferring SubsTitutable recipe Ingredients). Principal Investigator: Agnieszka Ławrynowicz, Ph.D. Dr. Habil.

Chapter 2

Selected aspects of Semantic Web

As the size of the World Wide Web keeps increasing rapidly, it is a tempting idea to make machines able to understand and share the knowledge presented on websites. Sir Tim Berners-Lee, the inventor of the World Wide Web, was aware of this quest since the early days of that technology. He was the first person to coin the term Semantic Web [14] to refer to an extension of the Web, which focuses on the data itself rather than its presentation and the possibilities to process that data by computers. The ultimate goal of the Semantic Web is to automatically share knowledge between websites and make it possible to reason about data so that humans and computers can cooperate better [159]. For this reason, many technologies were created to help represent, integrate, share, and reason about data. The most important are: Resource Description Framework (RDF) [116], Resource Description Framework Schema (RDFS) [22], Web Ontology Language (OWL) [6], reasoners [110], and RDF stores [3] that can be queried with query languages such as SPARQL Protocol And RDF Query Language (SPARQL) [158].

2.1 RDF: Resource Description Framework

In this section, we introduce a graph-based model for representing resources. First, we describe its building blocks and provide a formal definition of an RDF graph. Then, we discuss the most popular RDF graph serialization options.

2.1.1 RDF graphs

Resource Description Framework (RDF) is a graph data model used to represent information about resources on the Web [116]. RDF models information in the form of triples, each consisting of a subject, a predicate, and an object, with the use of the following building blocks:

- IRIs – Internationalized Resource Identifiers (IRIs) [43] are sequences of characters that uniquely identify resources. An example of an IRI is `https://www.google.com/search`.
- Literal values – Literals are used to represent data values, such as strings or integers. These are pairs (or triples if there is an optional language tag introduced, as described below) representing a given value as a string (lexical form) and a datatype IRI providing an interpretation of the value (e.g., integer, float, string, date). An example literal `"1"^^<http://www.w3.org/2001/XMLSchema#int>` states that the value `1` represents an integer.

If a literal describes a value of the `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString` datatype, one can introduce an optional language tag to specify in which language is the string expressed.

- Blank nodes – A blank node represents the existence of a resource without assigning it a particular IRI. They are used, e.g., to introduce complex structures, such as addresses. Each address can be represented as an unnamed resource that is described by a set of attributes: the postal code, the city name, the street name, and the street number.

Let I_{RDF}, L_{RDF}, and B_{RDF} denote all IRIs, literals, and blank nodes in the RDF graph G_{RDF}, respectively. Then, we use $T_{RDF} = I_{RDF} \cup B_{RDF} \cup L_{RDF}$ to denote RDF terms.
The RDF graph G_{RDF} is defined as a set of triples: (subject, predicate, object) $\in (I_{RDF} \cup B_{RDF}) \times I_{RDF} \times T_{RDF}$. Hereafter, we refer to the set of IRIs introduced in G_{RDF} as the vocabulary of G_{RDF}.

In Figure 2.1, we present a simple RDF graph consisting of three triples. Subjects and objects of these triples are represented with ovals, while predicates with directed arcs, pointing from subjects to objects. The grey oval represents a blank node and is used to tell that Mark works at some unnamed startup that is located in Poznan.

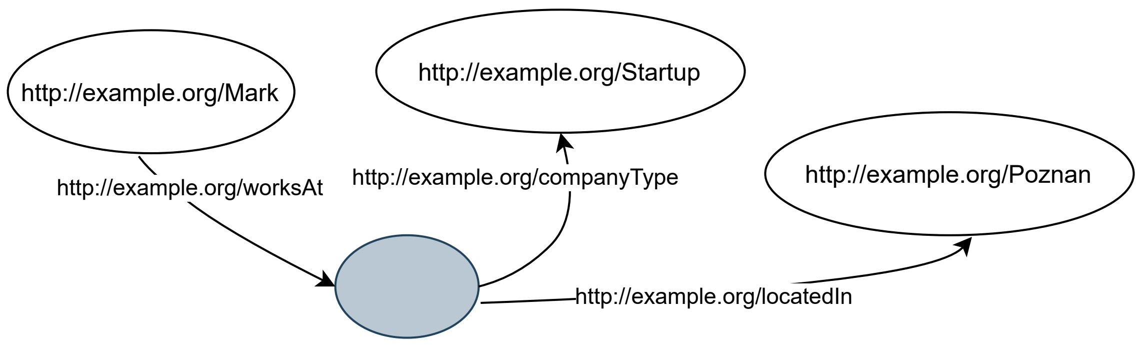

http://example.org/Mark
http://example.org/Startup
http://example.org/Poznan
http://example.org/worksAt
http://example.org/companyType
http://example.org/locatedIn

FIGURE 2.1: An RDF graph representation of the sentence: *Mark works at a startup that is located in Poznan.*

2.1.2 RDF serialization formats

RDF graphs are abstract objects that have to be serialized to be processed by machines or humans. There are several serialization formats, out of which the following are the most popular:

N-Triples N-Triples is a plaintext-based format where triples are serialized as whitespace-separated strings. Each triple ends with a dot (`.`) and is followed by a newline [157].

In N-Triples, we enclose IRIs with angle brackets. Serializations of blank nodes start with `_:`, followed by a blank node identifier. Literal values use the quotation mark (`"`) to wrap lexical forms. They may introduce an optional datatype IRI enclosed with angle brackets and prefixed with the `^^` sequence. If no datatype is specified, a language tag starting with the `@` character may be introduced after the lexical form. If there is no datatype and no language tag, the `http://www.w3.org/2001/XMLSchema#string` datatype is assumed. The N-Triples serialization of the RDF graph presented in Figure 2.1 can be defined as follows:

```
<http://example.org/Mark> <http://example.org/worksAt> _:company .
_:company <http://example.org/companyType> <http://example.org/Startup> .
_:company <http://example.org/locatedIn> <http://example.org/Poznan> .
```

RDF/XML RDF/XML is an XML-based serialization format [58]. Here, an XML document is used to store triples in an XML tree. It was the first serialization format for RDF and was recommended by World Wide Web Consortium (W3C). The RDF/XML serialization of the RDF graph presented in Figure 2.1 is defined as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:ns0="http://example.org/">

  <rdf:Description rdf:about="http://example.org/Mark">
    <ns0:worksAt>
      <rdf:Description>
        <ns0:companyType rdf:resource="http://example.org/Startup"/>
        <ns0:locatedIn rdf:resource="http://example.org/Poznan"/>
      </rdf:Description>
    </ns0:worksAt>

  </rdf:Description>
</rdf:RDF>
```

RDF/XML uses the `?xml` element to start the document.

As IRIs defined in a single ontology frequently share their beginnings, the notion of a namespace is introduced to shorten IRIs occurring in element and attribute names [93]. The `xmlns` attribute of the `rdf:RDF` element defines a list of namespaces that consists of pairs of namespace names and IRI beginnings separated by `=` signs. In the analyzed example, the namespace `ns0` shortens property IRIs (e.g., `http://example.org/worksAt`) by using the namespace name followed by a colon instead of the full IRI beginning (e.g., `ns0:worksAt`). We refer to the abbreviated form as a qualified name, with a prefix preceding the colon and a local name following it.

Each graph node, visualized in Figure 2.1 as an oval, is represented in RDF/XML as an `rdf:Description` element that, in the case of non-blank nodes, contains an `rdf:about` attribute specifying the IRI assigned to the graph node.

Predicates are represented as nested elements of their subjects and can be identified by their qualified names. If the predicate object is not used as a subject of another triple, it can be defined in the predicate element using an `rdf:resource` attribute. Otherwise, the object is an `rdf:Description` element nested in the predicate element.

Turtle Turtle is a compact, human-readable textual format [26]. It is an extension of N-Triples so that the N-Triples serializations of RDF graphs are valid Turtle documents. However, Turtle provides a syntax to simplify N-Triples notation. The Turtle serialization of the RDF graph presented in Figure 2.1 is as follows:

```
@prefix ns0: <http://example.org/> .

ns0:Mark ns0:worksAt [
    ns0:companyType ns0:Startup ;
    ns0:locatedIn ns0:Poznan
  ] .
```

The Turtle representation of the analyzed RDF graph starts with a prefix definition that introduces a namespace named `ns0`. The prefixes in Turtle work in the same way as in RDF/XML format, allowing to shorten IRIs. The first triple relates `Mark` with a blank node via the `worksAt` predicate. Both `Mark` and `worksAt` are represented using their qualified names. The blank node that serves as an object in this triple is represented with square brackets. Inside the square brackets, the blank node works as an implicit subject of the first triple so that only the predicate `ns0:companyType` and the object `ns0:Startup` are explicitly stated. Then, the colon sign informs that the same subject should be reused in the subsequent triple. In consequence, the second triple uses the same blank node as a subject, and only the predicate `ns0:locatedIn` and the object `ns0:Poznan` are explicitly stated.

In this dissertation, we use Turtle as our syntax of choice to represent all RDF graphs.

2.2 RDFS: RDF Schema

RDF Schema is a semantic extension of RDF providing data-modeling vocabulary [22] that we can use to describe:

Classes and Taxonomies RDFS introduces the notion of a class that allows to group multiple resources. Classes can further form hierarchies called taxonomies.

In RDFS, we introduce a class with `rdfs:Class` and define instances of classes with `rdf:type`. The following pair of triples:

```
example:C rdf:type rdfs:Class .
example:i rdf:type example:C .
```

states that `example:C` is a class, an instance of which is `example:i`.

The hierarchy of classes can be introduced using the `rdfs:subClassOf` predicate. This predicate says that every instance of a class stated in the subject is also an instance of a class stated in the object. For example:

```
example:Human rdfs:subClassOf example:Mammal .
```

states that every human is a mammal.

Properties Properties are binary predicates that relate subjects to objects [22]. Similar to classes, they may be organized into hierarchies with the use of the `rdfs:subPropertyOf` predicate. Moreover, properties may introduce typing of their subjects and objects using `rdfs:domain` and `rdfs:range`, respectively.

Human-readable resource descriptions Resources are frequently identified with IRIs that are hard to be processed by humans (e.g., `http://xyz.io/1324` to refer to the class representing dogs). To address this problem, RDFS provides a vocabulary that can be used to annotate resources with additional context. With the use of `rdfs:label` and `rdfs:comment` one can introduce a human-readable label for the resource and additional comments (e.g., usage examples), respectively.

Ability to entail new facts With the vocabulary provided by RDFS, we can infer new facts from the knowledge modeled in an RDF graph. Let us consider the following graph:

```
example:Human rdf:type rdfs:Class .
example:Mammal rdf:type rdfs:Class .
example:Human rdfs:subClassOf example:Mammal .
example:Mark rdf:type example:Human .
```

Even if it is not explicitly stated, we can infer that `Mark` is a `Mammal` because `Mark` is an instance of the class `Human`, and `Human` is a subclass of `Mammal`. As a result, a new inferred fact:

```
example:Mark rdf:type example:Mammal .
```

can be added to the graph.

2.3 OWL: Web Ontology Language

The Web Ontology Language (OWL) is a language proposed to represent ontologies for the Semantic Web [130] that is recognized as a standard by W3C [182]. The OWL language was introduced in 2004 and extended in 2009 in its second version named OWL 2.

Ontologies in OWL can be represented as RDF graphs. Similar to RDFS, OWL introduces a vocabulary adding special meaning to graphs. RDFS is intended to define the structure of data in terms of hierarchies of classes and properties, while OWL describes the semantic relationships between entities. OWL allows using class expressions to build new, complex classes from existing classes and property expressions [182] (e.g., a class expression can introduce the notion of *a final-year student working as a programmer in at least two companies*).

The building blocks provided by OWL can be grouped into 3 categories [119]:

- Entities – classes, properties, and individuals that refer to objects and relationships in the domain of interest. Similar to RDFS, classes represent groups of individuals, and properties represent binary predicates.
- Axioms – statements asserted to be true in a given domain the ontology describes [119].
- Expressions – complex descriptions formed using entities.

The OWL vocabulary provides various constructs. The following list provides a subset of some popular constructs used to define class axioms:

- `owl:disjointWith` can be used to state that given classes cannot have common instances.
- `owl:equivalentClasses` can be used to state that given classes are equivalent to each other.

The following list describes a subset of some popular constructs used to define class expressions:

- `owl:unionOf` can be used to declare new classes as unions of existing ones.
- Cardinality restrictions are useful to restrict the number of things referenced (e.g., to tell that every Ph.D. student has at least one supervisor). The OWL language provides various forms of cardinality-related constructs: `owl:minCardinality`, `owl:maxCardinality`, `owl:cardinality`, `owl:minQualifiedCardinality`, etc.

- `owl:someValuesFrom` and `owl:allValuesFrom` that helps restrict the classes of property values. For example, `owl:someValuesFrom` can be used to state that each mammal eats (some) fruits and `owl:allValuesFrom` can be used to state that each mammal eats only fruits.

OWL, which serves as a next layer over RDF and RDFS reusing their functionalities, can be serialized with RDF serialization formats. On top of that, it can also be serialized using OWL/XML, an XML-based format tailored to OWL [120], as well as with alternative formats that simplify the human understanding of the ontology. Two of the most popular are:

- Manchester Syntax [75] – providing a concise representation friendly for non-logicians. For example, the following snippet:

  ```
  Class: PhDStudent
      EquivalentTo: Person and (writes some DoctoralDissertation)
  ```

  states that there is equivalence between the notion of `PhDStudent` and a `person` who `writes` a `DoctoralDissertation`.

- Functional-style Syntax – another human-friendly representation that closely follows the structural specification of OWL [119]. For example, the following snippet in Functional-style Syntax:

  ```
  EquivalentClasses(
      :PhDStudent
      ObjectIntersectionOf(
          :Person
          ObjectSomeValueFrom(:writes
                              :DoctoralDissertation)
          )
      )
  ```

  represents the same axiom as analyzed in the context of the Manchester Syntax.

The possibilities given by OWL are at the cost of complexity of reasoning and decidability [130]. For this reason, in the first version of OWL, 3 sublanguages that are tailored to specific needs are provided:

- OWL Full – provides all primitives defined in OWL and allows an ontology to augment the meaning of either OWL or RDF vocabulary. For example, in OWL Full, we can define a class as its own instance. However, because of its expressive power, OWL Full is undecidable.

- OWL DL – provides all primitives defined in OWL, but it introduces several restrictions (e.g., a class cannot be declared as an instance of another class). OWL DL is named after its correspondence with description logics [7]. Because of the restrictions introduced, all entailments of the ontology are guaranteed to be computed, and we are sure that the computations will finish in finite time [182].

- OWL Lite – supports limited primitives of OWL. For example, the cardinality restrictions are limited to values 0 and 1. This language suits users who need to compute a classification hierarchy. The OWL Lite is the easiest to handle and support by tools [182].

OWL DL is the syntactic extension of OWL Lite. OWL Full is the syntactic extension of OWL DL and OWL Lite [182].

OWL 2 removes OWL Lite and introduces further 3 sublanguages called profiles:

- OWL 2 EL – especially useful in the context of ontologies with large numbers of classes and properties. OWL 2 EL assesses ontology consistency, class expression satisfiability, class expression subsumption, and instance checking in polynomial time with respect to the ontology size [76].
- OWL 2 QL – especially useful in the context of datasets with large sets of individuals, where query answering is the most important task [76].
- OWL 2 RL – especially useful in the context of applications that may trade language expressivity for efficiency. Reasoning systems based on OWL 2 RL can be implemented using rule-based engines [76].

Ontologies in the context of the open-world assumption Closed-world assumption (CWA) is a way of thinking that assumes complete information provided in the system [146]. It is frequently employed in databases. Under CWA, we state what is possible, and everything that is not stated is regarded as impossible. Let us consider a database on the academic world presented in Table 2.1.

Table 2.1: An excerpt of some academic database.

Person	Can be a supervisor?
Research Assistant	No
Assistant Professor	Yes
Associate Professor	Yes
Full Professor	Yes

Imagine the Table 2.1 is asked with the following question: Can a student be a supervisor?, the system working under CWA returns **false** as there is no information on students in the table.

A different way of thinking is represented in the open-world assumption (OWA). Under OWA, the lack of knowledge in the system does not imply its falsehood, as OWA assumes incomplete knowledge [82]. In the context of our example from Table 2.1, an OWA-based system would answer the same question with **I do not know** as there is nothing on students in Table 2.1. In particular, there is nothing about the possibility of a student becoming a supervisor. There should be a statement *No student can be a supervisor* represented in the system to generate **false** – the same answer as under CWA.

Ontologies expressed in OWL use the open-world assumption so that in an empty ontology, we assume everything to be possible. By adding new axioms, we constrain the ontology by defining what is not possible.

Ontologies in the context of the unique name assumption The unique name assumption (UNA) states that two entities assigned with different identifiers represent different things [147]. For example, under UNA, we assume that `http://example.org/a` and `http://example.org/b` refer to different things because they use different IRIs.

However, in OWL ontologies, UNA does not hold. If an ontology states that Poland has exactly one capital city and we model both `http://example.org/Warsaw` and `http://example.org/test` as capitals of Poland, the logical consequence of these portions of information will be that both IRIs refer to the same thing.

In OWL, we can explicitly state that two entities are different with the use of `owl:differentFrom`. We can also use `owl:sameAs` to explicitly state that the given entities represent the same thing.

2.4 Ontology verbalization

Understanding axioms formalized using logic-based languages such as OWL may be a challenge for people that are not trained in logic. For this reason, the possibilities of expressing ontologies using natural language are analyzed. Kaljurand [80] proposed to use a language called Attempto Controlled English (ACE) to express the knowledge encoded in an ontology using a subset of English. He showed that there is a mapping between ACE and OWL DL [81].

ACE is a Controlled Natural Language (CNL) representing a constrained natural language that relates to first-order logic [55]. ACE supports, among others, expressions for conjunctions, negation, existential and universal quantification, and numerical quantifiers. The admissible sentence structures allowed by ACE [53] cover expressions such as noun phrase + verb + complements + adjuncts or there is/are + noun phrase. Sentences are built from content words (nouns, verbs, adjectives, and adverbs) and predefined function words (e.g., determiners, pronouns, negation words).

2.5 Querying RDF with SPARQL

SPARQL **P**rotocol **a**nd **R**DF **Q**uery **L**anguage (SPARQL) is a query language that can be used to retrieve and modify data stored in RDF graphs on the Web or in an RDF store [158].

Let a Basic Graph Pattern (BGP) denote a set of triple patterns. Each triple pattern can be defined as a triple: $\in (T_{RDF} \cup V_{RDF}) \times (I_{RDF} \cup V_{RDF}) \times (T_{RDF} \cup V_{RDF})$, where V_{RDF} represents the set of query variables.

BGPs are stated in SPARQL queries to construct a graph that is then matched in the queried graph. If a query introduces variables, they are treated as wild cards. If a match is found, the variables are bound to particular IRIs and may be presented to the user.
An example query:

```
PREFIX ex: <http://example.org>
SELECT ?x WHERE {
    ?x ex:prepares ex:dissertation .
    ?x ex:hasStatus ex:PhDStudent .
}
```

consists of:

- The namespace prefix – similar to `@prefix` in Turtle, prefixes defined using `PREFIX` are used to make IRIs in the queries shorter.
- Query form, either `SELECT`, `ASK`, `DESCRIBE` or `CONSTRUCT`. In this case `SELECT` is chosen and is followed by a single variable `?x`, the bindings of which are presented to the user.
- The BGP that in this example consists of two triples:

```
?x <http://example.org/prepares> <http://example.org/dissertation> .
?x <http://example.org/hasStatus> <http://example.org/PhdStudent> .
```

Here, in both triples, the predicates and the objects are represented with IRIs, and the subjects are represented with the `?x` variable.

Considering the analyzed query used in the context of the following graph:

```
@prefix ex: <http//example.org>

ex:DWisniewski ex:prepares ex:dissertation ;
               ex:hasStatus ex:PhdStudent .
ex:JKowalski ex:prepares ex:masterThesis ;
             ex:hasStatus ex:student .
```

, the BGP specified in the query matches once and binds the variable `?x` to the IRI of `ex:DWisniewski`.

Entailment regimes The graph matching procedure processing only explicitly stated knowledge may provide incomplete results as the implicit knowledge is not considered.

Let us consider the following graph:

```
ex:Cat rdf:type ex:Mammal .
ex:Dog rdf:type ex:Animal .
ex:Mammal rdfs:subClassOf ex:Animal .
ex:isEatenBy rdfs:range ex:Animal .
ex:Banana ex:isEatenBy ex:Monkey .
```

This graph is visualized in Figure 2.2. The explicitly stated terms are represented with solid arcs and ovals. The implicit knowledge that can be inferred from this representation is presented using dashed lines and ovals.

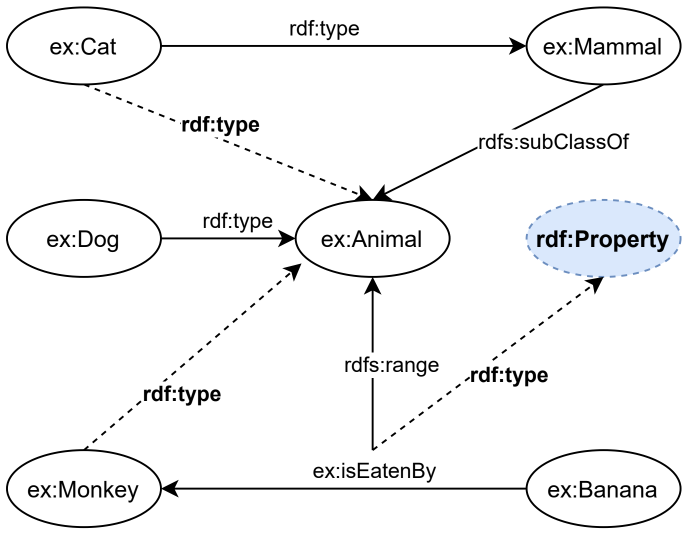


FIGURE 2.2: Visualization of a sample RDF graph. Solid ovals and lines represent explicit knowledge. Dashed ovals and lines represent inferred knowledge.

Then, let us consider the following SPARQL queries:

```
(Q1): SELECT ?x WHERE { ?x rdf:type rdf:Property }
(Q2): SELECT ?x WHERE { ?x rdf:type ex:Animal }
```

The first query searches for triples with `rdf:type` predicate and `rdf:Property` object and returns their subjects. The second query searches for triples with the same predicate but `ex:Animal` object and also returns their subjects.

In the analyzed graph, there is no explicitly introduced `rdf:Property`. For this reason, `Q1` cannot match and returns an empty result. Considering `Q2`, since there is the triple: `ex:Dog rdf:type ex:Animal` stated in the graph, `?x` binds to `ex:Dog` so that the `ex:Dog` is returned by the query.

However, these results are incomplete since, apart from the explicit knowledge, the graph also models indirect knowledge.

Entailment regimes are used to compute query results in SPARQL beyond simple subgraph matching. They use appropriate semantic interpretations of a given graph to entail new facts that can be processed by the query to make use of the indirect knowledge [61].

The following entailment regimes are possible in the context of SPARQL 1.1: RDF entailment, RDFS entailment, D-Entailment, OWL 2 RDF-Based Semantics entailment, OWL 2 Direct Semantics entailment, and RIF-Simple entailment [61]. The most popular ones are:

- RDF Entailment Regime – RDF provides only a little of semantic interpretation. However, we know that in each triple, the IRI stated in the predicate position represents a property. For this reason, from the following triple: `ex:Banana ex:isEatenBy ex:Monkey`, we can infer a new triple `ex:isEatenBy rdf:type rdf:Property`. Querying the graph using the RDF Entailment Regime returns `ex:isEatenBy` as the answer binded to `?x` in `Q1`.
- RDFS Entailment Regime – In this graph, there are two triples introducing RDFS-related vocabulary: `ex:Mammal rdfs:subClassOf ex:Animal` and `ex:isEatenBy rdfs:range ex:Animal`. With the use of the first one, we can conclude that because `ex:Cat` is a `ex:Mammal` it is also an `ex:Animal`. Moreover, as `esx:isEatenBy` expects an `ex:Animal` in its range, from `ex:Banana ex:isEatenBy ex:Monkey`, we can infer that `ex:Monkey` is an `ex:Animal`. The use of RDFS vocabulary to entail new facts for querying is called the RDFS Entailment Regime. Applying this kind of Entailment Regime, three resources can be bound to `?x` in `Q2`: `Dog`, `Cat`, and `Monkey`.
- OWL-related Entailment Regimes – Similar to RDF and RDFS, there are also entailment regimes related to OWL 2 that allow capturing the logical consequences of ontologies. There are two regimes to choose from:
  - OWL 2 Direct Semantics Entailment Regime – is based on OWL 2 Direct Semantics and maps the queried RDF graph as well as the BGP from the query to OWL structural objects. These can be extended to allow for variables [89]. OWL 2 Direct Semantics separates individuals, properties, and classes defined in the ontology and interprets classes and properties as sets and relations, respectively. Under OWL 2 Direct Semantics Entailment Regime variables can be used in place of names of individuals, literals, classes, object properties, and datatype properties [125]. The queried graph must correspond to an OWL 2 DL ontology.
  - OWL 2 RDF-based Semantics Entailment Regime – is a straightforward extension of the RDF and RDFS Entailment Regimes. It interprets a given RDF graph with the use of RDFS semantics with OWL 2 constructs handling. This kind of entailment regime assumes that queries are answered with respect to an OWL 2 RDF-Based datatype map [155]. This kind of semantics can reflect all logical conclusions of the OWL 2 Direct Semantics [155].

2.6 The need for SPARQL-OWL

SPARQL-OWL is SPARQL with OWL 2 Direct Semantics Entailment Regime [89]. It is a superset of SPARQL-DL [162] that is supported by a publicly available tool named OWL-BGP[1] and introduces strategies for good execution order and query rewriting to reduce the query execution time [89]. This language can be especially useful to query ontologies providing no or almost no individuals and large sets of terminological axioms, providing statements about how classes and properties relate to each other. An example of such an ontology is the Software Ontology [106], focused primarily on modeling software-related taxonomies and relations between classes.

The idea and example we describe here comes from the paper [185] presented during the Ph.D. symposium track of The Web Conference 2018. Let us consider the following scenario:

An engineer is working on a knowledge base (KB) – a type of knowledge representation that focuses mainly on individuals and uses an ontology as the schema. An excerpt of this knowledge base is presented in Table 2.2.

Table 2.2: A sample knowledge base represented using Functional-style Syntax.

(a): An assertional part.
`ex:Software(ex:Weka_(machine_learning))`
`ex:licence(ex:Weka_(machine_learning) ex:GNU_General_Public_licence))`
`ex:Software(ex:The_Witcher_(video_game))`
(b): A terminological part.
SubClassOf (`ex:Software`, `ex:EntityWithLicence`)
SubClassOf (`ex:EntityWithLicence`, ObjectSomeValuesFrom (`ex:licence`, `:Licence`))

Knowledge bases are split into two components [60]:

- Assertional part (ABox) that provides assertions on individual objects (e.g., individual `DawidWisniewski` is of type `PhDStudent`).
- Terminological part (TBox) that provides general assertions on classes and properties (e.g., classes `Programmer` and `SoftwareDeveloper` are equivalent, `Software` and `Hardware` are disjoint, or the class `Student` subsumes `PhDStudent`).

The KB presented in Table 2.2 consists of three assertions in the assertional part that can be interpreted in the following way:

- There is a piece of software called `Weka`.
- There is a piece of software called `The Witcher`.
- `Weka` is licensed under the `GNU` General Public licence.

Moreover, in the same table, there are two axioms providing terminological knowledge:

- The class `Software` is a subclass of `EntityWithLicence`.
- In general, `EntityWithLicence` has some licence assigned.

The engineer wants to use the knowledge base to answer the following CQ: Is it true that every piece of software has a licence?. They could state the following SPARQL query:

[1] `https://github.com/iliannakollia/owl-bgp`

```
ASK WHERE { ?x rdf:type ex:Software .
FILTER NOT EXISTS {?x ex:licence ?y} }
```

The query matches all pieces of software that are not related to any licence. If any match is found, the query returns **true**, what can be interpreted as *No, not every piece of software has a licence assigned.* This answer is returned by the query because in the assertional part of the KB, `The Witcher` is not related to any licence. In this query, the `NOT EXISTS` filter utilizes the closed-world assumption so that it expects complete information to be available in the KB.

It is easy to provide incomplete assertional knowledge that causes SPARQL queries to give wrong answers. To prevent such a behavior, one should use the general knowledge stated in the terminological part to form a query. The following SPARQL-OWL query:

```
ASK WHERE { ex:Software rdfs:subClassOf [
                rdf:type owl:Restriction ;
                owl:onProperty ex:licence ;
                owl:someValuesFrom ex:Licence ] . }
```

works on the schema of the KB (terminological part) and operates using the open-world assumption. As there are axioms relating `ex:EntityWithLicence` with `:Licence` via the existential restriction on `ex:licence` property and stating that each instance of `ex:Software` is an instance of `ex:EntityWithLicence`, the OWL 2 Direct Semantics Entailment Regime used by SPARQL-OWL make it possible to generate **true** as the answer to the query, which can be interpreted as *Yes, in general, each piece of software has a licence assigned.*

Chapter 3

Selected aspects of machine learning

3.1 The idea of machine learning

Machine learning (ML) is a term proposed by A. L. Samuel in 1959 to refer to a *field of study that gives computers the ability to learn without being explicitly programmed* [154]. This kind of approach makes computers learn from experience instead of providing the exact instructions to be performed. There are three main branches of ML:

- Supervised learning – a computer learns a mapping function from the input features into the output value [153] based on a set of examples. If the outputs to predict are continuous values, the solved problem is called regression. If the outputs are discrete labels, the problem is called classification.
- Unsupervised learning – a computer learns to detect patterns among data when no distinguished feature is provided (there is no output given as in the case of supervised learning).
- Reinforcement learning – a computer learns the strategy from reinforcements, which have the form of rewards or punishments [153]. For example, when the algorithm is playing chess, the final outcome – the information, whether the algorithm won or lost – is used to learn which movements contributed to the result.

3.2 Supervised learning

In the supervised scenario, having a training set $D = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$, where n represents the number of examples in D (i-th example is represented as a vector of features $\mathbf{x}^{(i)}$ annotated with a label $y^{(i)}$), our goal is to find a function: $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$, which discovers an underlying relation between features and labels so that each generated prediction $\hat{y}^{(i)}$ should be equal or close to the expected annotation $y^{(i)}$. The function $h(\cdot)$ is called hypothesis and it is expected to generalize so that given examples that were not seen in D, it can still generate accurate outputs.

Hereafter, we use the term predictions to refer to the outputs generated with machine learning models. The hypothesis $h(\cdot)$ generates predictions based on feature vectors which describe objects in terms of numbers and nominal values and a collection of parameters which determine the decisions. Parameters are selected in the process of model fitting (i.e., searching for the best model) that is based on examples in D. We use the term model to refer to the artifact created in the fitting process that provides predictions on the given data.

For example, considering a binary classification, where an object is described by a vector of features $\mathbf{x}^{(i)}$ and assigned with one of the two possible labels $y^{(i)} \in \{0, 1\}$, we can use logistic regression defined as

$$\hat{y}^{(i)} = h(\mathbf{x}^{(i)}) = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}^{(i)}}} \tag{3.1}$$

.

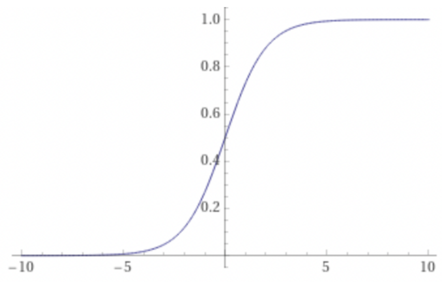


FIGURE 3.1: Sigmoid function.

Here, the dot product between the feature vector $\mathbf{x}^{(i)}$ and parameters vector $\mathbf{w}$ is transformed by the sigmoid function, which is visualized in Figure 3.1, to generate a single number with its value between 0 and 1. The calculated output can be interpreted as the probability of the object represented as $\mathbf{x}^{(i)}$ being of category $y = 1$. It can be further compared to a chosen threshold to transform probabilities into labels (e.g., if the probability ≥ 0.5 choose $y = 1$, else, choose $y = 0$). The model parameters and the threshold value define a decision boundary – a hyperplane that partitions the feature space into two regions, each related to one of the classes. To classify a given object, we check on which side of the decision boundary it falls. To fit the model (i.e., choose the best parameters $\mathbf{w}$ of $h(\cdot)$) to the training data, we introduce a loss function $\ell(y, \hat{y})$ measuring the difference between the model's prediction $\hat{y}$ and the expected output y. The value of the loss function depends on the model's parameters $\mathbf{w}$. The cost function $J(\mathbf{w})$ is an aggregation of the values of the loss function over multiple examples, e.g., the average value $J(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell(y^{(i)}, \hat{y}^{(i)})$, and is minimized using the chosen optimization algorithm to find the best parameters $\mathbf{w}$.

3.2.1 Optimization methods

In the context of supervised learning, the most popular optimization approach is gradient descent [121], which is an iterative method used to search for the local minimum of a given function. Gradient descent iteratively updates all parameters, repeating the transformation given as

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla J(\mathbf{w}^{(t)}) \tag{3.2}$$

, where $\nabla J(\mathbf{w}^{(t)}) = \frac{\partial J}{\partial \mathbf{w}^{(t)}}$ represents a gradient being a vector of partial derivatives, t represents a given point in time, and η represents the learning rate, which is a scalar determining how big should be the changes applied to the parameters.

Apart from gradient descent that utilizes first derivatives only, other optimization methods may use second-order derivatives providing information on the curvature of a given function to improve the convergence. For example, Newton's Method [121] uses the inverse of a Hessian $\mathbf{H}$, which represents the matrix of the second derivatives of a given function to define its curvature. The Hessian of a given cost function J and parameters $\mathbf{w}$ of length m is defined as: $\mathbf{H}_{i,j} = \frac{\partial^2 J}{\partial w_i, \partial w_j}$, for each $0 < i, j \leq m$.

Then, the Hessian is used in the optimization procedure in the following way:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \mathbf{H}^{-1} \nabla J(\mathbf{w}^{(t)}) \tag{3.3}$$

Not only the cost of calculating the Hessian matrix is large, but the matrix itself requires storing m^2 values, which may also become a limitation since many modern approaches optimize thousands or millions of parameters. For these reasons, attempts to reduce the complexity via an approximation of the inverse Hessian have been made. The most popular attempts are Broyden-Fletcher-Goldfarb-Shanno (BFGS) [23, 52, 63, 160], which reduces the time complexity of calculations and Limited-memory BFGS (L-BFGS) [99], which also approximates the Hessian matrix using a small number of vectors.

L-BFGS works iteratively to construct the approximated Hessian based on previous updates. Let:

- $\mathbf{y}^{(n)}$ be a vector of gradient changes $\mathbf{y}^{(n)} = \nabla J(\mathbf{w}^{(n+1)}) - \nabla J(\mathbf{w}^{(n)})$ between the current iteration $n+1$ and the previous one n.
- $\mathbf{s}^{(n)}$ be a vector of parameter changes changes $\mathbf{s}^{(n)} = \mathbf{w}^{(n+1)} - \mathbf{w}^{(n)}$ between the current iteration $n+1$ and the previous one n.
- $\rho_n = (\mathbf{y}^{(n)})^T \mathbf{s}^{(n)}$.

Then, the $\mathbf{H}^{-1}\nabla J(\mathbf{w}^{(t)})$ product in the update equation $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \mathbf{H}^{-1}\nabla J(\mathbf{w}^{(t)})$ can be approximated using the procedure presented in Algorithm 1.

```
Input: Gradient $\nabla J(\mathbf{w}^{(n+1)})$, the last $m$ vectors $\mathbf{s}^{(i)}$, $\mathbf{y}^{(i)}$, and $\rho_i$
Output: $\mathbf{s}^{(n+1)}$, which approximates $\mathbf{H}^{-1}\nabla J(\mathbf{w}^{(t)})$
1  $\mathbf{q}^{(m+1)} = \nabla J(\mathbf{w}^{(n+1)})$
2  for $i = m$ to $1$ do
3      j = i + n - m
4      $\alpha_i = \rho_j (\mathbf{s}^{(j)})^T \mathbf{q}^{(i+1)}$
5      $\mathbf{q}^{(i)} = \mathbf{q}^{(i+1)} - \alpha_i \mathbf{y}^{(j)}$
6  end
7  $\mathbf{z}^{(1)} = -\mathbf{h}_0^{-1} \mathbf{q}^{(1)}$
8  for $i = 1$ to $m$ do
9      $j = i + n - m$
10     $\beta_i = \rho_j (\mathbf{y}^{(j)})^T \mathbf{z}^{(i)}$
11     $\mathbf{z}^{(i+1)} = \mathbf{z}^{(i)} + \mathbf{s}^{(j)}(-\alpha_i - \beta_i)$
12 end
13 $\mathbf{s}^{(n+1)} = \mathbf{z}^{(m+1)}$
```

Algorithm 1: L-BFGS optimization procedure [123]. Here, $\mathbf{h}_0$ represents the initial Hessian approximation that is set to a scaled identity matrix.

3.2.2 Loss functions

Similar to optimization algorithms, there are many possible loss functions. Some of the most popular are:

- mean squared error (L2 error) – $\ell(y, \hat{y}) = (y - \hat{y})^2$, a loss function commonly used in regression problems.
- categorical cross-entropy – $\ell(y, \hat{y}) = -\sum_{i=1}^{K} y^{(i)} \log(\hat{y}^{(i)})$, where K represents the number of classes, measures the dissimilarity between two probability distributions. It is frequently applied to classification problems.

3.2.3 Regularization

Searching for the parameters that minimize the cost may lead to selecting a function that works perfectly on the data seen during the training but badly on unseen ones. The model suffers from *overfitting* when it overly fits the training data and misses the general trend the data represents. In Figure 3.2, we visualize a sample regression problem, where based on a set of 4 training examples (each marked with a cross shape $\times$), we search for a polynomial function that approximates the data. The function visualized on the left side fits perfectly the training data but considering an example that was not seen during training (marked with a rotated square), the prediction differs from the expected value. In contrast, the function presented on the right side, even if it gives less accurate predictions on the training data, generalizes better providing a prediction for a new example that is closer to the expected value. This problem, called *overfitting*, is caused by constructing a model that is too complex fitting to the noise that is frequently observed in datasets. To prevent *overfitting*, we aim to produce less complex models that may introduce fewer parameters (e.g., using a lower degree polynomial in the analyzed example) or that force the model parameters to be small.

To generate simpler models, we can use regularization methods that penalize a given model for being too complex. The most popular ones require adding a penalty score to the cost function that prevents the model from choosing high values of parameters:

- L1 regularization – in the context of regression problems also called *Lasso Regression* (Least Absolute Shrinkage and Selection Operator) [172], adds the $\lambda \sum_{i=1}^{m} |w_i|$ term to the cost function. The λ parameter value steers the importance of the regularization, while m represents the total number of parameters in the model. The larger the parameters $\mathbf{w}$ get, the larger $\sum_{i=1}^{m} |w_i|$ becomes. L1 regularization is useful in situations, where a subset of the most important features should be selected since L1 zeros weights related to the least important ones.

- L2 regularization – in the context of regression problems also called *Ridge Regression* [71], adds the $\lambda \sum_{i=1}^{m} w_i^2$ term to the cost function. It is analogous to the L1 regularization with the only difference in calculating the squares of parameters instead of absolute values. L2 regularization is useful when we would like to shrink all weights rather than zero their values.

- *Elastic Net* – both L1 and L2 are frequently combined into a weighted sum called *Elastic Net* regularization [200] $\lambda_1 \sum_{i=1}^{m} |w_i| + \lambda_2 \sum_{i=1}^{m} w_i^2$. That combination overcomes limitations of both approaches because *Ridge Regression* alone cannot eliminate irrelevant features, while *Lasso Regression* alone has problems with handling correlated features.

3.3 Neural networks

In the quest to make machines able to learn from experience, a lot of effort was made to mimic the behavior of the human brain. Since the work of McCulloch and Pitts from 1943 [113] introducing a mathematical model simulating a single neuron, a multitude of approaches and architectures incorporating neurons have been proposed to address machine learning tasks.

3.3.1 Neuron

The prediction process of a simple neuron consists of two phases – an aggregation of inputs followed by application of an activation function. In that sense, logistic regression is a kind of neuron as it

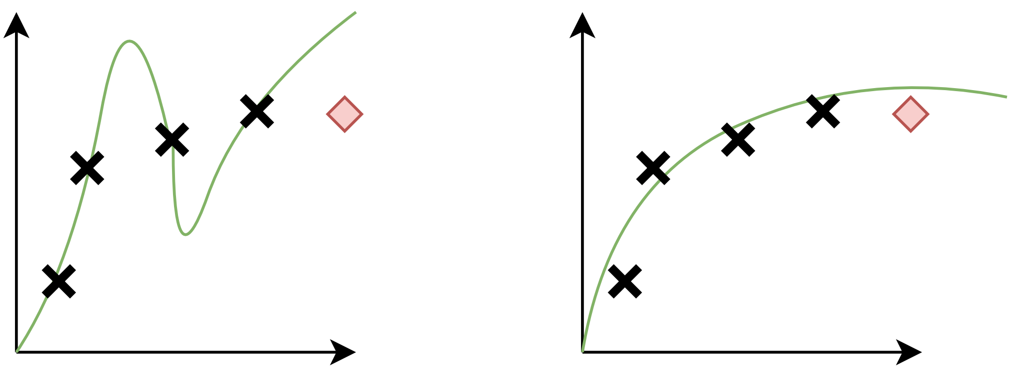


FIGURE 3.2: Two hypotheses that are fit to the same training data (represented with × shapes). The one presented on the left side overfits because it fits perfectly the training data but generates a large error for an unseen example, visualized using a rotated square. The function presented on the right side better captures the general trend in data. Even though it provides less accurate predictions on the training data, it generates a prediction closer to the expected value for a previously unseen example.

aggregates inputs and parameters (also called weights) first and then transforms the aggregated value using a nonlinear sigmoid function. A visualization of a neuron is presented in Figure 3.3.

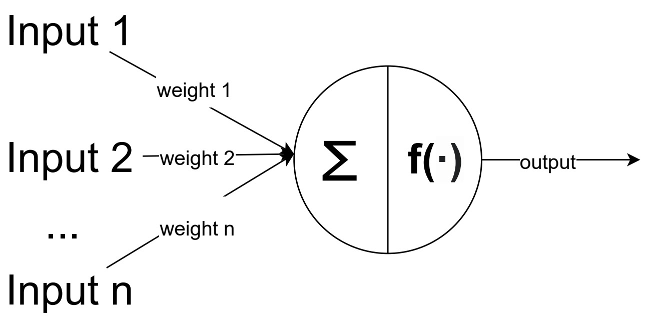


FIGURE 3.3: A neuron model with n inputs assigned with a weight each. The dot product between the weights and inputs processed by an activation function $f(\cdot)$ is the output.

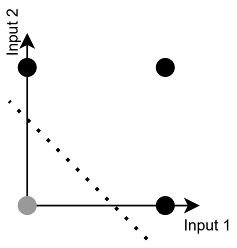
3.3.2 Feedforward neural networks

Early in the history of AI, scientists showed that a single neuron cannot solve problems in which nonlinear decision boundaries are required to separate classes. Pappert and Minsky [117] pointed that the XOR gate cannot be approximated by a single neuron as it is impossible to draw a single line – which represents the decision boundary when objects are described by pairs of features – that can separate XOR outputs set to 1 from those set to 0. In Figure 3.4, we analyze two logic gates: OR and XOR. We present their truth tables that describe pairs of inputs (features) mapped to outputs the gates return. We visualize the truth tables as graphs, where the horizontal and vertical axes represent the first input value and the second input value, respectively. We use black circles to visualize the outputs set to 1 and grey circles to visualize outputs set to zero. Only in the case of OR, it is possible to partition the space with a line so that all black circles are on the one side of the line (i.e., the decision boundary) and the grey circles are on the other.

However, if we decompose the XOR problem by transforming input features into a new feature space, the problem may become linearly separable in this new feature space. In Figure 3.5, we show that we can predict XOR outputs correctly if we introduce additional neurons and teach them to predict the AND and OR gates outputs. The outputs of these neurons generate new features that define each object using a pair of numbers representing the result of applying AND and OR gates over its input. In that, transformed feature space, the problem of predicting XOR output becomes linearly separable.

OR

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1



XOR

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

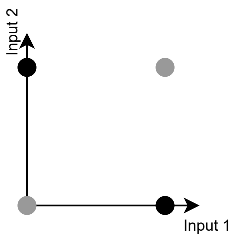


FIGURE 3.4: OR and XOR gates represented as their truth tables and visualizations in their feature space. Black circles represent coordinates assigned with the outputs set to 1. Grey circles represent coordinates assigned with the outputs set to 0. As can be seen, it is possible to separate black and grey circles with a line (represented as a dotted line) in the case of OR gate, but it is not possible for XOR.

A popular architecture using multiple neurons is a feedforward neural network. This approach organizes neurons into a selected number of layers, where each layer consists of a chosen number of neurons (different layers may contain different numbers of neurons).

All inputs of the feedforward network are processed by each neuron of the first layer independently, transforming them into a new representation using aggregation and transformation with a nonlinear activation function. The outputs of all neurons from the first layer become inputs of the second layer and the process of aggregation, transformation, and passing to the next layer is repeated until the last (output) layer of the network is reached. The output of the last layer represents the model's output.

Formally, a feedforward neural network can be defined as a sequence of transformations, where j-th transformation consumes the output of the previous layer (or the network input in the case of the first layer) and applies a nonlinear activation function g_j (g_j can be different for different layers):

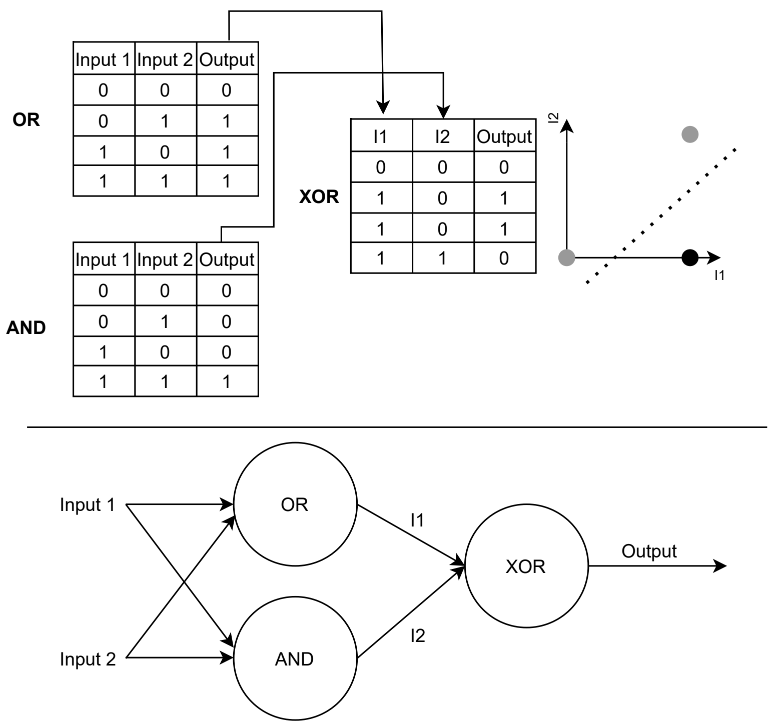
$$\mathbf{z}^{(j)} = g_j(\mathbf{W}^{(j)} \cdot \mathbf{z}^{(j-1)}) \tag{3.4}$$

$\mathbf{W}^{(j)}$ represents the weights related to connections between $j-1$-th and j-th layer. $\mathbf{z}^{(0)}$ is the feature vector describing the currently processed object represented as the feature vector $\mathbf{x}^{(i)}$.

To train a network involving multiple layers, we use the backpropagation algorithm [150].

3.3.3 Recurrent neural networks

In the late 1980s, the concept of recurrent neural networks (RRNs) was proposed to handle sequential data [150]. A simple RNN cell contains a special layer of neurons called hidden state that



OR

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

AND

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

XOR

I1	I2	Output
0	0	0
1	0	1
1	0	1
1	1	0

FIGURE 3.5: AND and OR gates are linearly separable problems. If the input values are processed by AND and OR gates and their outputs are fed to XOR gate, the XOR problem also becomes linearly separable in this new feature space. If we introduce two neurons, solving both AND and OR, and use their outputs as inputs of an additional neuron, we can train a network that predicts XOR labels correctly.

models the representation of the whole sequence processed by the cell so far. The hidden state is then used to compute outputs.

Assuming that $\mathbf{U}$, $\mathbf{W}$, and $\mathbf{V}$ are parameter matrices that are optimized during the learning, an RNN cell can be defined as a sequence of two transformations:

1. calculating the hidden state based on its previous state and the current input $\mathbf{h}^{(t)} = g_1(\mathbf{U} \cdot \mathbf{x}^{(t)} + \mathbf{W} \cdot \mathbf{h^{(t-1)}})$,
2. calculating the output $\mathbf{o}^{(t)} = g_2(\mathbf{V} \cdot \mathbf{h}^{(t)})$.

Here, $\mathbf{U}$ represents the matrix of weights joining the input and the hidden layer, $\mathbf{W}$ represents the matrix of weights joining the previous and the current hidden layer state, and $\mathbf{V}$ represents the matrix of weights joining the hidden layer and the output layer. $\mathbf{h}^{(t)}$ represents the hidden state at the timestep t, and $\mathbf{x}^{(t)}$ t-th input. $g_1(\cdot)$ and $g_2(\cdot)$ represent nonlinear activation functions (e.g., a hyperbolic tangent given as: $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Hyperbolic tangent squashes a given argument z to a number y, where $-1 \leq y \leq 1$). An RNN cell with 4 input values, the hidden state of size 2 and the output of size 2 used to process a sequence of 3 input values is presented in Figure 3.6.

As vanilla RNNs are hard to teach because they suffer from the vanishing gradient problem, making them unable to handle long sequences [69], extensions of RNNs introducing more complex cells were proposed. Long Short-Term Memory (LSTM) [70] and Gated Recurrent Unit (GRU) [28] are the most popular choices.

Recurrent neural networks are often used to transform one sequence of elements into another, possibly of a different length (e.g., in machine translation, summarization, question answering). Here, one RNN cell, called the encoder, embeds the representation of a whole sequence into a vector

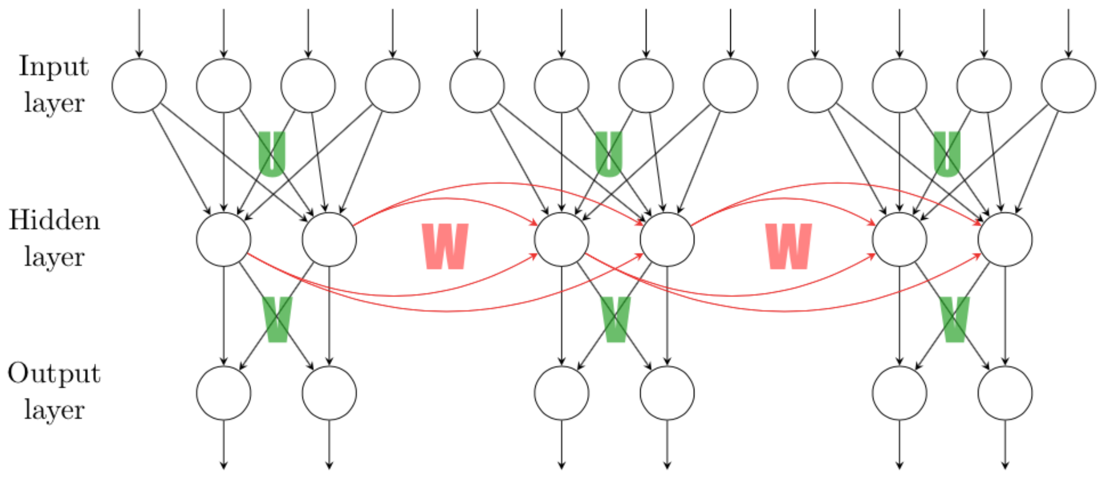


FIGURE 3.6: An example RNN cell unfolded to 3 time steps. Red arrows represent recurrent connections between the previous and the current hidden state. The input layer does not introduce any activation function – the purpose of this layer is only to broadcast each input to each neuron in the hidden layer.

stored in the last hidden state, and another RNN cell, called the decoder, produces a sequence of outputs based on that state. We call such kind of processing as a sequence-to-sequence (Seq2Seq) approach [170].

3.3.4 Attention-based neural networks

In recent years, the idea of attention [8] was introduced to Seq2Seq models. It allows the the decoder to access all hidden states (instead of the last one only) generated by the encoder after it processed a given input sequence. The encoder's hidden states are then weighted according to their importance and summed so that the decoder obtains more information from the relevant inputs.

The notion of attention evolved, and further research proved that it could replace recurrent connections so that each sequence element can be processed in parallel [177]. In the same paper, introducing a model called Transformer, the authors proposed the notion of self-attention, a type of attention relating different positions in a single sequence. With self-attention, the representation of a given sequence element can be expressed as a mixture of representations of all elements in the sequence.

3.4 Structured prediction problems

Structured prediction problems [9] relate to a class of machine learning problems, where given a structured input, the task is to predict structured objects such as trees or sequences. Contrary to classical approaches, here, the outputs are interdependent. Such a scenario is frequent in the field of natural language processing, where there are numerous tasks, hereafter referred to as sequence tagging tasks, that require tagging each element of a given sequence with labels that are dependent on labels assigned to the surrounding elements (e.g., it is much more likely that the successor of an adjective is a noun rather than a verb). A popular approach to solve structured prediction problems is the algorithm named Conditional Random Fields (CRF) [91] – a type of discriminative undirected probabilistic graphical model.

Having a sequence of T objects described as feature vectors $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$ and labels assigned to each object $y^{(1)}, \ldots, y^{(T)}$, our goal is to find a mapping from features to labels that can utilize dependencies between labels. The most popular type of a CRF model is a linear-chain CRF, which at a given moment, uses information about the label assigned to the current sequence element and the previous one. Formally, a linear-chain CRF can be defined as:

$$p(\mathbf{y}|\mathbf{x}) = \frac{1}{Z(\mathbf{x})} \prod_{t=1}^{T} \exp\{\sum_{k=1}^{K} w_k f_k(y^{(t)}, y^{(t-1)}, \mathbf{x}^{(t)})\} \tag{3.5}$$

, where the w_k symbol represents a learnable weight associated with each of the K real-valued functions $f_k(y^{(t)}, y^{(t-1)}, \mathbf{x}^{(t)})$ called feature functions [171], which process:

- $\mathbf{x}^{(t)}$ – features describing t-th sequence element
- $y^{(t-1)}$ – label of the previous sequence element, $y^{(0)}$ represents a special label representing the beginning of the sequence.
- $y^{(t)}$ – label of the current sequence element

$Z(\mathbf{x})$ represents a normalization function:

$$Z(\mathbf{x}) = \sum_{\mathbf{y}' \in Y} \prod_{t=1}^{T} \exp\{\sum_{k=1}^{K} w_k f_k(y'^{(t)}, y'^{(t-1)}, \mathbf{x}^{(t)})\} \tag{3.6}$$

, where $\sum_{\mathbf{y}' \in Y}$ represents the sum over all possible label sequences of length T.

In recent years, approaches combining recurrent neural networks with CRFs became popular as they can outperform classical CRFs [78]. In these approaches, CRF is represented as an additional layer that can learn the constraints inferred from the training set, e.g., that each entity encoded in a sequence should start with a distinct tag.

3.5 Encoding labels in sequences

Frequently, one has to assign labels spanning over multiple elements in a sequence when solving structured prediction problems. Hereafter, we refer to such spans as entities (e.g., mentions of possibly multiword sequences, such as company names in blog posts split into words). In these scenarios, an annotation scheme is required to mark the spans in training sets so that ML models can handle and predict the occurrences of possibly multielement sequences of relevant information.

The most popular approach is to tag each sequence element (e.g., each word in the sentence) with IOB [141] tags in the following way.

- O (outside) – is used to tag all elements outside of the sequences that form the entities to be detected.
- B (beginning) – is used to tag the first elements of sequences that form entities to be detected.
- I (inside) – is used to tag all but the first elements of sequences that form entities to be detected.

Sometimes variations of the IOB tag set are used:

- IO – use only two tags: I (inside) and O (outside). The lack of a distinct B (beginning) tag makes this tag set unable to separate entities that are adjacent to each other.
- BIOES/BILOU – extensions of IOB introducing additional tags:

  – E (end) / L (last) – used to mark the last element of an entity to be detected
  – S (single) / U (unit) – used to mark entities consisting of only single elements.

  BIOES and BILOU are synonyms, as the interpretation of the added tags is the same in both tag sets.

If more than one entity type needs to be detected by a single model (e.g., company names, phone numbers and addresses), one can extend the selected tag set by introducing appropriate postfixes identifying categories. A popular convention is to follow IO/IOB/BIOES/BILOU tags with a dash and a short identifier of the category (e.g., B-COMPANY represents the first element (word) in a company name and I-ADDRESS represents the continuation of a given address). As the tag O is not related to any category, we do not add postfixes to it. An example of a sentence tagged with IOB is presented in Figure 3.7.

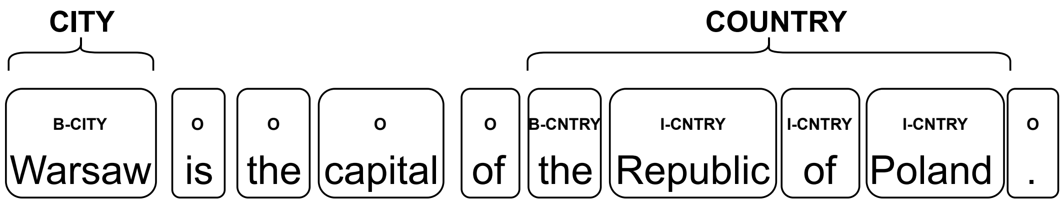


FIGURE 3.7: A sample sentence tagged with IOB tags. A single-token CITY and a multi-token COUNTRY are marked.

3.6 Evaluating machine learning models

Models trained on a given dataset have to be evaluated to check how well they work on unseen examples. To make the evaluation fair, the sets of examples used to train and evaluate the model should be disjoint, and both should follow the same distribution.

The following two approaches are the most popular ones to train and evaluate models taught in a supervised manner [17]:

- Hold-out – a given annotated dataset is split into two parts: a training set and a test set. Then, engineers train the model on the training set only and use the test set to check to what extent the predictions made by the model agree with the annotations. Frequently, another subset – named validation set – is also extracted from the training set. Engineers use the validation set to choose the hyper-parameters, which are parameters controlling the model's learning process. A popular scenario is to shuffle the dataset, use 60% of the examples as the training set, 20% as the validation set, and the remaining 20% as the test set.

- Cross-validation – Splitting the data using hold-out limits the size of the training set and leads to the arbitrary choice of the test and validation sets. To address these issues, one can use cross-validation. First, we split the dataset into k equal in size, disjoint subsets. Then, we iterate over each group, use the current one as the test set and the union of the remaining ones as a training set, train, and evaluate the model. As a result, there are k models produced tested against a different portion of the dataset each. Finally, we average the test scores. This way, the test set is not chosen arbitrarily as the whole dataset is used to calculate the evaluation scores. In each of the k steps, the procedure may be repeated to extract a validation set from the training set.

In supervised approaches, one can quantify the quality of a model on a dataset annotated with labels, that is disjoint with the training set, be it a validation or test set. Below, we discuss the most popular choices.

For regression, a popular choice is to calculate the mean squared error (MSE) [1]. Since the examples in the test and validation sets are annotated with numbers (e.g., stock prices to be predicted), the mean squared difference between the expected and generated values represents the quality of the model. Ideally, such a difference would be equal to 0. Formally, the MSE is defined as $\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$, where n is the number of examples, $y^{(i)}$ is the expected value of the i-th test example and $\hat{y}^{(i)}$ is the prediction on i-th example made by the model.

For classification and sequence tagging, the quality may be calculated per each label type, by counting the following cases:

- True positive (TP) – if the considered label is predicted by the model and is indeed expected for a given example or sequence element.
- False positive (FP) – if the considered label is predicted by the model, but is not expected for a given example or sequence element.
- False negative (FN) – if the considered label is expected for a given example or sequence element but is not predicted by the model.

Then, we can define the following aggregations per each label:

- Precision (P) – defined as the number of true positives divided by the sum of true positives and false positives $P = \frac{TP}{TP+FP}$. It shows what fraction of the model's decisions to assign a given class is correct.
- Recall (R) – defined as the number of true positives divided by the sum of true positives and false negatives $R = \frac{TP}{TP+FN}$. It informs what fraction of a given class occurrences in the test set is correctly identified.
- F_1 score – the harmonic average of precision and recall, given by $F_1 = \frac{2P \cdot R}{P+R}$. It aggregates precision and recall into a single number that can be used to rank different models.

The generalization of F_1 is F_β, given by $F_\beta = (1+\beta^2)\frac{P \cdot R}{\beta^2 P+R}$, which allows to give higher importance to precision or recall. For example, $\beta = 2$ weights recall higher, while $\beta = 0.5$ weights precision higher.

If the classes are evenly distributed, one can calculate the accuracy, given as the number of correct decisions made by a classifier divided by the number of all decisions. If the class distribution is skewed, we prefer to use other measures, among which there are F_1 or its generalization F_β.

The precision, recall, and F_1 can be averaged over all label types into a single number. Such aggregates are called macro-precision, macro-recall, and macro-F_1. Alternatively, one can calculate the total TP, FP, and FN summed over each label type and generate a single: precision, recall, and F_1. These measures are called micro-precision, micro-recall, and micro-F_1.

Chapter 4

Selected aspects of natural language processing

In this chapter, we would like to introduce some aspects of natural language processing (NLP), a subfield of AI, computer science and linguistics, focusing on how texts can be automatically understood, analyzed, generated and translated by machines. NLP tries to address a multitude of problems, among which there are:

- Automatic Speech Recognition – with the goal of providing textual transcription of spoken language.
- Machine Translation – focusing on providing the best quality translations of texts expressed in one language into another.
- Question Answering – with the goal of generating answers to questions stated in natural language automatically.
- Text Classification – with the goal of assigning a given piece of text one or more predefined labels.
- Named Entity Recognition – focusing on the identification and classification of important sequences of words.
- Language modelling – focusing on constructing models able to predict a next word as a continuation of a given sequence.

In this dissertation, we focus on concepts and approaches relevant to the dissertation only.

4.1 Tokenization

Each text itself is represented as a sequence of characters. Processing a piece of text as a sequence of consecutive characters is hard to automate for several reasons:

- Texts (especially longer sequences of characters) tend to be unique.
- Humans think in terms of concepts and relations between them, and these are linked to words, not the whole texts.

For these reasons, tokenization, the task of chopping texts up into pieces called tokens [107] is applied to split documents into a list of meaningful elements: words, punctuation, numbers, emoticons.

The most naive tokenization method is to split a given piece of text by spaces. Such an approach can frequently detect words correctly, as most words are surrounded by spaces, but at the same time it will fail in each case, where a word is followed by a punctuation instead of space (e.g., *it!*, *that?*).

However, adding more advanced rules, as those separating punctuation from sequences of letters, will not solve the problem completely. Sometimes, the decision whether a non-alphanumeric character should separate two tokens or be a part of a single token depends on the context. Imagine two examples with hyphens used (i) *hi-fi* (ii) *New York-based.*

In the first example, the whole *hi-fi* represents a device playing high-fidelity sound. Splitting it into separate tokens *hi* and *fi* leads to a loss of meaning. However, considering the second example of *New York-based*, we expect the tokenizer (the tool performing tokenization) to split the sequence by hyphen to produce *York* and *based* separately.

Modern solutions, such as those implemented in popular NLP toolkits like: spaCy [74] and NLTK [16] use predefined lists of rules to handle the most common problematic cases.

Tokenization is an important step when building various tools:

- In the information retrieval context – one can search if a given document contains a given token. When a document is tokenized, it is easier to reject matches being substrings of tokens.
- In the machine learning context – documents are frequently transformed into fixed-sized vectors of numbers. A popular approach is to use bag-of-words representation, the basic form of which can be built as follows:

  1. Tokenize all documents $d_i \in D_{DOCS}$.
  2. Create the vocabulary V – a list of unique tokens collected from all tokenized documents $d_i \in D_{DOCS}$. V is often sorted by the number of occurrences of a given token, so the most frequent ones are on the top of the list. Its main purpose is to map tokens to their positions in V.
  3. Use the vocabulary V of size n to transform the document d_i into a vector of numbers $\mathbf{d_i}$ of size n. First, initialize $\mathbf{d_i}$ with zeros. Then, iterate over each token t_j from d_i and increment $\mathbf{d_i}$ at the position that is assigned to t_j in V.

  Such a representation may be useful to discover relations between the number of occurrences of a given word and a category of a document (e.g., documents about sport may frequently contain tokens like: *goal*, *competition*, *match*).

Sometimes, it is helpful to consider sequences of consecutive tokens instead of each token separately. For example, the aforementioned bag-of-words representation in the context of machine learning would work better if pairs of consecutive words are also considered – having observed that the sequence of two words *New York* occurs in a given text is often a strong indicator that a given text is about the U.S.A. However, considering only single tokens would introduce ambiguity, since *New* is a frequent word that may occur in the majority of texts and *York* may be related to New York in the U.S.A, but also the city of York in the United Kingdom. We call a sequence of n consecutive tokens an n-gram.

Modern neural network-based approaches often provide pretrained models that can be used without resource- and time-consuming training [38, 100]. These often require a fixed vocabulary of tokens, mapping a given token into a number that is the same number as it was used during model training. The vocabulary, on the one hand, should be big enough to cover as many words as possible, but on the other hand, should be small enough not to consume too much memory. From these conflicting requirements, new methods of tokenization arise, splitting words into smaller pieces, called subword units.

The most popular methods providing subword units are WordPiece [156] and Byte Pair Encoding (BPE) [57]. Both start with an initial vocabulary of single characters, and then add new entries that are frequent sequences of characters.

As a result, even if a pretrained model with a fixed vocabulary is used against a piece of text with words that are not included in the vocabulary, the tokenizer splits such words into smaller portions (subword units) that are present in the vocabulary.

4.2 Part-of-speech tagging

Part-of-speech (POS) tagging is the process of assigning each token its appropriate part of speech information [108]. The tools performing POS-tagging – so-called POS-taggers – can use various sets of parts-of-speech to tag tokens. For example, the popular NLP toolkit spaCy [74] provides two tagsets to choose from:

- Universal POS-tags – a coarse-grained tagset used in Universal Dependencies project [35]. This project aims to provide a cross-linguistically consistent morphosyntactic annotation of human language. Universal POS-tags define 17 distinct part-of-speech tags, among which the `NOUN` tag represents a noun, `ADJ` an adjective, and `VERB` is used to tag verbs.

- OntoNotes (version 5) / Penn Treebank POS-tags – a fine-grained tagset coming from the OntoNotes project, the goal of which is to annotate a large corpus containing various genres of texts written in English, Arabic and Chinese with stuctural information [181]. This tag set defines 53 tags, providing more detailed information than those from Universal Dependencies. For example, OntoNotes POS-tags distinguish singular nouns (`NN`) from plural (`NNS`) and verbs in various grammatical forms, such as its base form (`VB`), past tense (`VBD`), gerund or present participle (`VBG`), past participle (`VBN`), 3rd person singular present (`VBZ`), and non-3rd person singular present (`VBP`).

Figure 4.1 presents a sample sentence tagged using Universal and OntoNotes tagsets.

The main challenge of POS-tagging comes from the ambiguity of words – many of them represent different parts of speech depending on their usage context. For example, the word *book*, if used to reference some printed work is a noun, but if it expresses the process of reserving accommodation, the same sequence of characters is used as a verb. However, even simple methods based on an analysis of annotated examples assigning a given token the most frequent tag lead to over 90% of per-token accuracy [27].

Today, POS-tagging is performed using machine learning-based approaches, out of which Conditional Random Fields (CRF)-based [91], bidirectional long short-term memory (LSTM) [19] or bidirectional encoders from Transformers (BERT) [68] are the most popular ones.

4.3 Dependency parsing

Dependency parsing is the name of an NLP task of discovering the syntactic structure of a sentence [90]. It analyzes the relations between words, linking them with binary asymmetric arcs and assigning each relation a label representing its functional category.

Each relation points from a head token to a dependent one, where:

	The	big	brown	fox	jumped	over	the	fence	.
ONTONOTES	**DT**	**JJ**	**JJ**	**NN**	**VBD**	**IN**	**DT**	**NN**	**.**
UNIVERSAL	**DET**	**ADJ**	**ADJ**	**NOUN**	**VERB**	**ADP**	**DET**	**NOUN**	**PUNCT**

FIGURE 4.1: Part-of-speech tags assigned to each token using Universal and OntoNotes tagsets. Here, `DT` and `DET` represent a determiner, `JJ` and `ADJ` represent an adjective, `NOUN` – a noun, `NN` – a noun, singular or mass, `VERB` – a verb, `VBD` – a verb in past tense "`.`" and `PUNCT` – a punctuation mark, `IN` – a preposition, and `ADP` – an adposition.

- A head governs the phrase and determines its syntactic category (e.g., if the head of a phrase is a noun, the whole phrase becomes a noun phrase) [201].
- A dependent token is modified by the head (e.g., it must agree with the head's number or gender).

Dependency parses are graphs that are [90]:

- Acyclic – The directed arcs do not form cycles.
- Connected – There is a path between every pair of nodes (tokens).
- Single-headed – There is only one incoming edge for every node (token) other than a distinguished ROOT node. The edge points from the head to the dependent.

Acyclic and connected graphs represent trees. For this reason, the result of dependency parsing is frequently called a dependency parse tree, which has a distinguished ROOT node that is the head of the entire sentence.

We call an arc that points from a head to its dependent projective if there exists a path pointing from the head to every token that lies between the head and the dependent in a given sentence. If all arcs in a dependency tree are projective, the tree itself is projective.

Frequently, an additional constraint is added on dependency parse trees to ensure their projectivity. It is caused by the fact that the annotated examples used to construct parsers, i.e., treebanks, contain only projective trees and many algorithms suffer from computational limitations that force them to build projective trees [79].

A popular source of the label set is the Universal Dependencies project [35], which provides, among others, labels representing the nominal subject (*nsubj*), object of a preposition (*pobj*), passive auxiliary (*auxpass*), coordination (*cc*), or direct object (*dobj*).

The most popular dependency parsing implementations are ML-based. Some of them are a second-order TreeCRF model [197], a self-attention-based joint syntax and semantics parser [199], or a BERT-based model [180].

Figure 4.2 visualizes a dependency parse tree generated for a sample sentence.

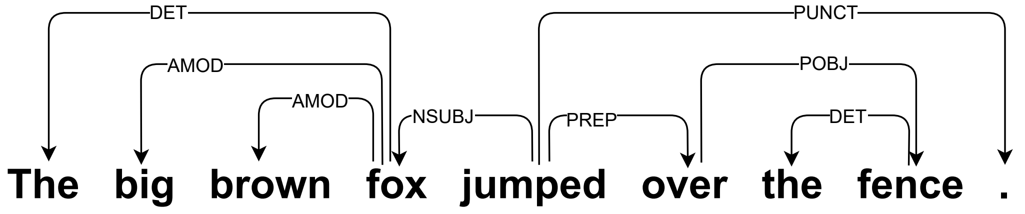


FIGURE 4.2: Dependency parse tree constructed for a sample sentence.

4.4 Regular expressions

Regular expressions are sequences of characters used to specify the pattern to be matched in a given piece of text. They originate from the work of Kleene on regular languages and finite automata [88]. Kleene proved the equivalence between regular expressions and finite state automata (FSA) – abstract models of computation that can be defined as a 5-tuple: $Q_{RE}, \Sigma_{RE}, \delta_{RE}, q^0_{RE}, F_{RE}$, where:

- Q_{RE} – represents the (finite) set of states.
- Σ_{RE} – represents the alphabet.
- δ_{RE} – represents the transition function $Q_{RE} \times \Sigma_{RE} \to Q_{RE}$ ($Q_{RE} \times \Sigma_{RE} \to P_{RE}(Q_{RE})$, where $P_{RE}(\cdot)$ is a powerset function in case of nondeterministic automata introduced further in this section).
- q^0_{RE} – represents the initial state $q^0_{RE} \in Q_{RE}$.
- F_{RE} – represents the accept states set $F_{RE} \subseteq Q_{RE}$.

A regular expression consists of literals and meta-characters. For example, the `abc+d?` regex matches a sequence consisting of letters *ab* followed by at least one occurrence of letter *c* with an optional letter *d* afterwards. In this example, letters `a`, `b`, `c` and `d` are literals matching exactly the letters they represent, while `+` and `?` are meta-characters, the first of which expecting the letter c to be repeated at least once and the second marking letter d as optional.

Regular expressions, when executed, are translated into one of two kinds of FSA so that the automata are used to match patterns in text. These two kinds of FSA are:

- Deterministic Finite Automata (DFA) – FSA that require reading input for the transition between states. Moreover, each transition in DFA is uniquely determined by an input and the source state.
- Nondeterministic Finite Automata (NFA) – FSA that do not require reading an input for the transition between states. It allows transitions to be nonuniquely determined by an input and the source state [140]. Reading an input sequence, it is not determined in which of the accepting states the processing will finish.

Figures 4.3 and 4.4 present the visualizations of DFA and NFA constructed from `abc+d?` regular expression.

Considering the above, regular expressions can be interpreted as a human-readable shortcut for defining finite state automata.

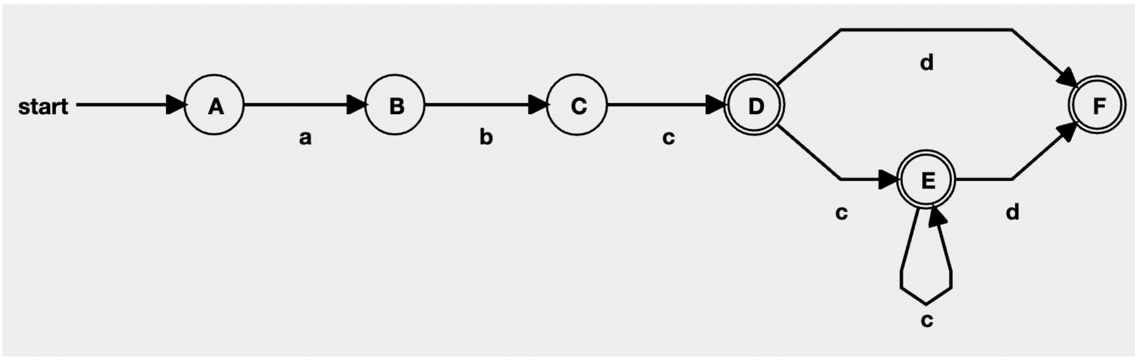


FIGURE 4.3: The abc+d? regex translated into DFA.

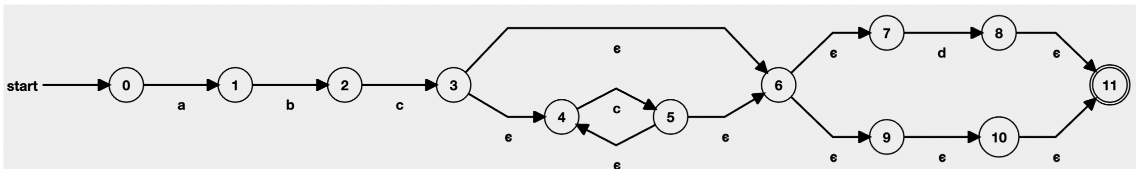


FIGURE 4.4: The abc+d? regex translated into NFA.

4.5 Distributional semantics and word embeddings

In 1950s, the research made by linguists Harris [66] and Firth [51] gave birth to so-called distributional semantics [21] – the field of study that defines the meaning of a word by the context they appear in. According to Firth *You shall know a word by the company it keeps* [51].

This idea was adopted later in NLP. For example, in 2003, Bengio et al. [11] proposed a neural network-based method of calculating distributed representations of words so that each word is mapped into a real-valued vector representing its meaning. The vectors representing the meaning of words are called word embeddings.

The major advantage of word embeddings is that they can be computed on a large dataset and then the generated representations can be reused for various tasks. The following list provides the most popular word embedding generation methods:

1. Word2Vec [115] – a model providing two neural architectures for cumputing embeddings. These are
   - Skipgram – in which the model learns to predict the words that surround a given word,
   - Continuous Bag-of-Words (CBOW) – in which the model learns to predict a given word based on words that surround it.
2. GloVe [131] – a model learning representations based on global co-occurences between words.
3. FastText [20] – a model based on CBOW that learns representations of character n-grams instead of full words. The vector representation of a given word is generated by taking the sum of vectors assigned to the n-grams the word contains.

However, all the listed methods map a given token into a fixed vector representation calculated on some chosen dataset. It is a major drawback, as there are words that are highly context-dependent (e.g., the word apple may refer to a fruit or a company, and all personal pronouns (I, she, it) cannot be represented as meaningful vectors without taking into consideration the contexts in which they were used). It would be beneficial to inject the context of the currently processed text to generate embeddings that are aware of the context a given word is currently used.

This problem was the inspiration for new methods for generating context-aware word embeddings. ELMo [133] uses a bidirectional LSTM architecture to construct a vector representation of each word. The bidirectional LSTM consists of two LSTM cells: one reading the text in the left-to-right order, while the other in the right-to-left. When the ELMo processes a word, the LSTM cells process their left and right contexts, and their hidden states are concatenated to produce a context-aware representation. Universal Language Model Fine-tuning (ULMFiT) [77] is a similar idea that incorporates special LSTM networks called AWD-LSTM [114] to generate contextual embeddings. However, the authors focus on the application of transfer learning to their solution so that the representation of the embeddings can be fine-tuned depending on the task to solve. Shortly after the publication of ELMo and ULMFiT models, a new breakthrough arrived with the BERT model [38], which is another neural model generating context-aware word embeddings. BERT allowed establishing new state-of-the-art scores for most NLP tasks. BERT utilizes the encoder as defined in the Transformer [177] model. The encoder is aware of both contexts at once since it uses self-attention to express the meaning of each token as the combination of the meanings of all the tokens in the processed sequence. In contrast, LSTM cells, even in bidirectional scenarios, use only left or right context at once so that the left-to-right processing cell is unaware of the context processed by the right-to-left processing cell. Moreover, BERT can be trained faster, as it does not use recurrent connections, which slow down the training process of LSTM cells. The BERT models are pretrained on two tasks:

- Masked language model (MLM) – representing a scenario in which some tokens in the text are replaced with placeholders. The model learns to predict the masked tokens.
- Next sentence prediction (NSP) – representing a scenario in which having a pair of sentences provided as an input, the model learns to decide whether the second sentence is the continuation of the first one.

and made publicly available.

The publication of BERT started a new era of Transformer-based models generating context-aware embeddings. A popular modification of BERT called RoBERTa [100] is its promising evolution trained on the MLM task only. It is trained using 10 times more data than the original BERT.

Each of the described methods allows for the calculation of the semantic similarity between tokens. Having two embeddings representing the meaning of two tokens: **a** and **b**, we can use the cosine similarity to calculate how similar the tokens are:

$$cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} \tag{4.1}$$

Frequently, a similarity between sequences of tokens needs to be calculated (e.g., phrases, sentences, documents). In that case, a popular heuristic is to calculate word embeddings for each sequence element and then average the embeddings to produce a single vector representing the whole sequence. The similarity between the sequences can be then calculated as the similarity between averaged vectors.

Chapter 5

Related work

5.1 Ontology engineering methodologies

In recent decades, multiple approaches focusing on structuring the ontology development process have been proposed.

The first methodology can be traced to the Cyc project – a common sense knowledge base constructed by Lenat et al. [96]. The approach proposed by Lenat et al. considers creating an ontology from scratch, applying the manual extraction of knowledge from books, articles, and journals as an initial step. However, as an early methodology, it focused on general recommendations on how to improve the quality of ontologies rather than an in-detail description of the steps to perform and tools to use.

A more formalized approach and one of the first well-known methodologies is the one proposed by Gruninger and Fox in the context of the TOVE ontology [65]. In this case, a set of motivating scenarios is created to define informal competency questions. These are then used to formalize the terminology and axioms that should be modeled using first-order logic and guide the evaluation of ontology quality by measuring how many CQs the ontology answers correctly.

Uschold and King [176] split the ontology development into a sequence of steps to be performed. These include identification of the ontology's purpose, ontology capturing focused on concepts and relations identification, ontology coding, evaluation, and documentation. The methodology proposes to integrate existing ontologies when possible and use CQs to measure the quality of ontologies.

Bernaras et al. [13] introduced the KACTUS approach. This methodology describes the idea of ontology reuse and integration. KACTUS lists several processes to be performed: specification of the application, preliminary design based on relevant top-level ontological categories, and ontology refinement and structuring.

METHONTOLOGY introduced by Lopez et al. [50] defines a set of activities that the development should follow. They are, namely: planning, specification, knowledge acquisition, conceptualization, formalization, integration, implementation, evaluation, documentation, and maintenance of the ontology. These steps are used in an evolving prototype scenario, allowing to come back to previous activities when building the ontology. Engineers can use CQs for specification purposes. These help to build the glossary of terms, one of the artifacts of the conceptualization activity.

Noy et al. proposed an interactive approach called Ontology Development 101 [124]. The authors suggest that it is impossible to present a single solution for ontology building, so they define a set of good practices instead. These include determining the domain and scope of the ontology, identifying resources to reuse, listing terms, building taxonomies, defining properties and property restrictions, and creating instances. Apart from the steps of the ontology engineering process, a discussion on important dilemmas is presented. Noy et al. try to answer common questions like: Should we represent our term as a class or an instance? What should be the scope of the ontology? Should we create a new class?

Moreover, a set of general good practices is introduced (e.g., mark classes as disjoint when possible, follow a coherent naming convention, avoid cycles in class hierarchies).

To determine the scope and domain of the ontology, the authors suggest preparing a list of CQs, out of which nouns are often good candidates for classes or instances, and verbs are good candidates for properties.

Suarez-Figueroa et al. [167] prepared the NeOn methodology consisting of 9 scenarios and providing life-cycle models, processes glossary, and methodological guidelines. Moreover, it describes how to create ontologies collaboratively. In NeOn, ontology development consists of the following steps: conceptualization, ontology requirements specification, ontology reuse, and ontology evaluation. CQs are used in NeOn as part of the ontology requirements specification.

On-To-Knowledge proposed by Sure et al. [169] focuses on ontology development processes and knowledge meta-processes. It starts with the feasibility study to identify problems and solutions regarding the domain. Then, the kick-off step focuses on the preparation of the ontology requirement specification document, providing, among others, the scope, CQs, usage scenarios, and defining potential users. An ontology is constructed in the refinement phase and evaluated in the evaluation phase. For evaluation, the ontology is used in the target application, and CQs are used to check the quality of the knowledge modeled in the ontology. The maintenance and evolution step is devoted to managing the organizational maintenance process and evolving the ontology. On-To-Knowledge allows repeating refinement evaluation processes to check whether the requirements defined in the kick-off phrase are fulfilled.

The test-driven development approach, well known in the domain of software engineering, was presented in the context of ontologies by Vrandecic et al. [178], who justified the need for unit testing in ontology development. Keet et al. [85]. proposed a test-driven development approach, which provides a set of 42 kinds of tests to evaluate the knowledge modeled in an ontology. Those tests verify both the TBox and ABox-level knowledge. This approach also uses the notion of CQs as a possible input passed to a test.

Blomqvist et al. [18] proposed another test-based methodology for ontology development. In that approach, tests are created from each of the requirements, and the requirements are formalized to verify the answers.

Peroni [132] proposed an agile ontology development methodology called SAMOD, which uses test cases, each consisting of a motivating scenario, a list of CQs related to the scenario, a glossary of terms addressed by the scenario, a formal model encoded in the appropriate language (e.g., OWL2), a set of examples described in the motivating scenario, and a list of queries formalized using a query language (e.g., SPARQL).

eXtreme Design (XD) is a test-driven ontology development methodology proposed by Suárez-Figueroa et al. [36] later revisited by Presutti et al. [137]. In this approach, a set of CQs is formalized as SPARQL queries, which serve as tests.

The problem of ontology testing was also analyzed by Fernández-Izquierdo et al. [48], who proposed Themis – a tool for validating ontologies through requirements. Themis provides a testing process, which is split into 3 phases: test design, test implementation, and test execution.

As can be seen, multiple attempts to provide ontology engineering methodologies are proposed. Many of them use the notion of competency questions to outline the requirements an ontology should meet. Moreover, some methodologies incorporate the idea of testing adopted from the software development field and use an appropriate formalization language (e.g., SPARQL) to verify the content of the ontology.

5.2 Collections of CQs and their analyses

Fernández Izquierdo et al. [49] introduced CORAL, a corpus of ontological requirements annotated with Lexico-Syntactic Patterns (LSPs). LSPs are requirement abstractions expressed using regular expression-like syntax that were first provided by Daga et al. [32]. Fernández Izquierdo et al. [49] extended the LSPs presented by Daga et al. and collected 834 CQs and statements stated for 14 ontologies. In CORAL, all requirements are linked to one of the 29 predefined LSPs, recommendations of OWL constructs that can be used to formalize knowledge, and appropriate DL expressivity.

An example of a requirement in CORAL is: The devices can be classified into categories: FunctionRelated, EnergyRelated, and BuildingRelated which is related to the following LSP:

```
NP<superclass> be | CATV [PARA] [(NP<resource1>, )* and | or ], NP<subclass>
```

, where `NP`, `CATV`, and `PARA` represent a noun phrase, verb of classification, and paralinguistic symbol (e.g., a colon), respectively.

This kind of requirement is suggested to be encoded in OWL using classes and rdfs:subClassOf keyword.

CORAL helps engineers guide their modeling decisions showing how general patterns in CQs can be formalized in ontologies. However, the dataset is created manually and does not introduce automation procedures.

Ren et al. [148] collected 145 CQs and transformed them into patterns by replacing ontology-dependent parts with placeholders representing their modeling decisions. For example, the CQ: What pizza has the lowest price? maps to the What [CE] has the [NM] [DP]? pattern. This pattern indicates that pizza represents as a class expression (CE), price is a data property (DP), and the word lowest represents a numeric modifier (NM). The patterns are then aggregated into archetypes depending on the features list providing information on:

- Question type indicating if the CQ is a binary, counting, or list question,
- The presence of implicit elements,
- The polarity of the question, whether it is expressed in a positive or negative manner,
- Relation type, depending on whether the data or object property is used,
- Quantity and numeric modifiers,
- Domain-independent elements, representing time or place.

The patterns and archetypes are used to construct authoring tests, checking the ontology quality in terms of satisfiability of class expressions and the presence of vocabulary. This dataset is constructed manually and gives engineers guidance on how CQs can be tested. However, it does not provide any automation procedure. Moreover, the CQ patterns introduced are entangled with the knowledge representation so that the information, e.g., whether a given property is of data or object type, is encoded in the pattern.

Suarez-Figueroa et al. [168] provided guidelines on how ontology requirements should be collected to form the Ontology Requirements Specification Document. These guidelines define several steps that aim to ensure the good quality of the collected requirements. These are purpose, scope, ontology language, end-users, and intended uses identification, formalization of requirements, which are further grouped, validated, and prioritized.

Rao et al. [143] discussed knowledge elicitation techniques for deriving CQs. The authors proposed three knowledge elicitation techniques called 20 questions, triad analysis, and card sort that, used separately or jointly, can help gather knowledge from domain experts.

Bezerra et al. [15] constructed CQChecker, a tool that verifies CQs by deciding whether a question is a binary decision problem or if it is asked over the assertional or terminological part of the ontology and then answers the questions constructing a SPARQL query or using Pellet reasoner. CQChecker introduces 14 patterns of CQs, e.g., Does ⟨class⟩ + ⟨property⟩ ⟨class⟩?. However, the technical details of how the queries are constructed and the procedure of pattern extraction are not provided.

5.3 Ontology modeling styles

Knowledge can be modeled in OWL ontologies in various ways. For example, the notion of parenthood can be expressed as an object property between a parent and a child. Alternatively, one can introduce a class named Parent and model all parents as its subclasses. For this reason, Gangemi [59] introduced the notion of ontology design patterns (ODPs) as design choices that can be used to address recurrent ontology design problems. They support ontology engineers showing how knowledge should be modeled and make ontology integration easier, as the modeling choices may be shared among multiple ontologies.

Espinoza-Arias et al. [45] analyzed the domain of smart cities to extract ODPs related to that field. Their analysis of 15 ontologies and their requirements resulted in a list of 7 ODPs representing, e.g., objects located in cities or public services.

Ławrynowicz et al. [92] used frequent tree-mining algorithms to extract potential ODPs from a large set of axioms stated in 331 ontologies from BioPortal [183]. They showed that some patterns are widespread as they have more than 300 000 occurrences and found relations between the existing ODPs and those mined by them.

Mortensen et al. [118] encoded 68 ODPs using the Ontology PreProcessor Language (OPPL) [44] to verify which of them are prevalent in BioPortal ontologies. Their analysis proved that 33% of ontologies use at least one ODP.

Wang et al. [179] analyzed over 1300 ontologies to see what trends are shared among them and how engineers represent knowledge in these.

5.4 Entity linking

The task of relating mentions in a piece of text to unique identifiers (e.g., IRIs in ontologies) is called Entity linking (EL) [142]. Entity Linking aims to disambiguate mentions having multiple meanings (e.g., Apple can represent a fruit or a company name). Moreover, it allows identifying cases where different names represent the same thing (e.g., flu and influenza).

Classical approaches to EL frequently introduce sequences of processing steps to solve this kind of task. Ling et al. [98] proposed a pipeline-based solution that consists of mention identification using NER, dictionary-based entity candidate generation, probability estimation incorporating entity types, coreference resolution, and coherence estimation.

Modern solutions are frequently deep learning-based [25]. Wu et al. [190] proposed BLINK, an EL solution linking mentions to Wikipedia entries. BLINK implements a two-stage BERT-based [38] zero-shot linking algorithm. First, a bi-encoder embeds the context of a given mention and an entity description into the embedding space. Then, a cross-encoder ranks candidates.

Yamada et al. [193] constructed LUKE, a state-of-the-art solution regarding the Named Entity Recognition task. LUKE stands for **l**anguage **u**nderstanding with **k**nowledge-based **e**mbeddings and is found useful to solve the EL problem. This method incorporates a modification of BERT called RoBERTa [100], which LUKE modifies by adding a new pretraining objective and incorporating an entity-aware self-attention mechanism.

Ravi et al. [144] proposed CHOLAN: a modular approach for neural entity linking on Wikipedia and Wikidata. This solution is another pipeline-based one, consisting of two models based on the Transformer [177] architecture. The first model finds entity mentions in the text, and then, for each mention, the second one performs classification into a predefined list of candidate entities. The second Transformer is enriched with context information providing the entity description collected from Wikipedia and the local context of the mention.

Cao et al. [25] proposed GENRE, a **g**enerative **en**tity **re**trieval model that uses a sequence-to-sequence approach to generate entity names in an autoregressive way conditioned on the context. This approach is built based on a pretrained Transformer-based architecture [177] named BART [97], and further fine-tuned to generate entity names. GENRE can solve the EL task in an end-to-end manner. Currently, it is the state-of-the-art solution on several benchmarks (e.g., AIDA-B [72, 25]).

5.5 Question generation

The problem of question generation (QG) based on an information source is an important field of NLP. QG is applied to question answering or computer-aided education, among others [151, 196]. To solve that problem, researchers proposed various approaches including rule-based text transformations and generative deep neural networks.

Fabbri et al. [47] proposed a template-based QG method that uses predefined sentence transformations to build questions. Given an input text with tokens marked as the answer, the question asking about the masked sequence is constructed. Depending on the type of the masked entity, an appropriate wh-word is used in the generated question (e.g., who to ask for a person). Du et al. [40] created a deep learning-based end-to-end approach that, instead of templates, uses LSTM [70] cells and GloVe [131] embeddings to generate questions. Duan et al. [41] used convolutional neural networks [56, 94] and RNN cells to train a system on a large-scale set of questions related to answers. They show that the questions generated with their method may serve as silver-standard datasets in question answering tasks, increasing the quality of the produced solutions.

Alsubait [4] and Stasaski et al. [164] showed that it is possible to generate questions automatically from ontologies. The formal knowledge modeled in ontologies may provide groups of distractors (i.e., wrong answer candidates) in multichoice questions.

Lee et al. [95] proposed a model called Info-HCVAE to address the data scarcity problem in question answering tasks. They use a hierarchical conditional variational autoencoder [87] to produce pairs of questions and answers. They also propose an infoMax regularizer that maximizes the mutual information between questions and answers to enforce consistency.

Xiao et al. [191] proposed ERNIE-GEN, an enhanced multi-flow pretraining and fine-tuning framework for natural language generation. This method uses a deep neural network that incorporates infilling generation step that forces the model to focus more on the former words. Moreover, ERNIE-GEN increases the robustness of the training process by replacing a portion of tokens in the training data with random ones to add noise. Those novelties and the span-by-span generation scenario they propose make the model one of the best solutions in the context of question generation or automatic text summarization.

The methods described in this section are mainly focused on generating a small number of precise questions. However, in Chapter 8, we discuss a scenario in which it is beneficial to construct a large and diverse set of query forms related to a given input.

5.6 Translating text to structured queries

The problem of transforming natural language into structured queries stated for knowledge graphs and databases is the main focus of the knowledge-based question answering (KBQA) research area [39]. Many groups compete in challenges or benchmarks such as Question Answering over Linked Data (QALD) [30]. Most of the approaches apply pipeline-based methods that use semantic parsing of questions expressed in natural language, mapping fragments of questions to entities in KBs, and constructing formal representations that can be understood by knowledge bases.

Natural language processing approaches such as POS-tagging, dependency parse tree construction, or regular expressions are used to understand the question. To find the question type, PowerAqua by Lopez et al. [101] uses regular expressions, TBSL by Unger et al. [173] uses POS-tags and a set of heuristics to group tokens.

To find relations between the question and the KB, QAKiS by Cabrio et al. [24] uses a named entity recognition model to identify phrases that should be linked with the KB. SINA by Shekarpour et al. [161] and CASIA by He et al. [67] produce n-grams of tokens in a question to link them to the entities available in the KB.

Various strategies are proposed to construct SPARQL queries. Approaches such as QAKiS by Cabrio et al. [24], ISOFT by Park et al. [129], or PowerAqua by Lopez et al. [101] use simple templates to construct SPARQL queries from questions. Approaches such as FREyA by Damljanovic et al. [33], DEANNA by Yahya et al. [192], or QAnswer by Ruseti et al. [152] propose to parse questions and use relations between entities mentioned in each question to find entities in the KB. These are then arranged into triples used to create SPARQL queries. Zemmouchi-Ghomari et al. [195] proposed a CQ into SPARQL translation approach, which consists of question type identification, expected answer determination, entities extraction, entity type recognition, and query construction. However, this approach requires manual extraction of entities and manual linking of CQ phrases to ontological entities.

Templates are frequently constructed manually or semi-manually. However, some works focus on generating them automatically. Cui et al. [31] provided a method that can learn templates (e.g., How many people are there in $city?, where $city represents a placeholder).

Zheng et al. [198] proposed to incorporate an uncertain graph join task to find the best SPARQL query match for a question stated in natural language.

Other examples of QA systems for RDF data are AskNow by Dubey et al. [42], ORAKEL by Cimiano et al. [29], AquaLog by Lopez et al. [102], and Cube-QA by Hoeffner et al. [73].

An overview of approaches for question answering over knowledge bases is presented in [39].

The problem of question answering is related to the automatic translation of CQs into SPARQL-OWL. However, the translator provided in this dissertation is devoted to supporting the ontology engineering workflow. It is used to validate and verify the constructed ontology. In contrast to QA methods, we do not assume that the answers are provided in the ontology, and we do not focus on the assertional part of the ontology.

Chapter 6

CQ2SPARQLOWL: a dataset of CQs translated into SPARQL-OWL queries

The goal of making the process of translating CQs into SPARQL-OWL queries automatic can be reached if the relationship between these two formalizations is well explored. Knowledge of how competency questions and their translations are constructed as well as how the presence of specific phrases influences the choice of query language constructs can help to understand how to generate queries from questions without human intervention. However, no existing dataset of multiple ontologies providing CQs translated into SPARQL-OWL queries was available. To fill this gap, in this chapter, we describe and analyze our own dataset named CQ2SPARQLOWL that consists of five ontologies and their CQs translated into SPARQL-OWL queries. We published the dataset as well as its analysis on Github[1], and described CQ2SPARQLOWL in two peer-reviewed journal papers [189, 135].

This chapter is structured as follows: In Section 6.1, we describe the process of collecting the dataset. In Section 6.2, we analyze how CQs are constructed and propose domain-agnostic CQ patterns in Section 6.3. In Section 6.4, we analyze queries and analyze their abstract, domain-independent forms in Section 6.5. We discuss the relation between CQs and queries in Section 6.6. In Section 6.7, we discuss the relation between CQ patterns and SPARQL-OWL query signatures. In Section 6.8, we summarize the observations.

6.1 Dataset collection process

In this section, we provide a general overview of the dataset. We start with the description of the origins of CQ2SPARQLOWL, focusing on how we collected ontologies and CQs as well as how we generated SPARQL-OWL queries.

Ontologies and CQs The core of CQ2SPARQLOWL is a set of publicly available ontologies describing diverse domains and created by various researchers. Although hundreds of ontologies are available online[2], only a few of them come with CQs involved in ontology development.

Moreover, even if CQs for a given ontology are provided, they are unlikely to be formalized as SPARQL-OWL queries. To compile a dataset of CQs translated into SPARQL-OWL queries, we collected a set of publicly available ontologies that have CQs published along and manually

[1]`https://github.com/CQ2SPARQLOWL/Dataset`

[2]For example, the website `https://lov.linkeddata.es/dataset/lov/vocabs/owl` provides a non-exhaustive list of 280 ontologies expressed in OWL.

generated SPARQL-OWL queries where required. Each ontology accompanied by its CQs, in order to be included in CQ2SPARQLOWL, had to meet each of the following criteria [189]:

1. For a given ontology, there has to be a technical report or a peer-reviewed published paper available.
2. The CQs must have questions for the terminological part of an ontology.
3. Either SPARQL-OWL queries must be already provided as formalizations of CQs, the expected answers to CQs should be listed, or the domain of a given ontology should be familiar to us.

The first criterion prevents low-quality data from being included in CQ2SPARQLOWL. The second one ensures that at least some queries generated for a given ontology would benefit the SPARQL-OWL language as described in Section 2.6. Finally, the third criterion ensures that we are competent to construct SPARQL-OWL queries if they are not already provided.

With a simple web search, using keywords such as ontology, competency questions, cq1, we identified 5 ontologies meeting each of the aforementioned criteria. These ontologies are:

1. The Software Ontology (SWO) [106] – an ontology describing software, software types, licenses, tasks and data formats. This ontology comes with 90 CQs.
2. Dementia Ambient Care Ontology (Dem@Care) [165] – an ontology describing vocabulary related to health and patients with dementia. Dem@Care comes with 107 CQs and their expected answers.
3. Ontology of Datatypes (OntoDT) [127] – an ontology describing datatypes, their taxonomies and qualities. This ontology comes with 14 CQs defined.
4. African Wildlife Ontology (AWO) [84] – an ontology describing African wildlife. AWO has 14 CQs defined.
5. Stuff Ontology (Stuff) [83] – an ontology describing macroscopic stuff (e.g., colloids, bulk, emulsions). It is provided along with 11 CQs.

The CQs collected were preprocessed in the following way [135]:

1. As we found two pairs of duplicates in the SWO ontology (CQs consisting of exactly the same sequences of characters), we removed duplicates. The duplicates are found only in the SWO CQ set, and their removal reduced the number of CQs stated for that ontology to 88.
2. For CQs missing required context (e.g., What other alternatives are there? stated for the SWO), we added the most probable one based on our expertise, the domain of the ontology, and the expected answers if they are provided (e.g., What other alternatives to this software are there?).
3. We removed comments listing possible alternatives that can be used to create additional CQs (e.g., removed (+ Cost + function) from the CQ defined for the SWO: (+ Cost + function) Which visualisation software is there for this data and what will it cost?).
4. Some CQs contain fragments of texts that are too broad to be used in CQs. Let us consider the following question: Where is the documentation of it? – in this CQ, the phrase it does not represent any particular entity labeled in the ontology as `it`, but is a kind of a placeholder

that in a CQ should be substituted with some entity label. In our data, the too-broad fragments are represented using the pronoun it or other phrases that indicate a placeholder (e.g., software X, this software, suggesting that a concrete piece of software should be used instead). In each case, we marked such too-general phrases with square brackets. Henceforth, we call such phrases in square brackets *placeholders*. In consequence, the mentioned CQ was transformed into Where is the documentation of [it]?

As a result of applying the aforementioned preprocessing steps, we collected 234 CQs. We included all 234 preprocessed CQs and all five ontologies in CQ2SPARQLOWL.

SPARQL-OWL queries None of the collected ontologies provided SPARQL-OWL translations of their CQs. For this reason, we defined the queries ourselves. Out of each CQ stated for a given ontology, a single expert familiar with the ontology domain attempted to formulate a query. In problematic cases, where an engineer was unsure if it is possible to construct a query or what form it should take, a group of 3 or 4 engineers collectively decided if it is possible to propose a query and how it should look like.

We constructed the queries using the following procedure [135]:

1. Search for phrases in CQs that may be related to ontology vocabulary and match them with ontological entities manually.
2. Identify the set of answers expected to be returned by each query.
3. Construct queries with regard to entities identified in step 1 and step 2 of this procedure.
4. Use OWL-BGP[3] to test if queries constructed in step 3 give the same answers as provided in step 2.

As a result of applying this procedure, out of 234 CQs in total, 131 were translated into queries. A detailed summary providing the numbers of translated CQs, grouped by the ontology they query, can be found in Table 6.1.

Table 6.1: Per ontology summary of the number of CQs translated into SPARQL-OWL queries.

Ontology	CQs	CQs translated into SPARQL-OWL queries
SWO	88	42
Dem@Care	107	60
OntoDT	14	13
AWO	14	7
Stuff	11	9

There are several reasons why only 131 out of 234 CQs are translated into queries. As described in [189], these are:

1. Missing vocabulary in a given ontology – 58 CQs contain phrases that refer to ontology vocabulary that is not (yet) modeled (e.g., the query for the CQ with an ID SWO_71: *Does [it] have a tutorial?* should contain the IRI of a resource representing tutorial, which is not modeled).

[3]`https://github.com/iliannakollia/owl-bgp`

2. CQ ambiguity – 26 CQs are formulated in a way that may be understood differently depending on its interpretation (e.g., SWO_74 How well documented is [the software] for developers? introduces the how well phrase. For one person, good documentation describes all use cases in-depth, while for others, it may be understood as short and concise).

3. CQs used to query the assertional rather than the terminological part of the ontology – 15 CQs are asking about the assertional knowledge. These CQs should be expressed using SPARQL rather than SPARQL-OWL queries and are considered out of scope (e.g., Dem@Care_11: What data are collected for FAB?).

4. Missing knowledge – in the case of 8 CQs the knowledge in an ontology is incomplete, which makes it impossible to construct a query (e.g., Dem@Care_95: What physiological measurements are detected? contains phrases physiological measurements and detected, which can be mapped to ontology resources labeled as `PhysiologicalMeasurement` and `Detection`. In the ontology, all we know about their relations is: `Detection` describes an `Event`, while `PhysiologicalMeasurement` is an `Event`. This knowledge cannot be used to generate the query).

Five CQs in CQ2SPARQL cannot be translated into SPARQL-OWL queries because more than one category of problems applies. The CQ: Dem@Care_26 What is the nature of a directed task?, requires that the notion of nature should be modeled in the ontology. However, because it is not, the word nature becomes a vague term, which may be interpreted in various ways: for some people – nature represents a set of abilities, while for others – it may refer to what a directed task is in terms of taxonomical relations (what is its superclass). As a consequence, this CQ cannot be translated into a query because there is missing knowledge in the ontology, and the CQ is ambiguous.

6.2 CQs analysis

We start with an analysis of CQs from 3 perspectives. The first one investigates CQ lengths to check if the complexity of questions, understood as the number of tokens, varies among ontologies. The second perspective focuses on the words that start the questions, as they frequently determine the kind of question as well as the type of answer that should be provided. The third one checks how frequently placeholders are introduced in CQs.

6.2.1 Lengths of CQs

There are domains that tend to use vocabulary consisting of lengthy words (e.g., a medical domain with entities like: Achondrogenesis-hypochondrogenesis, Acromioclavicular joint, Echocardiography), while others may introduce mostly short names (e.g., programming languages domain with entities like: C++, R, Java, Python). Since some domains may introduce longer words than others, we decided to analyze the length of a given CQ as the number of tokens it contains. Such analysis can help understand if there is a difference in the complexity of CQs stated for various ontologies. To achieve it, we used `word_tokenize` function from NLTK [16] to split CQs. We did not count square brackets as tokens, since they are artificial markers indicating placeholders and are a by-product of CQs preprocessing procedure described in Section 6.1.

The comparison of CQ lengths stated for different ontologies is provided in Figure 6.1. As can be seen, the average length of a CQ is 9 tokens, and 50% of CQs in CQ2SPARQLOWL consist of 8 to 11 tokens.

The shortest CQs are made of 4 tokens (ignoring square brackets): SWO_65: Is [it] FOSS?, SWO_35: Who developed [it]?, SWO_77: Is [it] scriptable?, SWO_78: Is [it] extensible?. The longest CQs consist of 22 tokens, these are: Dem@Care_71: What food and drink preparation-related situations indicate a problem or possibly problematic behaviour that needs to be highlighted to the clinician? and Dem@Care_72: What food and drink consumption-related situations indicate a problem or possibly problematic behaviour that needs to be highlighted to the clinician?.

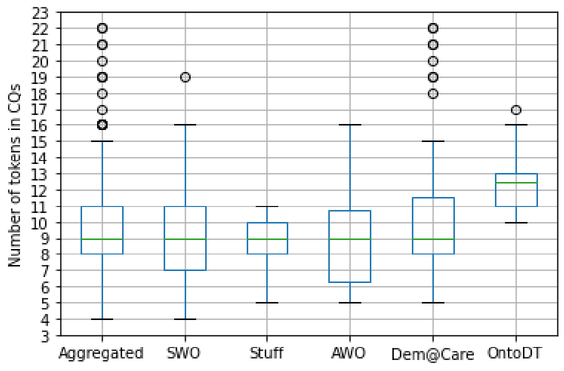


FIGURE 6.1: The number of tokens among CQs. The boxplot named as 'Aggregated' is related to the whole CQ2SPARQLOWL. In this figure, heights of the rectangles represent interquartile ranges (IQR) defined as differences between third and first quartiles (Q3 and Q1, respectively). Each whisker represents a range from Q1 - 1.5 IQR to Q3 + 1.5 IQR. The circles represent outliers.

6.2.2 Words at the beginnings of CQs

The CQs can be grouped according to the words they start with. In CQSPARQLOWL the following question starters are observed:

1. What (used in 144 CQs), Which (21 CQs) – These words start CQs expected to obtain entities meeting the given restrictions (e.g., OntoDT_03: What is the value space for [a datatype X]?).
2. Is (16 CQs), Are (6 CQs) – In 19 cases these CQs should be answered with either yes or no depending on the state of affairs in the ontology (e.g., Stuff_06: Are solutions never emulsions?). In the remaining 3 cases, the question starting with Is should be answered with one or more alternatives explicitly mentioned in the CQ (e.g., Stuff_01: Is [this stuff] a pure or a mixed stuff?).
3. How (13 CQs) – In 6 cases they are expected to provide procedures (e.g., SWO_32: How can I get problems with [it] fixed?). In the remaining cases, How introduces a question about a measure or quality:
   - In 4 cases the word How is used to ask for intensity, magnitude or extent of a given thing or process (e.g., SWO_47: How reliable is [it]?).
   - In 2 cases it is used to ask for a quantity of something (e.g., SWO_64: How many licenses do we need to run [it] productively?).
   - In one case, How is used to ask for a time period (SWO_44: How long has [this software] been around?).
4. Can (7 CQs) – 3 CQs started with Can are used to ask for capabilities of modelled entities (e.g., SWO_22: Can [software A] work with data that are output from [software B]?) and the

remaining 4 to ask for capabilities of people using the ontology (e.g., SWO_27: Can I render [it] if the software supplier goes out of business?).

5. Does (6 CQs), Do (4 CQs) – are used to express binary questions that should be answered with either yes or no, similarly to Is and Are (e.g., SWO_41: Does [this software] meet the ISO-4 standard?).
6. Where (6 CQs) – is used to ask for locations (e.g., SWO_29: Where can I get [the software]?).
7. Who (4 CQs) – is used to ask for agents (e.g., SWO_62: Who owns the copyright for [it]?).
8. In (what/which) (3 CQs) – is used in the same context as What and Which (e.g., AWO_11: In what kind of habitat do [this animal] live?).
9. When (1 CQs) – is used to ask about a point in time (SWO_57: When was the 1.0 version of [it] released?).
10. Given (1 CQ) – is used to add the context to the question (SWO_28: Given [input x], what are the data exports for [this version] of [x]?)
11. At (1 CQ) – is used to ask for a point in time (SWO_61: At what point did the license type of [it] change?).
12. To (1 CQ) – is used to ask about the extent (SWO_23: To what extent does [the software] support appropriate open standards?).

Among all 234 CQs collected, 6 of them are expected to return more than one category of things at once:

- SWO_10: What are the primary inputs and outputs [of this software]?
- SWO_11: Which visualisation software is there for [this data] and what will it cost?
- SWO_16: What are the input and output formats for [this software]?
- SWO_59: What license does [it] have, and what is its permissiveness?
- SWO_70: Is there any documentation for [it] and where can I find it?
- Dem@Care_56: How are the statistics and identified problematic situations about the monitored functional areas reported to the clinician?

No CQ in CQ2SPARQLOWL asks for more than 2 categories of things in a single CQ.

6.2.3 Materialized vs dematerialized CQs

Almost 50% (116 out of 234) CQs contain one or more placeholders introduced during step 4 of CQs preprocessing procedure described in Section 6.1. These are special-purpose sequences of tokens wrapped with square brackets, which cover words or phrases that are too general to be used in a CQ but are expected to be replaced with more concrete ontological entities to form CQs.

For example, the CQ: SWO_06: Does [this software] provide XML editing? contains a placeholder [this software], which informs that any piece of software defined in the modeled domain can be used as a substitute for the placeholder to construct multiple CQs.

As a consequence, CQs with placeholders (further called dematerialized CQs) are just templates to produce multiple similar CQs and should be used to query ontologies only if the placeholders are substituted with phrases representing ontological entities, either already modeled or expected to be modeled. We refer to CQs introducing no placeholders or CQs with all placeholders substituted as materialized. In CQ2SPARQLOWL, there are 13 CQs introducing two placeholders each and 2 CQs introducing 3 of them each.

From the analyzed dematerialized CQ SWO_06: Does [this software] provide XML editing?, the following examples of materialized CQs can be produced:

- Does Weka provide XML editing?
- Does AIDA provide XML editing?
- Does MATLAB provide XML editing?

Table 6.2: Number of materialized and dematerialized CQs per ontology.

Ontology	Materialized CQs	Dematerialized	Fraction of materialized
SWO	1	87	1.1%
Dem@Care	107	0	100%
OntoDT	0	14	0%
AWO	6	8	42.9%
Stuff	4	7	36.4%

As can be seen, introducing dematerialized CQs can help to increase the number of CQs. In the context of such questions, for example, the analyzed CQ SWO_06 Does [this software] provide XML editing?, an automatic tool could extract all subclasses of `software` in the SWO, and for each of them, use the subclass label to substitute the placeholder and generate a new CQ. Since there is a relationship between placeholders and ontology vocabulary, we consider placeholders as domain-dependent sequences of tokens. The prevalence of dematerialized CQs varies among the collected ontologies. The summary providing per ontology counts can be found in Table 6.2.

6.3 CQ patterns analysis

In this section, we define CQ patterns that group similarly constructed CQs. Then, we analyze which CQ patterns are there in CQ2SPARQLOWL, and which of them are shared among CQs stated for different ontologies.

6.3.1 Domain-dependent and domain-independent tokens

Each CQ is constructed using two kinds of tokens:

1. Domain-dependent tokens – these tokens or sequences of tokens make up phrases that are expected to point to entities defined in an ontology (e.g., in the CQ Dem@Care_41: What data are measured for dynamic balance?, there are 3 such phrases: data, dynamic balance and are measured for).

2. Domain-independent tokens – these do not represent any ontological entity. Tokens or phrases such as: what, when, and, or, is, are there, what kind of help to define questions that are grammatically correct, introduce the question target, and define relations between domain-dependent tokens (e.g., in the CQ Stuff_08: What distinguishes structured from unstructured

stuff? the word what informs that the distinction is expected to be the answer to the question, and distinguishes ... from introduces relation between two domain-dependent phrases structured (stuff) and unstructured stuff).

We use the distinction between domain-dependent and domain-independent tokens to define CQ patterns.

6.3.2 Domain-independent CQ patterns

Each CQ in CQ2SPARQLOWL consists of a unique sequence of characters. Considering CQs stated for a single ontology, the CQs differ because each question introduces a separate requirement the ontology should meet. Any pair of CQs stated for two different ontologies is unlikely to share domain-related phrases, since ontologies frequently describe different domains.

For this reason, we introduce a process aiming to generate domain-independent CQ patterns. Each CQ is processed using Algorithm 3 presented along with its related functions in Appendix A.1. These were introduced in [189]. In short, these procedures can be summarized as follows:

1. Normalize a given CQ using: lowercasing, redundant spaces removal, and removal of dashes.
2. Tokenize the CQ, assign POS-tags to each token, and generate the dependency parse tree.
3. Using the predefined set of rules identify entity chunks (phrases referring to things) and predicate chunks (phrases referring to actions or states).
4. For predicate chunks, attach auxiliary verbs if present.
5. Replace each chunk with a domain-independent marker in the following way:
   - For each entity chunk, use the EC{IDX} marker as its substitution. The {IDX} part represents a numerical identifier unique for each chunk of a given type.
   - For each predicate chunk, use the PC{IDX} marker as its substitution. The {IDX} part represents a numerical identifier unique for each chunk of a given type.

   For example, considering the CQ: Which countries are larger than Germany?, the rules would classify countries and Germany as entity chunks and are larger than as a predicate chunk. These would be replaced with EC and PC chunk markers forming domain-independent form: Which EC1 PC1 EC2?.
6. Validate the generated output and manually fix errors made by the POS-tagger and imperfect rules that were designed to aid human verification only.

The application of the described procedure to each of the 234 CQs from CQ2SPARQLOWL transforms each CQ into a domain-independent CQ pattern candidate. Since a pattern is a regularly repeated arrangement, at least one condition introduced below must be satisfied to classify a given pattern candidate as a pattern:

1. A given pattern candidate is observed more than once in CQ2SPARQLOWL.
2. A given pattern candidate is created from a dematerialized CQ (CQ containing placeholders).

The first condition ensures that all repeated pattern candidates are classified as patterns. The second one includes candidates that may not be explicitly repeated in CQ2SPARQLOWL, but since they are dematerialized, they are expected to be transformed in multiple similar CQs.

From all 234 CQs, 106 distinct patterns are extracted. Table 6.3 summarizes the number of pattern candidates, patterns extracted for each ontology, and the percentage of CQs covered by patterns. We say that a given CQ is covered by a given pattern if the pattern candidate extracted from the CQ is equal to the pattern. The first column in the table, named `Pattern Candidates`, shows from how many CQs pattern candidates are constructed. Because the algorithms presented in Appendix A.1 always provide a pattern candidate for a given CQ, the number in this column is always equal to the number of CQs stated for the ontology.

The second column, named `Patterns` shows how many CQ pattern candidates represent patterns. In OntoDT, all CQs are covered by patterns, since all CQs are dematerialized. Similarly, all but one CQ from the SWO are dematerialized and the only remaining CQ forms a pattern candidate that is shared among more questions. The most unique CQs can be observed in the context of Stuff. Here, only 7 CQs are covered by patterns, and the remaining 4 are unique as their pattern candidates are not repeated.

Multiple CQs may be covered by the same pattern. To check how intensively patterns are reused among the CQs, the column named `Distinct Patterns` is introduced to count how many different patterns are found in a given CQ set. This column tells us that in the case of the SWO, from the 88 CQs that are covered by patterns, we identify 72 different patterns (some patterns cover more than one CQ). In all but one CQ sets, more than 50% of CQs covered by patterns are covered by distinct patterns. An interesting outlier is Dem@Care, where only 18 distinct patterns are extracted, while 90 CQs are covered by patterns. It is due to the lack of dematerialized CQs in Dem@Care. For this reason, in Dem@Care, there are multiple similar (sharing the same pattern) CQs stated so that the 18 distinct patterns are highly reused. The last column `CQs covered by patterns` presents the ratio of the total number of CQs covered by patterns to the number of CQs. We present the average number of CQs covered by a pattern in Figure 6.2. All extracted patterns are listed in Appendix B.1.

From the union of all CQs stated for each of the five collected ontologies, we extracted 106 different patterns. We found that only six of them are shared among more than one ontology (covering at least one CQ in more than one ontology). Table 6.4 provides a list of these patterns as well as the names of the ontologies where each pattern was observed.

Table 6.3: Per ontology summary of the number of CQs covered by pattern candidates and patterns as well as the number of distinct patterns and the percentage of CQs covered by patterns.

Ontology	Pattern Candidates	Patterns	Distinct Patterns	CQs covered by patterns
SWO	88	88	72	100%
OntoDT	14	14	8	100%
Dem@Care	107	90	18	84.1%
AWO	14	10	9	71.4%
Stuff	11	7	6	63.6%
Total	234	209	106	89.3%

6.3.3 Domain-independent higher-level CQ patterns

The list of patterns provided in Appendix B.1 exposes groups of patterns sharing the same semantics, but differing using synonym words, grammar, or introducing words that are not mandatory. An example pair of CQ patterns introducing synonyms is: What EC1 PC1 EC2? and Which EC1 PC1 EC2?. Here, both Which and What indicate that the formalization of a question should list the entities that belong to a set related to EC1 PC1 EC2, and can be used interchangeably.

Table 6.4: The list of patterns shared among multiple ontologies. Reprint from [189].

Pattern	CQ sets for ontologies
What EC1 PC1 EC2?	SWO, Dem@Care
Which EC1 PC1 EC2?	SWO, AWO
What are EC1 for EC2?	SWO, OntoDT
What is EC1 for EC2?	SWO, OntoDT
What is EC1 of EC2?	SWO, AWO
Which EC1 are EC2?	Dem@Care, AWO

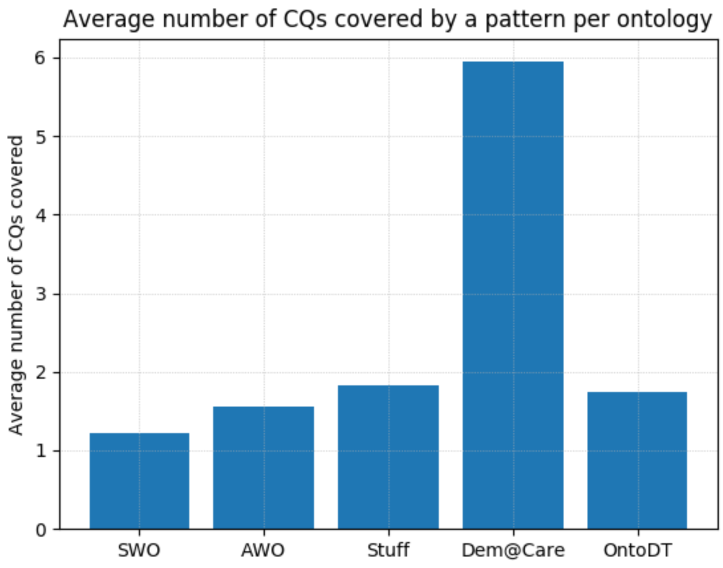


Figure 6.2: Average number of CQs covered by a pattern. Reprint from [189].

A difference induced by the grammar can be presented using a pair: What is EC1 for EC2?, What are EC1 for EC2?. The decision on whether to use is or are depends on whether EC1 is in the singular or in the plural form.

An example of a difference caused by a word that is not mandatory may be: Is there EC1 for EC2 and Are there any EC1 for EC2. Here, apart from the difference between the be verb forms, only one CQ pattern introduces the word any. However, if the word was omitted, the meaning of the question would not change.

To make CQ patterns robust to such small differences, based on the CQ patterns list, we introduced the normalization procedure, transforming similar CQ patterns into a common form. Table 6.5 lists tokens or token sequences that are normalized and presents the normalization steps.

Moreover, our design choice behind constructing CQ patterns was to treat noun phrases that are separated by a preposition as separate entities (entity chunks). However, considering SWO_68: What level of expertise is required to use [it]? as an example, the ontology engineer using this CQ has to decide whether to model level of expertise as a single class, or to introduce separate level and expertise classes to the modeled ontology. As we cannot be sure what modeling decision the ontology engineer makes, we cannot predetermine if the CQ pattern should be of form What EC1 of EC2 PC1 EC3? or What EC1 PC1 EC2?, where the EC1 stands for the whole level of expertise. Therefore, to group CQ patterns, we also introduced an additional normalization step aggregating EC chunks separated with in/from/with/of into a single EC chunk.

The application of both normalization steps to the CQ patterns set introduces a new set of higher-level patterns listed in the Appendix B.2. The higher-level patterns set consists of 81 entries.

Using normalization, we can transform the following CQ patterns:

1. *What is EC1 of EC2?*
2. *Which are EC1 of EC2?*
3. *What is EC1 for EC2?*
4. *What is EC1 of EC2 for EC3?*
5. *What are EC1 for EC2?*
6. *What is EC1?*
7. *What are EC1?*

into a common higher-level pattern What is EC1?

Regarding higher-level patterns, more of them are shared among multiple ontologies as can be seen in Table 6.6. However, no higher-level CQ pattern is shared among all ontologies. Table 6.7 provides the comparison between the number of patterns and higher-level patterns per each ontology. In Figure 6.3, we present how many CQs are covered by higher-level patterns per each ontology.

TABLE 6.5: The transformations used to normalize similar CQ patterns. The special token "—" represents situations where a given text is removed from the pattern during normalization. Reprint from [189].

Textual pattern	Normalized form
are	is
any	—
did	do
we	I
does	do
which of	which
has	have
which kind	what kind
will	is
Which (if at the beginning of a sentence)	What
possible	—
are there	—

TABLE 6.6: Higher-level patterns shared among multiple ontologies. Multiple occurences of PC1 in a single pattern are due to auxiliary verbs that we consider parts of predicate chunks (e.g., does support in What formats does Weka support? represent a single predicate chunk even if does and support are separated with another word. Reprint from [189].

Pattern	In CQ sets for ontologies
What type of EC1 is EC2?	SWO, Stuff, Dem@Care
What EC1 PC1 EC2?	SWO, Dem@Care, AWO
What is EC1?	SWO, OntoDT, Dem@Care
What EC1 PC1 I PC1 EC2?	SWO, AWO
Is EC1 EC2?	SWO, AWO
Is there EC1?	SWO, AWO
What EC1 PC1 EC2 PC1?	SWO, AWO
What EC1 is EC2?	Dem@Care, AWO

TABLE 6.7: Comparison of the number of patterns and higher-level patterns per ontology.

Ontology	Distinct patterns	Distinct higher-level patterns
SWO	72	60
Stuff	6	5
AWO	9	8
Dem@Care	18	15
OntoDT	8	4
Total	106	81

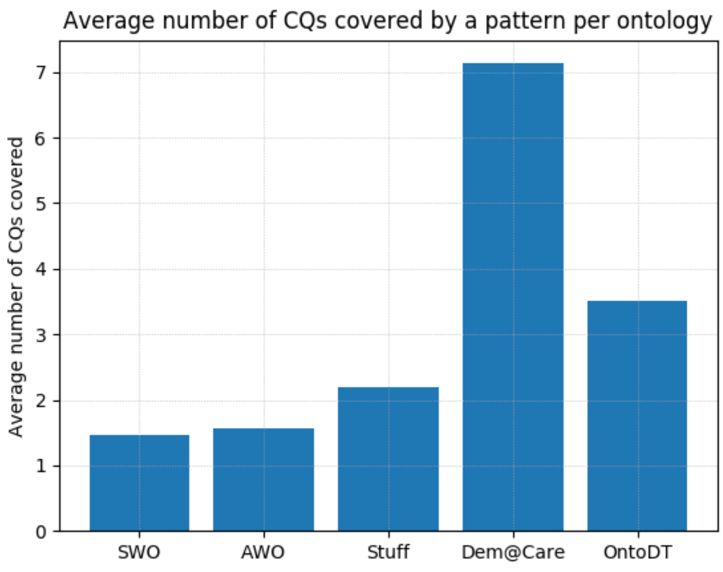


FIGURE 6.3: Average number of CQs covered by a higher-level pattern. Reprint from [189].

6.4 SPARQL-OWL queries analysis

In this section, we analyze how SPARQL-OWL queries are constructed and what groups of queries arise in our dataset.

6.4.1 Query forms used in CQ2SPARQLOWL

To keep the analyzed queries short and easy to read, from now on, for each query, we use prefixes as defined in Table 6.8.

There are four query forms of SPARQL queries defined in the SPARQL language standard [158]:

- ASK – used to verify whether or not a query pattern can be matched in a given ontology.
- SELECT – used to list values of variables that can be bound to a match of a query pattern.
- DESCRIBE – used to construct an RDF graph that provides a description of the resource(s) found.
- CONSTRUCT – used to obtain an RDF graph created by substituting variables in a triple templates set.

Table 6.8: Prefixes used in SPARQL-OWL queries. Reprint from [135].

Prefix	Namespace
rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs:	http://www.w3.org/2000/01/rdf-schema#
owl:	http://www.w3.org/2002/07/owl#
xsd:	http://www.w3.org/2001/XMLSchema#
awo:	http://www.meteck.org/teaching/ontologies/AfricanWildlifeOntology1.owl#
stuff:	http://www.meteck.org/files/ontologies/stuff.owl#
event:	http://www.demcare.eu/ontologies/event.owl#
exch:	http://www.demcare.eu/ontologies/exchangemodel.owl#
home:	http://www.demcare.eu/ontologies/home.owl#
lab:	http://www.demcare.eu/ontologies/lab.owl#
swo:	http://www.ebi.ac.uk/swo/
efo-swo:	http://www.ebi.ac.uk/efo/swo/
maturity:	http://www.ebi.ac.uk/swo/maturity/
interface:	http://www.ebi.ac.uk/swo/interface/
license:	http://www.ebi.ac.uk/swo/license/
obo:	http://purl.obolibrary.org/obo/
OntoDT:	http://www.ontodm.com/OntoDT#
OntoDT2	http://ontodm.com/OntoDT#

Out of these possibilities, only ASK and SELECT are used in CQ2SPARQLOWL. In Table 6.9 we summarize how many of them are provided per each ontology. In total, 87% of queries are of SELECT type and in case of two ontologies: OntoDT and Dem@Care no query of the ASK type is provided.

Table 6.9: Number of ASK and SELECT queries per ontology.

Ontology	Count ASK	Count SELECT
SWO	12	30
Dem@Care	0	60
OntoDT	0	13
AWO	3	4
Stuff	2	7
Total	17	114

6.4.2 Solution modifiers

There are 6 solution modifiers that can be used in SPARQL (and SPARQL-OWL) queries [158]. These are `PROJECTION, DISTINCT, ORDER, LIMIT, REDUCED` and `OFFSET`. From that list, `DISTINCT` can be found in CQ2SPARQLOWL, which is used in 71 queries (4 times in AWO, 57 in Dem@Care, 4 in SWO, and 6 in Stuff). The `DISTINCT` keyword used in a query assures that in the solution list generated, no two solutions are representing the same results. The projection is used to select a subset of variables to be returned to the user in 7 queries in CQ2SPARQLOWL (2 times in Stuff and 5 times in SWO).

6.4.3 Basic graph patterns

The most variability among SPARQL-OWL queries comes from applying different BGPs. In this paragraph, we discuss the most interesting cases.

In CQ2SPARQLOWL, the most commonly used BGP is a single existential property restriction or a single `owl:hasValue`. 41 out of 131 queries in total are conforming to this kind of BGP.

However, this BGP is present in various forms:

- with a variable in the place of the subject, e.g., AWO_7:

```
SELECT DISTINCT * WHERE {
    ?eats rdfs:subClassOf awo:animal, [
        a owl:Restriction ;
        owl:onProperty awo:eats;
        owl:someValuesFrom $PPx1$ ] .
    $PPx1$ rdfs:subClassOf awo:animal .
    FILTER(?eats != owl:Nothing) }
```

- with a variable in the place of the object, e.g., Dem@Care_36:

```
SELECT DISTINCT * WHERE {
    lab:S1_P21_SentenceRepeatingTask rdfs:subClassOf [
        a owl:Restriction;
        owl:onProperty lab:measuredData;
        owl:someValuesFrom ?x ].
    ?x rdfs:subClassOf lab:MeasuredData.
    FILTER(?x != lab:MeasuredData) }
```

- without variables in ASK queries, e.g., SWO_88:

```
ASK WHERE {
    $PPx1$ rdfs:subClassOf [
        a owl:Restriction ;
        owl:onProperty swo:has_license ;
        owl:someValuesFrom  <http://www.ebi.ac.uk/swo/license/SWO_1000002>].}
```

- introducing multiple variables, e.g., AWO_1:

```
SELECT DISTINCT * WHERE {
    ?eats rdfs:subClassOf awo:animal, [
        a owl:Restriction ;
        owl:onProperty awo:eats;
        owl:someValuesFrom ?eaten ] .
    ?eaten rdfs:subClassOf awo:animal .
    FILTER(?eats != owl:Nothing) }
```

The only ontology for which such kind of query is not observed is Stuff Ontology. The `$PPx1$` sequences are placeholders related to placeholders from CQs (e.g., [it]). When a given CQ is materialized, both placeholders should be filled: the one stated in a CQ with appropriate entity label and `$PPx1$` with the IRI of that entity.

The second most common BGP relates to queries asking only about subclasses of a given named class. In CQ2SPARQLOWL there are 25 such cases, with 18 of them asking about proper subclasses, e.g., Dem@Care_29:

```
SELECT DISTINCT * where {
    ?e rdfs:subClassOf lab:SemiDirectedTask .
    FILTER(?e != lab:SemiDirectedTask && ?e != owl:Nothing) }
```

and 7 asking about direct subclasses of a given class, e.g., Dem@Care_88:

```
SELECT DISTINCT * WHERE {
    ?e rdfs:subClassOf event:Event .
    FILTER NOT EXISTS {
        ?e rdfs:subClassOf ?f .
        ?f rdfs:subClassOf event:Event.
        filter(?f != event:Event && ?e != ?f)
    }
    FILTER(?e != event:Event && ?e != owl:Nothing) }
```

Queries about direct subclasses are related to CQs asking for main types/kinds/categories of things. Queries about subclasses of a given named class only are observed in Dem@Care and AWO.

The third big group of queries are questions about properties. These queries are observed in 24 queries from Dem@Care and are not present in any other ontology. An example of such a query is Dem@Care_101:

```
SELECT DISTINCT * WHERE {
    [] rdfs:subClassOf exch:Report, [
        owl:onProperty ?p;
        owl:someValuesFrom []
    ]. }
```

The 3 aforementioned groups constitute 69% of the query set. The remaining 31% of queries contain numerous forms of BGPs, like:

- Cardinality restriction (e.g., Stuff_07):

  ```
  SELECT DISTINCT * WHERE {
      ?stuff rdfs:subClassOf [
          a owl:Restriction ;
          owl:onProperty :hasSubStuff ;
          owl:cardinality "2"^^xsd:nonNegativeInteger ] }
  ```

- Complex BGPs with more than one property, multiple variables, intersections, unions, or functions used (e.g., OntoDT_07):

  ```
  SELECT ?x0 WHERE {
  ?x0  rdfs:subClassOf* OntoDT2:OntoDT_378476 .
  ?x0  rdfs:subClassOf [ rdf:type owl:Restriction ;
  ```

Table 6.10: Keywords/functions used in SPARQL-OWL queries collected in CQ2SPARQLOWL. Reprint from [189]. .

Keyword	Count	Occurences in ontologies
PREFIX	131	Dem@Care (60), SWO(42), OntoDT(13), Stuff(9), AWO(7)
WHERE	131	Dem@Care (60), SWO(42), OntoDT(13), Stuff(9), AWO(7)
rdfs:subClassOf	125	Dem@Care(57), SWO(42), OntoDT(13), Stuff(7), AWO(6)
SELECT	114	Dem@Care(60), SWO(30), OntoDT(13), Stuff(7), AWO(4)
owl:onProperty	96	SWO(42), Dem@Care(33), OntoDT(13), Stuff(2), AWO(6)
owl:someValuesFrom	83	SWO(31), Dem@Care(32), OntoDT(13), AWO(6), Stuff(1)
rdf:type / a	72	SWO(40), OntoDT(13), Dem@Care(11), AWO(6), Stuff(2)
DISTINCT	71	Dem@Care(57), Stuff(6), SWO(4), AWO(4)
owl:restriction	69	SWO(40), OntoDT(13), Dem@Care(8), AWO(6), Stuff(2)
NOT EXISTS	11	Dem@Care(7), SWO(2), Stuff(1), AWO(1)
FILTER	58	Dem@Care(31), SWO(16), Stuff(6), AWO(5)
owl:Nothing	34	SWO(6), AWO(4), Dem@Care(24)
ASK	17	SWO(12), Stuff(2), AWO(3)
owl:hasValue	13	SWO(13)
owl:intersectionOf	7	SWO(7)
owl:unionOf	4	AWO (2), Dem@Care(1), SWO(1)
isURI	3	SWO (3)
UNION	3	SWO(2), Dem@Care(1)
owl:disjointWith	3	Stuff(2), AWO(1)
owl:allValuesFrom	1	Dem@Care(1)
owl:cardinality	1	Stuff(1)
rdf:first	1	Dem@Care(1)
rdf:rest	1	Dem@Care(1)
STRSTARTS	1	SWO(1)
BIND	1	SWO(1)

```
        owl:onProperty OntoDT:OntoDT_0000405;
        owl:someValuesFrom ?x1
 ] .
?x1 rdfs:subClassOf [ rdf:type owl:Restriction ;
        owl:onProperty obo:OBI_0000298;
        owl:someValuesFrom $?PPx1$
 ] .
$PPx1$ rdfs:subClassOf OntoDT2:OntoDT_283020 .
}
```

- Operations on fetched results (e.g., SWO_44, asking about the period of time between a given point in time and the present moment):

```
SELECT ?result WHERE { $PPx1$ rdfs:subClassOf swo:SWO_0000001,[
    a owl:Restriction ;
    owl:onProperty maturity:SWO_9000068 ;
    owl:hasValue ?date
]
bind(now()-xsd:dateTime(?date) as ?result) }
```

Table 6.10 lists all SPARQL, RDF(S), and OWL keywords observed among the queries collected in CQ2SPARQLOWL. As can be seen, 16 of them are used in the context of at least two ontologies.

6.5 SPARQL-OWL query signatures

A signature of a SPARQL-OWL query represents a domain-independent pattern similar to CQ patterns, which is obtained from the query using the following steps:

1. Transform a given query into a SPARQL algebra expression, also expand every abbreviation inherited from Turtle serialization (e.g., `?x a :C, :D` is expanded into `?x a :C . ?x a :D`) in the process.
2. Remove solution modifiers.
3. Remove, from each BGP expressed in the SPARQL algebra expression, triple patterns of forms (·, `rdf:type`, `owl:Restriction`) and (·, `rdf:type`, `owl:Class`), where · stands for any node [189].
4. Remove expressions of the form `?var != owl:Nothing`, where `?var` represents a blank node or a variable. Remove a filter if the aforementioned comparison was the only expression used in the filter. Filtering applied for `owl:Nothing` is a user preference: some experts may prefer to remove them from the results while others may prefer to keep them. With this step we make the decision consistent.
5. Remove symbols: + and * from property paths in cases where + and * refer to a known transitive property [189] (e.g., `rdfs:subPropertyOf` or `rdfs:subClassOf`). With respect to the SPARQL-OWL entailment regime these symbols are redundant.
6. Replace, to remove redundancy, a property path of the form `rdf:type/rdfs:subClassOf` with `rdf:type` [189].
7. Merge all BGPs being arguments of a single SPARQL join operation present in the SPARQL algebra expression to generate a single BGP [189].
8. Replace every IRI from namespaces other than OWL, RDFS, RDF and XSD with new blank nodes in a consistent manner, i.e., the same IRI is consistently replaced with the same blank node within a query. This allows to generalize queries among more questions.

The aforementioned steps are formalized in Algorithm 10 that can be found in Appendix A.2. The list of steps does not take into consideration differences in the order of triple patterns, the usage of various variable names, and blank nodes among different queries, so that queries with different names of variables or with different order of triple patterns are considered different. To overcome that limitation, we propose the definition of signature equivalence. Two signatures S1, S2 are equivalent if and only if it is possible to find a one-to-one mapping (i.e., a bijection) from the set of variables and blank nodes in S1 to the set of variables and blank nodes in S2 such that by applying to S2 one obtains an expression that is isomorphic with S1, i.e., identical to S1 barring the order of operators [189].

Let us consider the following pair of queries:

```
SELECT DISTINCT * WHERE {
    _:c1 rdfs:subClassOf
      lab:DemographicCharacteristicsRecord, [
        owl:onProperty ?p;
        owl:someValuesFrom _:c2
    ].
}
```

that is a formalization of Dem@Care_3 and

```
SELECT DISTINCT * WHERE {
    ?eats rdfs:subClassOf :animal, [
    a owl:Restriction ;
    owl:onProperty :eats;
    owl:someValuesFrom :impala].
    filter(?eats != owl:Nothing)
}
```

that is a formalization of a materialized AWO_07.

These queries are transformed into the following signatures:

```
_:c1 rdfs:subClassOf _:c4.
_:c1 rdfs:subClassOf _:c3.
_:c3 owl:onProperty ?p.
_:c3 owl:someValuesFrom _:c2.
```

created from Dem@Care_03 and

```
?eats rdfs:subClassOf _:b2.
?eats rdfs:subClassOf _:b1.
_:b1 owl:onProperty _:b3.
_:b1 owl:someValuesFrom _:b4.
```

created from AWO_07. Because we can observe the following bijection between blank nodes and variables:

`_:c1`$\mapsto$`?eats`,
`_:c3`$\mapsto$`_:b1`,
`_:c4`$\mapsto$`_:b2`,
`?p`$\mapsto$`_:b3`,
`_:c2`$\mapsto$`_:b4`,

we consider both signatures as equivalent so that we can say that both queries share the same recurring pattern.

From all 131 SPARQL-OWL queries, the signature construction algorithm created 46 distinct query signatures. 9 of them are shared among more than a single query as provided in Table 6.11. 6 signatures are shared among at least two ontologies as listed in Table 6.12. Considering the signatures presented in Table 6.11, the signature:

```
    [] rdfs:subClassOf [], [owl:onProperty []; owl:someValuesFrom []]
```

is the most widely used, being shared by 27 queries (20.5% of the queries in CQ2SPARQLOWL) stated against 4 out of five ontologies. The majority of queries in CQ2SPARQLOWL are constructed using recurrent patterns. 82 queries (62.6% of queries in CQ2SPARQLOWL) are constructed with queries having signatures shared by at least 3 queries. Dem@Care is the ontology that uses such signatures the most, with 57 out of 60 (95%) SPARQL-OWL queries covered by some signature introduced in Table 6.11. We say that a signature S covers a query Q if the signature S_Q generated from the query Q is equivalent to S. In contrast, Stuff contains queries, each of which produces a signature different than those presented in Table 6.11 so that 0 out of 9 (0%) of queries are covered by signatures. Considering other ontologies, 8 out of 13 (62%) for OntoDT, 4 out of 7 (57%) and 13/42 (33%) of SPARQL-OWL queries are covered by signatures from Table 6.11.

Focusing on the signatures shared among at least 2 ontologies, 49 queries are covered by the 6 signatures presented in Table 6.12.

Considering the number of queries transformed into signatures that are not shared by any two or more ontologies, based on the results presented in Table 6.12, we see that SWO contains 34 (85%), Stuff 7 (78%), AWO 5 (71%), Dem@Care 29 (48%) and OntoDT 5 (38%) of such ontology-wise unique signatures.

TABLE 6.11: The list of signatures that are shared by more than 2 queries in CQ2SPARQLOWL. The number of occurrences of a given signature in a given ontology is provided in the column Ontologies. Reprint from [189].

Signature	Ontologies
`[] rdfs:subClassOf [], [owl:onProperty []; owl:someValuesFrom []]`	Dem@Care (23), AWO (1), OntoDT (2), SWO (1)
`?x rdfs:subClassOf ?y FILTER ( ?x != ?y )`	Dem@Care (17)
`[] rdfs:subClassOf [], [owl:onProperty []; owl:someValuesFrom ?x]. ?x rdfs:subClassOf [].`	SWO (3), OntoDT (6)
`[] rdfs:subClassOf [owl:onProperty []; owl:someValuesFrom ?x]. ?x rdfs:subClassOf ?y FILTER ( ?x != ?y )`	SWO(1), Dem@Care (7)
`?x rdfs:subClassOf ?y FILTER NOT EXISTS { ?x rdfs:subClassOf ?z . ?z rdfs:subClassOf ?y FILTER(?z != ?y && ?x != ?z) } FILTER(?x != ?y)`	Dem@Care (7)
`[] rdfs:subClassOf [ owl:onProperty []; owl:hasValue [] ]`	SWO (5)
`[] rdfs:subClassOf [], [ owl:onProperty [] ; owl:hasValue [] ]`	SWO (4)
`[] a []`	Dem@Care (3)
`[] rdfs:subClassOf ?x, [owl:onProperty []; owl:someValuesFrom ?y]. ?y rdfs:subClassOf ?x`	AWO (3)

6.6 Relationship between CQs and SPARQL-OWL queries

In this section, we analyze if certain words relate to constructs in SPARQL-OWL. Moreover, we discuss the problem of approximated queries, which are stated due to missing vocabulary, and which relate a single phrase in a CQ to a complex expression spanning over multiple ontological entities.

Signal words To understand better what is the relation between queries and CQs, we analyzed which words or phrases among CQs (further called signal words and signal phrases) co-occur with

Table 6.12: The list of signatures shared among more than one ontology. The number of occurrences of a given signature in a given ontology is provided in the column Ontologies. Reprint from [189].

Signature	Ontologies
`[] rdfs:subClassOf [], [owl:onProperty []; owl:someValuesFrom []]`	Dem@Care (23), AWO (1), OntoDT (2), SWO (1)
`[] rdfs:subClassOf [], [owl:onProperty []; owl:someValuesFrom ?x]. ?x rdfs:subClassOf [].`	SWO (3), OntoDT (6)
`[] rdfs:subClassOf [owl:onProperty []; owl:someValuesFrom ?x]. ?x rdfs:subClassOf ?y FILTER ( ?x != ?y )`	SWO (1), Dem@Care (7)
`[] rdfs:subClassOf [] ; rdfs:subClassOf [ owl:onProperty [] ; owl:someValuesFrom [ owl:onProperty [] ; owl:someValuesFrom ?x ]]. ?x rdfs:subClassOf ?y FILTER ( ?x != ?y )`	SWO (1), Stuff (1)
`[] rdfs:subClassOf [ owl:onProperty [] ; owl:someValuesFrom []]`	Dem@Care (1), SWO(1)
`[] rdfs:subClassOf [], []`	AWO (1), Stuff (1)

which queries or parts of queries. To check which fragments of CQs co-occur with fragments of SPARQL-OWL queries, we used the following procedure [189]:

1. Tokenize CQs and create a list of all anygrams (all possible sequences of n consecutive tokens, where n is between 1 and the number of tokens in a given CQ).
2. Tokenize SPARQL-OWL queries and create a list of all anygrams.
3. Create a Cartesian product between all anygrams created from CQs and those created from queries and count how many times a given pair from the product was observed in CQs and queries defined in CQ2SPARQLOWL.
4. Manually verify the list of most frequent co-occurrences in order to filter out uninteresting cases (e.g., frequent co-occurrence of words like the/a/an with common keywords like `WHERE`, `SELECT`).

Using the procedure described above, we discovered that certain words and phrases relate to sequences of tokens used among SPARQL-OWL queries. Table 6.13 shows that there are certain phrases, especially CQ starters, that can help to detect how SPARQL-OWL queries are constructed in CQ2SPARQLOWL.

Similarly, in Table 6.14 we can observe that Wh- question starters always relate to `SELECT` type of queries. In most cases the same applies to Is/Are/Can/Does, but as we already discussed in Section 6.2.2, questions starting with these words can be used with SELECT queries if they contain a list of alternatives to be chosen from (e.g., Is carbonara a pizza or pasta?). Interestingly, words like or, and, do rarely imply the choice of `owl:unionOf` and `owl:intersectionOf`.

Approximated queries In most cases, a single phrase mentioned in a CQ maps to a single IRI. However, in CQ2SPARQLOWL, there are 32 examples where a single phrase maps to a more complex construct due to missing vocabulary.

TABLE 6.13: The list of frequent signal phrases mapped to frequent SPARQL-OWL queries they are observed with. Each `:IRI` in the table may represent a different IRI. Reprint from [189].

Signal	Corresponding SPARQL-OWL	Co-occurrences
What are the possible types ...	`SELECT DISTINCT * WHERE { ?x rdfs:subClassOf :IRI . FILTER(?x != :IRI && ?x != owl:Nothing) }`	3/3 (100%)
What are the types of ...	`SELECT DISTINCT * WHERE { ?x rdfs:subClassOf :IRI . FILTER(?x != :IRI && ?x != owl:Nothing) }`	3/4 (75%)
What types of ... is/are ...	`SELECT DISTINCT * WHERE { [] rdfs:subClassOf :IRI, [ owl:onProperty ?x; owl:someValuesFrom [] ]. }`	8/11 (72.3%)
Which/what kind of ... is/are ...	`SELECT DISTINCT * WHERE { :IRI rdfs:subClassOf ?x . ?x rdfs:subClassOf :IRI. FILTER(?x != :IRI && ?x != :IRI) }`	2/3 (66.7%)
What are the main types of ...	`SELECT DISTINCT * WHERE { ?x rdfs:subClassOf :IRI. FILTER NOT EXISTS { ?x rdfs:subClassOf ?y . ?y rdfs:subClassOf :IRI. } FILTER(?x != :IRI && ?x != owl:Nothing) }`	6/9 (66%)
exactly NUMBER ENTITY	`owl:cardinality "NUMBER"^^xsd:nonNegativeInteger`	1/1 (100%)

TABLE 6.14: The list of words and parts of SPARQL-OWL queries they co-occur with. Reprint from [189]. .

Signal	Corresponding SPARQL-OWL	Co-occurences
Which/What/Who/Where/When – at the beginning of CQ	SELECT type query	107/107 (100%)
Is/Are/Can/Does – at the beginning of CQ	ASK type query	16/18 (88.9%)
or – used as part of CQ	`owl:unionOf` – present in SPARQL-OWL	2/9 (22.2%)
and – used as part of CQ	`owl:intersectionOf` – present in SPARQL-OWL	2/11 (18.2%)

Consider the following CQ: SWO_76: Is there a publication with [it]?. It is translated into the following query:

```
ASK WHERE { $PPx1$ rdfs:subClassOf [ rdf:type owl:Restriction ;
                owl:onProperty swo:SWO_0000043 ;
                owl:hasValue ?doc ].
filter(STRSTARTS(?doc, "http://dx.doi.org/")) }
```

, which can be verbalized as: Is there a documentation (`swo:SWO_0000043`) assigned for a given software (`$PPx1$`), where the IRI of the documentation starts with "http://dx.doi.org"?. In the query, the sequence `$PPx1$` relates to a placeholder [it]. If the placeholder in the CQ is materialized using the entity label l, the IRI of l should be used in place of `$PPx1$`.

In this example, the missing class representing a publication is approximated using the notion of documentation IRI, which is used in the range of the object property `swo:SWO_0000043`. We expect the IRI to point to some DOI, as scientific publications have commonly DOIs assigned.

Another interesting example is the CQ SWO_02: What are the alternatives to [this software]?. It is expressed using the following query:

```
SELECT ?sw2 WHERE {$PPx1$ rdfs:subClassOf swo:SWO_0000001 ,
                     [
                        a owl:Restriction ;
                        owl:onProperty efo-swo:SWO_0000740;
                        owl:someValuesFrom ?alg
                     ] .
?sw2 rdfs:subClassOf swo:SWO_0000001 ,
                     [
                        a owl:Restriction ;
                        owl:onProperty efo-swoSWO_0000740;
                        owl:someValuesFrom ?alg
                     ] .
?alg rdfs:subClassOf obo:IAO_0000064 .
filter(?alg != obo:IAO_0000064) .
filter(?sw1 != ?sw2). }
```

, which can be verablized as What other software (`swo:SWO_0000001`) implements (`efo-swo:SWO_0000740`) the same algorithm (`obo:IAO_0000064`)?. In this example, the missing notion of an alternative software is understood as another software implementing the same algorithm.

6.7 Relationship between CQ patterns and query signatures

In this section, we relate CQ patterns to query signatures to understand if similarly constructed CQs are similarly expressed as queries.

CQ patterns and query signatures Out of 106 CQ patterns detected in CQ2SPARQLOWL only 63 map to some SPARQL-OWL query signature. The lack of signatures for 43 CQ patterns is a consequence of inability to provide SPARQL-OWL queries for 103 out of 234 CQs in CQ2SPARQLOWL. Most of CQ patterns that map to signatures, map to a single signature (50 CQ patterns) and 13 CQ patterns map to more than one signature. It means that 13 CQ patterns are expressed using more than one query form. The CQ pattern Is EC1 EC2? is mapped to 4 (the highest number in the dataset) distinct query signatures [189].

Performing the same analysis of the mapping between higher-level CQ patterns and SPARQL-OWL signatures, we see that from 81 higher-level CQ patterns: 31 of them map to a single query signature, 11 map to more than one signature and 34 patterns have no mapping defined. The higher-level CQ pattern What is EC1? is the most diverse regarding the number of possible signatures used to express the pattern being mapped to 8 of them.

All 46 signatures generated from CQ2SPARQLOWL map to at least one CQ pattern. 23 of them map to a single CQ pattern and the remaining 23 of them map to at least two distinct CQ patterns. The signature that is connected to the highest number of CQ patterns is:

```
SELECT * WHERE {
    ?placeholder_PPx1 rdfs:subClassOf _:b0 .
    _:b0 owl:onProperty _:b1 ;
    owl:someValuesFrom _:b2 .
    ?placeholder_PPx1 rdfs:subClassOf _:b3
}
```

This signature is mapped to the following CQ patterns:

- Which EC1 PC1 EC2?
- What EC1 is of EC2 regarding EC3 and EC4?
- What EC1 are of EC2 with respect to EC3?
- What EC1 is of EC2 regarding EC3?
- What EC1 PC1 EC2?
- What types of EC1 are EC2?
- What types of EC1 PC1 EC2?
- PC1 EC1 PC1 EC2?
- What is EC1 of EC2 for EC3?

Considering higher-level CQ patterns, every signature has at least one one higher-level CQ pattern: 25 have exactly one pattern assigned and 21 have more patterns assigned.

The same signature as in the context of CQ patterns is also related to the highest number of higher-level CQ patterns. There are 9 distinct higher-level CQ patterns related to it:

- What EC1 is of EC2 regarding EC3 and EC4?
- What EC1 is of EC2 with respect to EC3?
- What EC1 is of EC2 regarding EC3?
- What is the main type of EC1?
- What type of EC1 PC1 EC2?
- What type of EC1 is EC2?
- What EC1 PC1 EC2?
- PC1 EC1 PC1 EC2?
- What is EC1?

6.8 Summary

In this chapter, we introduced CQ2SPARQLOWL, the first dataset of CQs assigned with SPARQL-OWL queries. We showed that CQs share their forms among questions stated for a single ontology and among different ontologies. In total, among the 234 CQs, we identified 106 domain-agnostic CQ patterns and 81 domain-agnostic higher-level CQ patterns.

We also analyzed the SPARQL-OWL queries stated for 131 CQs. Using a procedure generating ontology-agnostic SPARQL-OWL signatures, we identified 46 of them in the dataset.

We also analyzed the relation between CQs and SPARQL-OWL queries. Considering the CQ patterns and SPARQL-OWL signatures, we observed that there is a many-to-many relation between them as a single CQ pattern can be expressed using various SPARQL-OWL queries depending on how the knowledge is modeled, and a single SPARQL-OWL query can be expressed using various question forms.

We made the dataset public on GitHub[4] under the Creative Commons Attribution 3.0 Unported License.

[4]`https://github.com/CQ2SPARQLOWL/Dataset/`

Chapter 7

Automatic glossary of terms extraction

CQs are used in many ontology development methodologies as a source of the vocabulary engineers need to model. For this purpose, engineers collect CQs to outline the requirements and then decide which classes, instances, and properties they will model in the ontology to make it competent to answer the gathered questions. For example, in the NeON methodology [167], phrases extracted from CQs are listed in the glossary of terms included in the Ontology Requirements Specification Document.

However, in the case of large ontologies, based on multiple CQs stated by groups of experts, the automatization of this task could speed up the ontology authoring. Tools detecting glossaries of terms can also fuel automatic translators of CQs into queries as they can indicate domain phrases in CQs related to the vocabulary in an ontology.

Based on the insights gained in Section 6.3.2, where we introduced the first – requiring human supervision – heuristics for detecting domain phrases for domain-agnostic CQ patterns construction, we propose two methods that do not require human intervention: a machine learning-based tagger and a manually handcrafted rules-based approach.

The machine learning-based method proposed in this chapter was published as a conference poster [186], and the rule-based approach is accepted for publication in *Foundations of Computing and Decision Sciences.*

The remainder of this chapter is structured as follows: In Section 7.1, we introduce the datasets used to prepare and evaluate each of the methods. In Sections 7.2 and 7.3, we describe both methods in detail. We evaluate both approaches in Section 7.4, discuss the results and errors in Section 7.5, and conclude in Section 7.6.

7.1 Materials

We use the two largest available datasets of ontological requirements to construct the following disjoint sets:

- Training set – used to train the machine learning-based model and handcraft rules for the rule-based method.
- Evaluation set – used to measure the quality of both approaches and choose the better quality one.

In this section, we describe how we construct each of these sets.

7.1.1 Training set

CQ2SPARQLOWL, introduced and analyzed in Chapter 6, was used to generate domain-agnostic CQ pattern candidates in a semi-automatic manner. It contains 234 CQs, transformed into pattern candidates by replacing domain phrases with EC and PC markers. These domain phrases represent vocabulary to model in an ontology and can be used to populate the glossary of terms.

For this reason, we can use CQs and their CQ pattern candidates to identify domain-related phrases. As shown in the upper box presented in Figure 7.1, we can make the alignment between the CQ and the CQ pattern candidate to achieve it. Considering the example given in Figure 7.1, the following relation between chunks is observed:

- EC1 – the placeholder [it]
- EC2 – the word FOSS

We iterated over each of the 234 CQs and marked which phrases relate to chunks. In the case of 116 CQs from CQ2SPARQLOWL, one or more chunks relate to placeholders that should be replaced with appropriate strings to produce multiple similar CQs. As CQ2SPARQLOWL provides ontologies related to CQs, we can replace placeholders with actual labels from a given ontology (e.g., [it] in Is [it] FOSS? should be replaced with various pieces of software modeled in SWO). To materialize CQs, we applied the following procedure:

For a given CQ *cq* and an ontology $\mathbb{O}$:

1. Identify the placeholder *ph* in *cq*.
2. Extract all labels L from $\mathbb{O}$ that are possible materializations of *ph*.
3. For each label l in L, generate a new CQ by replacing the placeholder *ph* with l.

In SWO and its CQs, most placeholders refer to pieces of software. This ontology defines a rich taxonomy of software types, providing 537 names of software. We noticed that some CQs regarding SWO introduce more than a single placeholder (e.g., the SWO_22: Can [software A] work with data that are output from [software B]?). In all such CQs, to prevent combinatorial explosion (where, e.g., we materialize CQ SWO_22 with all possible pairs of software, generating $537^2 = 288,369$ materializations of this single CQ), we use only a single random label to materialize each placeholder.

The materialization of CQs in CQ2SPARQLOWL increased the number of questions from 234 to 47,313. As each placeholder replacement was extracted from an appropriate ontology, we automatically marked those labels as domain-related fragments. Moreover, each placeholder in CQ2SPARQLOWL represents a class or an instance, so we automatically labeled each replacement as an entity chunk. This way, we transformed the initial set of 234 CQs manually marked with domain-related vocabulary candidates (via the analysis of the alignment between the CQs and the CQ patterns) into 47,313 materialized examples. We visualize the materialization process in the lower box presented in Figure 7.1. We also provide detailed information on the number of materialized CQs generated per ontology in Table 7.1.

All 47,313 materialized CQs marked with domain-related vocabulary were serialized using the IOB format in the following process:

1. Tokenize each CQ, i.e., split it into a sequence of tokens.
2. Tag each token that is not marked as domain-related with the O tag. This tag informs that a given token does not represent any class, instance, or property.

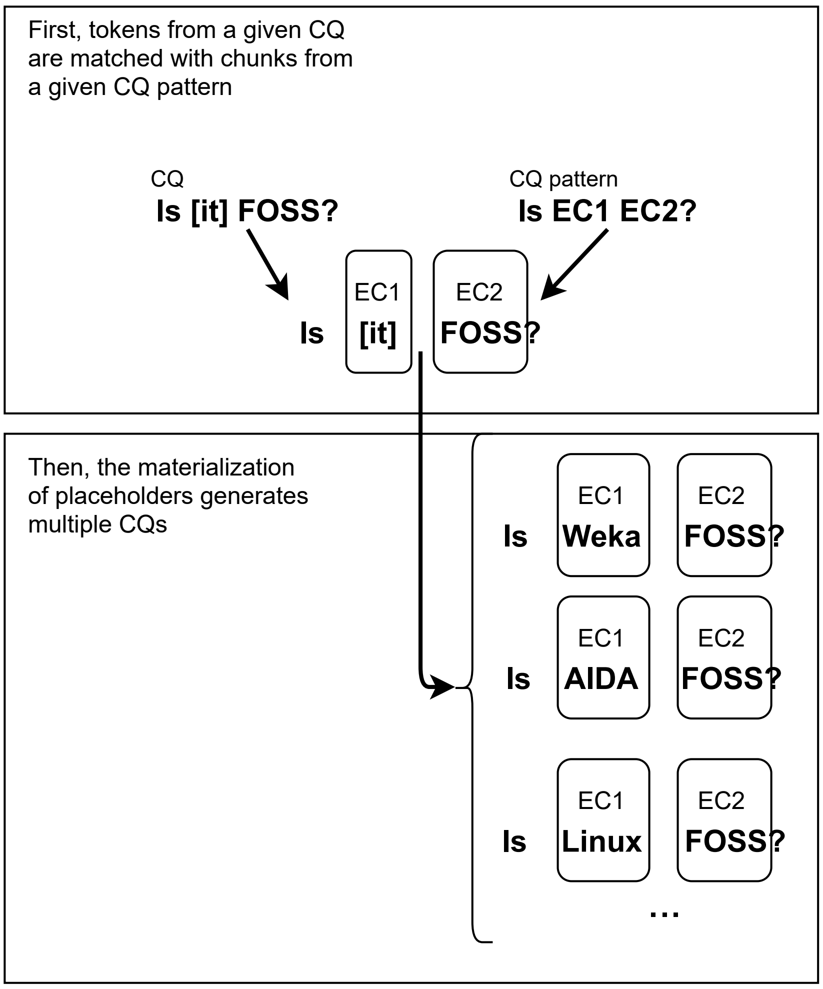

First, tokens from a given CQ are matched with chunks from a given CQ pattern
CQ
Is [it] FOSS?
CQ pattern
Is EC1 EC2?
Is
EC1 [it]
EC2 FOSS?
Then, the materialization of placeholders generates multiple CQs
Is EC1 Weka EC2 FOSS?
Is EC1 AIDA EC2 FOSS?
Is EC1 Linux EC2 FOSS?
...

FIGURE 7.1: The process of marking phrases with chunks followed by the materialization of placeholders to generate multiple training examples from CQ2SPARQLOWL automatically.

TABLE 7.1: The number of materialized CQs per ontology in CQ2SPARQLOWL.

Ontology	# of CQs before materialization	# of CQs after materialization
SWO	88	46,465
Dem@Care	107	107
OntoDT	14	482
AWO	14	47
Stuff	11	90

Table 7.2: Number of labels in materialized CQ2SPARQLOWL. Column **Label** represents a given label, column **Occurrences** couns how many times a given label occurs, and column **Relative occurrences** summarizes what percentage of tokens in CQ2SPARQLOWL are tagged with a given label.

Label	Occurences	Relative occurences
O	188,438	42.48%
B-E (Begin entity)	86,639	19.53%
I-E (Inside entity)	115,953	26.14%
B-R (Begin relation)	29,844	6.72%
I-R (Inside relation)	22,697	5.12%

3. For each sequence of tokens that is marked as an entity chunk (EC):
   - Mark the first token in the sequence with the `B-E` tag, which indicates the beginning of the entity (a class or an instance candidate).
   - Mark the remaining tokens in the sequence with the `I-E` tag, which indicates continuation of the entity started with `B-E`.
4. For each sequence of tokens that is marked as a predicate chunk (PC):
   - Mark the first token in the sequence with the `B-R` tag, which indicates the beginning of the relation (a property candidate) the PC represents.
   - Mark the remaining tokens in the sequence with the `I-R` tag, which represents continuation of the relation started with `B-R`.

The differentiation between `B-` and `I-` is needed to separate two entities if they are located next to each other, as presented in Figure 7.2.

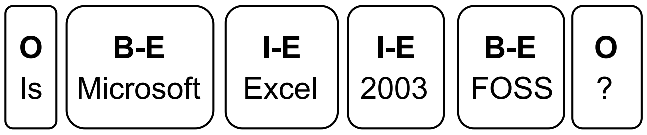


Figure 7.2: Example CQ tokenized and tagged with IOB tags. Here, Microsoft Excel 2003 is adjacent to FOSS.

With the use of IOB encoding, the automatic glossary of terms extraction becomes a sequence tagging problem, where each token t_i in some tokenized document d is mapped onto one of the labels from the label set using a mapping function $m(\cdot)$:

$$\forall t_i \in d : m(t_i) \in \{\text{O}, \text{B-E}, \text{I-E}, \text{B-R}, \text{I-R}\}$$

Using this procedure, we generated 47,313 CQs consisting of 443,571 tokens in total. The CQs are tagged with 86,639 candidates for classes or instances (labeled with IOB as entities) and 29,844 for properties (labeled with IOB as relations). The summary providing the numbers of the IOB labels assigned to tokens can be found in Table 7.2. This dataset serves as the *training set* for the machine learning-based tagger and a set providing examples for the rule-based tagger.

7.1.2 Evaluation set

To evaluate the methods, we searched for a dataset providing requirements stated for ontologies different than those included in CQ2SPARQLOWL. We chose the largest collection of ontology

Table 7.3: Evaluation set extracted from CORAL.

Ontology name	Number of questions	Number of statements
VICINITY Core [109]	58	68
VICINITY WOT [109]	7	17
VICINITY WOT mappings [109]	0	15
BTN100 [128]	77	0
SAREF [46]	1	70
SAREF4BLDG [46]	97	1
SAREF4ENVI [46]	18	40
OneM2M [46]	0	58
ODRL [166]	0	12
BTO [105]	0	18
Total	258	299

Table 7.4: The summary of the gold standard phrases tagged manually in the *evaluation set* by an expert.

Questions or Statements	Number of entities	Number of relations
Questions	472	89
Statements	892	327

requirements called CORAL [49], containing requirements expressed as CQs and statements. From this dataset, we discarded those requirements that are part of CQ2SPARQLOWL, so from the initial number of 834 requirements, 557 formed our *evaluation set*. Although the *training set* consists of CQs only and the *evaluation set* contains both CQs and declarative statements, we decided to use both kinds of requirements to check if any of the methods designed to work on CQs generalizes to statements. In total, our *evaluation set* consists of 258 CQs and 299 statements. The summary outlining the ontologies used in the *evaluation set* and the numbers of questions and statements per each ontology is provided in Table 7.3.

For each requirement in the *evaluation set*, we manually marked the phrases that are candidates for ontological entities to be included in the glossary of terms. Similar to the *training set*, we split those candidates into two categories: entities - representing candidates for classes or instances, and relations representing candidates for properties.

We marked 1,780 glossary of terms candidates in the *evaluation set* and published this data online[1],[2]. Since different ontologies may use different modeling styles (e.g., whether to use a class, an instance, or a property to represent a verb), the glossary of terms candidates were constructed independently of the actual ontology vocabulary, e.g., all domain-related noun phrases are marked as entities and phrases representing actions as relations. This approach ensures consistency among all examples.

In Table 7.4, we summarize the number of candidates proposed for questions and statements as well as entities and relations.

7.2 Machine learning-based tagger

The first tagger we propose is a model based on Conditional Random Fields (CRFs) [91]. We used *python-crfsuite*[3], a Python wrapper using CRFSuite [126] as the implementation of CRFs.

[1]https://github.com/reqtagger/ReqTagger/blob/master/evaluation_cqs.json
[2]https://github.com/reqtagger/ReqTagger/blob/master/evaluation_statements.json
[3]https://pypi.org/project/python-crfsuite/

To choose the list of features describing each token in a CQ and select the values of hyperparameters, which are parameters that influence the learning process, we extracted a holdout set from the *training set* to form two disjoint subsets:

- *SWO set* – consisting of CQs stated for the SWO ontology (46,465 CQs in total).
- *validation set* – consisting of CQs stated for other ontologies (107 stated for Dem@Care, 482 for OntoDT, 47 stated for AWO, 90 stated for Stuff – 726 CQs in total).

Using such a split, we ensure that there are different terms in the *validation set* than in the *SWO set* so that the *validation set* can be used to verify how well the model works on unseen data.

First, we used the default hyperparameter values:

- C1 = 0.001
- C2 = 0.001
- max iterations = 10
- training method = L-BFGS

to handcraft the list of features that maximizes the F_1 score of the model trained on the *SWO set* and evaluated on the *validation set.* Here, C1 and C2 correspond to the λ_1 and λ_2 parameters of the *Elastic Net* regularization, respectively. The max iteration parameter determines the maximum number of iterations of the L-BFGS optimization algorithm.

In consequence, we selected the following list of features to describe each token:

- The lowercased token.
- A flag (set to 0 or 1) indicating whether the token is an auxiliary verb (e.g., does, have).
- The sequence of the token's last 3 characters.
- A flag (0 or 1) that is set to 1 if and only if the token starts with a capital letter.
- A flag (0 or 1) that is set to 1 if and only if the token contains more than one capital letter.
- A number that represents how many tokens in the CQ are verbs.
- A flag (0 or 1) that is set to 1 if and only if the token is a number.
- Part-of-speech (POS) tag from the Universal POS tag set assigned to the token.
- The label assigned to the token in a dependency parse tree constructed over the CQ.
- The label assigned to the token's head in a dependency parse tree constructed over the CQ.

To pass more context to the model, we enrich the current token description with features describing 2 previous and 2 next tokens (each of those context tokens is described using the following features from the list above [186]: the lowercased token, a flag indicating if the token starts with a capital letter, a flag indicating if the token contains more than one capital letter, POS tag, dependency tree label of the token, dependency tree label of the token's token).

Then, we used grid search to optimize the values of the hyperparameters. We verified the following values:

1. C1: [0.001, 0.01, 0.1, 1, 10]

2. C2: [0.001, 0.01, 0.1, 1, 10]
3. max iterations: [10, 100, 1,000]

Grid search evaluates each combination of hyperparameter values and chooses the one that maximizes the F_1 score calculated on the *validation set*.

We found the following values lead to the highest F_1 score:

- C1 = 1
- C2 = 0.001
- max iterations = 100
- training method = L-BFGS

We published the tagger implementation and the best model online[4]. The tagger splits a given CQ into tokens and calculates features for each token. Then it produces the output, which is a sequence of IOB labels assigned to the tokens.

7.3 ReqTagger: a rule-based tagger

The second tagger we propose is an evolution of the Algorithm 3 presented in Appendix A.1 that is used to extract CQ patterns from CQs. Contrary to the algorithm from Appendix A.1, our solution, hereafter referenced to as ReqTagger, can be used without human supervision. Figure 7.3 visualizes the workflow of ReqTagger, which can be summarized in three steps:

1. Pass the requirement to a rule-based extractor to identify which fragments of text represent candidates for properties or classes/instances.
2. If overlapping spans of texts are extracted, merge them.
3. Filter out phrases that do not represent candidates for classes, instances, or properties.

ReqTagger produces two output lists: candidates for properties and classes/instances separately. In the following subsections, we describe each processing step in detail.

7.3.1 Rule-based extractor

ReqTagger is based on a predefined set of rules that are expressed in our own rule language. First, it tokenizes the input text and assigns each token its POS tag. Then, it uses rules to match sequences of POS tags and passes the sequences of tokens related to them to subsequent processing steps.

Let us define the rule language using Extended Backus-Naur Form [184]:

```
<rule> ::= <item>, {" ", <item>};
<item> ::= [ <modifier> ], <POStags>;
<modifier> ::= "{0+}" | "{1+}" | "{1?}";
<POStags> ::= <POS> , {"|", <POS>};
```

[4]https://github.com/dwisniewski/CRFBasedGlossaryOfTermsExtraction

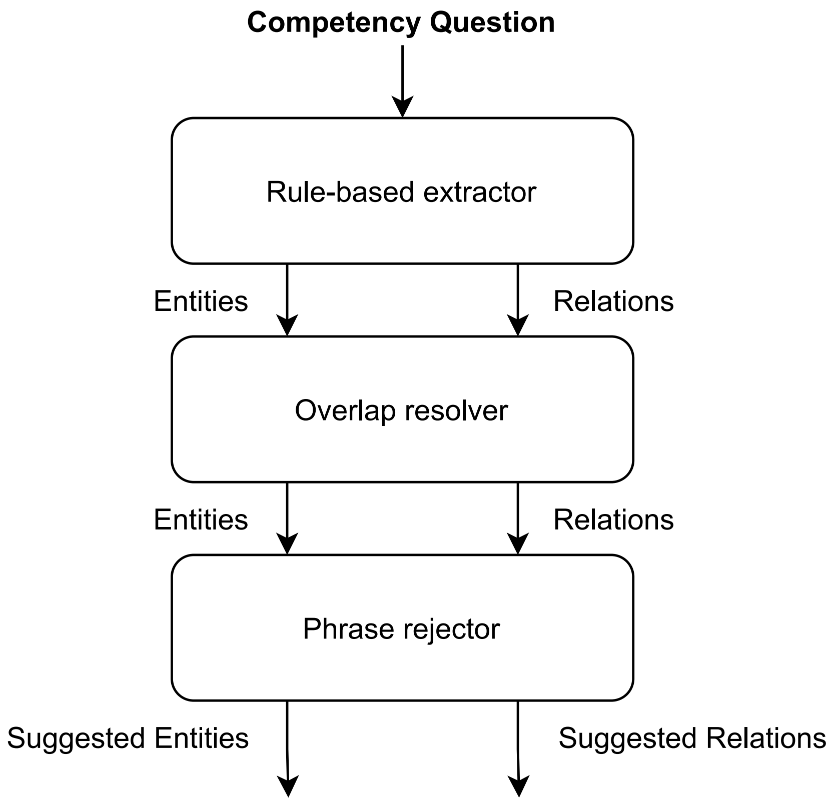


FIGURE 7.3: The workflow of REQTAGGER. Suggested entities relate to classes and instances, while suggested relations relate to data and object properties.

Each rule (`<rule>`) consists of a sequence of items (`<item>`) separated by a single space. Each item consists of a set of |-separated POS tag (`<POStags>`) alternatives that can be preceded by an optional modifier (`<modifier>`) specifying how many tokens can be matched by this item. We support the following modifiers:

- {`0+`} – we expect a sequence of at least zero tokens matching any of the POS tags in a given POS tag set.
- {`1+`} – we expect a sequence of at least one token matching any of the POS tags in a given POS tag set.
- {`1?`} – we expect at most one token matching any of the POS tags in a given POS tag set.

`<POS>` is a non-terminal symbol that represents a POS tag. spaCy[5], one of the most popular NLP libraries, uses two POS tag sets:

- Universal POS tags: a coarse-grained tag set proposed in Universal Dependencies[6]. This tag set provides 17 distinct tags.
- OntoNotes (version 5) of the Penn Treebank POS tags (further referenced as OntoNotes POS tags): a fine-grained tag set providing 53 tags[7].

To verify which tag set is better for the glossary of terms extraction, in REQTAGGER, we support both tag sets. An example of a rule using Universal POS tags is:

[5]`https://spacy.io`
[6]`https://universaldependencies.org/u/pos/`
[7]`https://spacy.io/api/annotation#pos-en`

`{0+}VERB {1+}NOUN|PROPN ADJ`

The rule can be interpreted as: search for a sequence of at least zero verbs, followed by at least one token that represents either a noun or a proper noun, followed by exactly one adjective.

Since modifiers: `{0+}` and `{1+}` may produce many possible matches (e.g., the rule `{1+}NOUN` applied to a sequence of 2 nouns may match the first noun only, the second one only, or the sequence of both), we decided that they should work greedily – matching only the longest possible sequence. Internally, each rule is transformed into a regular expression[8] so that the rule language is a more human-friendly way to state regexes. Then, these regexes are translated into nondeterministic state automata to find matches. As we support two part-of-speech tag sets, separate sets of rules are prepared to support Universal POS tags and OntoNotes POS tags. Moreover, because we suggest candidates for classes/individuals (called entities) and properties (called relations) independently, we defined separate sets of rules for entities and relations.

In Table 7.5, we listed all rules expressed using Universal POS tags, while in Table 7.6 all rules expressed using OntoNotes POS tags. The description of each POS tag used is provided in Table 7.7 for Universal POS tags and in Table 7.8 for OntoNotes POS tags.

These rules were constructed manually on a subset of the *training set*. We selected all CQs that did not introduce placeholders and a single random materialization from those CQs that initially contained placeholders. As a result, the rules were handcrafted on a sample of 234 materialized CQs.

Table 7.5: Rules for entities and relations extraction based on Universal POS tags. The identifiers are described in Table 7.7.

Entities extraction rules
1. `{0+}ADJ {1+}NOUN|PROPN`
2. `{0+}ADJ {1+}NOUN|PROPN {0+}ADJ {1+}NOUN|PROPN`
3. `VERB NOUN|PROPN`
Relations extraction rules
1. `{0+}PART|VERB|AUX VERB`
2. `{0+}VERB ADV|AUX`
3. `{0+}PART|VERB|AUX {1+}AUX|VERB|ADJ|ADV ADP|SCONJ`

7.3.2 Overlap resolver

With multiple rules defined (e.g., 10 rules based on OntoNotes POS tags to extract relations), a pair of rules may mark the same sequence of tokens or mark sequences, one of which is the substring of the other. We use the overlap resolver to identify such cases and reject:

- The shorter sequence, if a longer match is found.
- One sequence, if both matches are identical (i.e., two rules mark the same sequence).

The main goal of the overlap resolver is to remove duplicated candidates from the extracted phrases.

7.3.3 Phrase rejector

We found that some matched phrases have special meanings and do not represent ontology vocabulary. For example, nouns such as kind, category, type most likely indicate the subclass relation

[8]As defined in `https://github.com/reqtagger/ReqTagger/blob/master/reqtagger/reqtagger.py` in the `parse_rule()` function.

Table 7.6: Rules for entities and relations extraction based on OntoNotes POS tags. The identifiers are described in Table 7.8.

Entities extraction rules
1. `{1?}DT {0+}JJ|JJS|FW|NN|NNS|NNP NN|NNP|NNS`
2. `DT VB|VBG|VBD|VBZ|VBN {0+}NN|NNS|NNP|JJ NN|NNS|NNP`
Relations extraction rules
1. `{1+}JJ IN`
2. `JJR IN`
3. `RB VBD`
4. `{0+}MD {1+}VBZ|VBN TO VB`
5. `{0+}MD VB VBN IN`
6. `{1?}TO VB|VBN|VBZ|VBP {1?}JJ IN`
7. `{0+}VB|MD {0+}TO VB`
8. `{1?}TO {0+}VB|VBP|VBZ|VBN|VBD|VBG {1+}RBS|VB|VBZ|VBN|VBD|VBG|IN RBS|IN`
9. `{1?}TO VB|VBZ|VBP|VBG|VBD VB|VBN|VBG|VBG|VBD|JJR|RP`
10. `VBN VBG`

Table 7.7: Descriptions of identifiers used in rules based on Universal POS tags.

Identifier	Description	Example
ADJ	adjective	blue
NOUN	noun	chair
PROPN	proper noun	Mike
VERB	verb	reading
PART	particle	possesive marker ’s
AUX	auxiliary verb	should
ADV	adverb	up
ADP	adposition	during
SCONJ	subordinating conjunction	through

Table 7.8: Descriptions of identifiers used in rules based on OntoNotes POS tags.

Identifier	Description	Example
VB	verb, base form	be
VBZ	verb, 3rd person, singular, present	is
VBN	verb, past participle	been
VBG	verb, gerund/present participle	being
VBP	verb, singular, present, non-3d	are
RP	particle	give up
RB	adverb	good
RBS	adverb, superlative	best
DT	determiner	the
JJ	adjective	blue
JJS	adjective, superlative	tallest
JJR	adjective, comparative	taller
NN	noun, singular or mass	chair
NNS	noun plural	chairs
NNP	proper noun	Mike
IN	preposition, subordinating conjunction	of
MD	modal	will

rather than resources with those labels (e.g., in the CQ: What kinds of cars are produced in Poland?). We handcrafted a list of rejected phrases based on the *training set* and our expertise. The phrase rejector analyzes each extracted candidate, and if it is found on the list, it is not presented to the user. We provide the rejected phrases as two lists regarding entities and relations in Appendix C.

7.4 Evaluation

In the *evaluation set*, an expert marked zero or more candidates for entities and relations in each requirement. We call these *gold standard annotations.* Each of our taggers also marks phrases (sequences of tokens) that it considers as candidates. Those we call *tagger predictions.* To measure the quality of each tagger, we compare the *gold standard annotations* with *tagger predictions* for entities and relations separately.

To introduce the metrics used to evaluate taggers, first, let us define what we consider a correctly extracted phrase. If a given phrase extracted by a tagger covers exactly the same tokens as one of the *gold standard annotations* and the extracted phrase represents the same type (either entity or relation candidate) as stated in the gold standard, we consider the extracted candidate as correct. We still consider the extracted phrase as correct if:

- The only difference between the extracted phrase and any annotation in the *gold standard* is the use of an article (either a, an, or the). For example, the extracted word car is equal to a gold standard annotation the car.
- The extracted phrase and the gold standard phrase both represent a relation and differ only in the presence of an auxiliary verb. For example, the extracted phrase is published by is equal to a gold standard annotation published by.

If the extracted phrase is considered correct, we call it a true positive (TP).

There can be 2 kinds of errors introduced:

- If a phrase extracted by the tagger cannot be matched with any annotation from *gold standard annotations* of a given type, we call it a false positive (FP) because we falsely mark some phrase as a candidate.
- If a phrase in the *gold standard annotations* is not extracted by the tagger or it is extracted but is assigned a different type, we call it false negative(FN), as we falsely assume an expected phrase as a non-candidate.

If the extracted phrase is a subsequence of the phrase marked in a gold standard or covers more tokens, both false positive and false negative are produced.

The true positives, false positives, and false negatives can be aggregated to introduce two measures:

- Precision P, defined as $P = \frac{TP}{TP+FP}$, which tells us what fraction of phrases marked as candidates of a given type are indeed the expected ones.
- Recall R, defined as $R = \frac{TP}{TP+FN}$, which tells us what fraction of phrases of a given type, marked in the *gold standard annotations*, is correctly extracted.

Those measures allow us to look at the errors from two perspectives. Moreover, we introduce a single number $F_1 = \frac{2PR}{P+R}$ that is a harmonic mean of P and R, to rank the methods.

We compared the following methods:

- ReqTagger using OntoNotes POS tags-based rules.
- ReqTagger using Universal POS tags-based rules.
- The method provided in Appendix A.1 with no human postprocessing (referenced as CQ2SPARQLOWL RULES + CHUNKING).
- CRF-based tagger as introduced in Section 7.2. It is trained on all materialized 46,465 CQs from the SWO ontology only.
- CRF-based tagger trained on all materialized 47,313 CQs from all ontologies included in CQ2SPARQLOWL. It uses the same hyperparameter values as the previous one.

We evaluated the methods from four perspectives. These perspectives are created by splitting the dataset to separate CQs from statements and entities from relations. As a result, we:

1. Measure the quality of extracting entities from CQs,
2. Measure the quality of extracting relations from CQs,
3. Measure the quality of extracting entities from statements,
4. Measure the quality of extracting relations from statements.

The evaluation scores calculated for entities and relations extracted by taggers from CQs are presented in Table 7.9. The scores calculated for phrases extracted from the statements are provided in Table 7.10.

In Table 7.11, we ranked the methods by a single measure, which is a macro F_1 that is calculated as an average over F_1 scores calculated for questions marked with entities, questions marked with relations, statements marked with entities, and statements marked with relations.

Table 7.9: Methods scored using precision, recall, and F_1, calculated over evaluation questions. The scores are calculated separately over entities and relations. The highest scores are marked in bold.

Setting	Scores entities			Scores relations		
	P	R	F_1	P	R	F_1
ReqTagger (OntoNotes POS tags)	**0.968**	**0.972**	**0.970**	0.835	**0.854**	**0.844**
ReqTagger (Universal POS tags)	0.958	0.964	0.961	0.773	0.843	0.806
CQ2SPARQLOWL rules + noun chunking	0.645	0.778	0.705	0.788	0.753	0.770
CRF-based tagger (SWO)	0.872	0.780	0.823	0.875	0.236	0.372
CRF-based tagger (CQ2SPARQLOWL)	0.898	0.824	0.860	**0.923**	0.270	0.417

Table 7.10: Methods scored using precision, recall, and F_1, calculated over evaluation statements. The scores are calculated separately over entities and relations. The highest scores are marked in bold.

Setting	Scores entities			Scores relations		
	P	R	F_1	P	R	F_1
ReqTagger (OntoNotes POS tags)	**0.891**	**0.888**	**0.889**	**0.749**	**0.869**	**0.805**
ReqTagger (Universal POS tags)	0.860	0.861	0.861	0.718	0.865	0.785
CQ2SPARQLOWL rules + chunking	0.837	0.830	0.833	0.722	0.706	0.714
CRF-based tagger (SWO)	0.544	0.355	0.43	0.678	0.180	0.285
CRF-based tagger (CQ2SPARQLOWL)	0.604	0.401	0.482	0.687	0.174	0.278

7.5 Discussion

Based on the evaluation scores, in this chapter, we compare the methods proposed and discuss the errors made by the best solution.

Table 7.11: Aggregated F_1 scores. Each score is the arithmetic mean of F_1 scores calculated for entities extracted from questions, entities extracted from statements, relations extracted from questions, and relations extracted from statements.

Method	Macro-averaged F_1 score
ReqTagger (OntoNotes POS tags)	**0.877**
ReqTagger (Universal POS tags)	0.853
CQ2SPARQLOWL rules + chunking	0.756
CRF-based tagger on CQ2SPARQLOWL	0.509
CRF-based tagger on SWO	0.478

7.5.1 Methods comparison

To better understand the results presented in Tables 7.9, 7.10, and 7.11, we provide three perspectives:

Global perspective Considering all categories (entities, relations, questions, and statements) jointly, in Table 7.11, we observe a strong dominance of ReqTagger. Considering the model using OntoNotes POS tags, the results gathered in Tables 7.9 and 7.10 prove that it outperforms all competitors in 11 out of 12 scenarios. The only scenario where it scores lower than any other competitor is the task of relation extraction from questions. However, although the CRF-based tagger outperforms ReqTagger in terms of precision, it has a very low recall.

ReqTagger with OntoNotes POS tags consistently scores higher than with Universal POS tags. We suppose that it is due to the much larger tag set provided in OntoNotes POS tags, which allows to state more precise rules.

Extracting phrases from questions and statements We see that each of the considered methods scores higher on questions than on statements. We expected such behavior, as the methods were prepared on the *training set*, which contains only CQs. However, it is interesting that the rule-based methods (ReqTagger and CQ2SPARQLOWL rules + chunking) are scored much better than both CRF-based taggers. We hypothesize that the low scores of the CRF-based taggers measured on statements are due to the information on neighboring tokens it uses during training. As the grammatical constructions are different between questions and statements, they form different contexts that may misguide the CRF-based taggers.

Extracting entities and relations candidates As we can see in Tables 7.9 and 7.10, it is much easier to extract candidates for entities than for relations. We noticed that in the *training set* most noun phrases represent candidates for entities, and these are easy to identify.

However, the relations are more vague as they can be represented in various forms (such as verbs, adjectives, or even nouns). The analysis of errors made on the *evaluation set* shows that some rejected phrases should be verified (e.g., we considered is in as the sequence to reject, but some requirements from the *evaluation set* support another decision – to make it a candidate for a relation).

An interesting observation is that CQ2SPARQLOWL rules + chunking scores very low on the entity extraction from questions task. This situation is caused by the way spaCy extracts noun phrases. spaCy builds a dependency parse tree for each input sequence and marks as a noun phrase every subtree, the head of which is a noun. These are then suggested as candidates for classes or instances. However, as can be seen in Figure 7.4 in questions starting with a Wh- word (what, which, why, who, whom, whose, where, when) followed by a noun, this noun is a head of the wh- word so that the whole Which software is extracted as an entity suggestion.

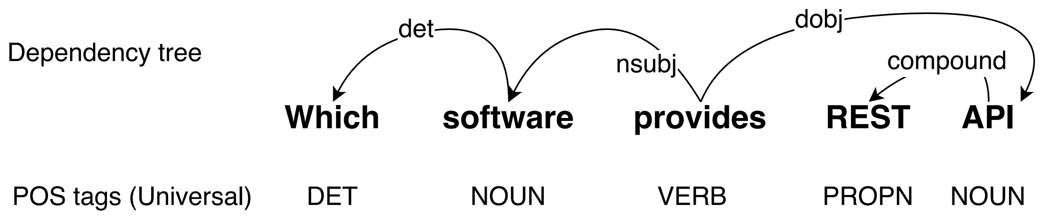


FIGURE 7.4: The dependency parse tree constructed over a sample CQ.

7.5.2 Error Analysis

As can be seen in Table 7.11, ReqTagger with OntoNotes POS tags achieved the best evaluation scores. However, the method is not perfect, so in this section, we discuss the errors generated by ReqTagger with OntoNotes POS tags in the evaluation process. The list of errors is split into two subsections: describing those related to entity suggestions and those related to relation suggestions.

Candidates for entities

Wrong POS tags In a CQ: Which properties does a weight scale affect?, the word affect is tagged by spaCy as a noun instead of a verb. For this reason, the whole phrase a weight scale affect is suggested as a single entity candidate.

An adjective *many* before a noun Two CQs from the *evaluation set*: How many organizations can have a partnership? and How many rivers flow into the sea or ocean X? contain sequences, in which the adjective many is followed by a noun (many organizations, many rivers). In both cases, ReqTagger wrongly attaches the word many as a word that should start the entity name. Since many is an adjective (tagged as `JJ`), from the perspective of ReqTagger, it is indistinguishable from other situations where adjectives should be attached to glossary of terms entries (e.g., African wildlife, Red Cross).

Gerund followed by a noun The CQ: Which are the FFCC stations that are stopping places and which are the FFCC stations that have passenger traffic? contains a phrase stopping places, which, from the perspective of POS tags, is a sequence of a verb ending with the -ing suffix (gerund) followed by a noun. However, ReqTagger suggests a single word places instead. The problem cannot be easily solved. Adding a rule with the gerund (`VBG` tag) followed by a noun would generate many false-positive errors. For example, in a CQ: What is keeping John from promotion?, the keeping John would be extracted as an entity candidate.

Candidates for relations

Wrong POS tags When processing one of the test requirements: IoT gateways expose endpoints., the POS tagger makes wrong decisions assigning a verb (VBZ) to a noun gateways. As a result, ReqTagger thinks that gateways expose is a sequence of two verbs and suggests it as a single relation name instead of the expected expose verb.

Verb in the past tense followed by a noun In the CQ What is a cooled beam?, ReqTagger extracts the token cooled and suggests it as a candidate for a relation, while it should be a part of the a cooled beam entity suggestion. If there is a determiner before the verb, a new rule could solve that type of problem. However, we observed that frequently no determiner precedes such verbs. In such cases adding a new rule matching a verb in the past followed by a noun would generate many wrong results (e.g., from the requirement I called John yesterday, the called John would be wrongly suggested as an entity).

Dropping tokens that should not be dropped In the CQ: Which historic places are in a county? the sequence are in can be a relation to be modeled in an ontology. However, the analysis of the *training set* provided arguments to discard such a phrase, so although the sequence is extracted by one of the rules, it is later rejected.

Extracting only fragments of relations When processing the CQ: Which villages does the road go through?, ReqTagger proposes the verb go as a relation candidate instead of go through. This kind of error can be fixed by adding new rules.

Splitting long relation suggestions In A mapping might be needed to be executed before another mapping., there is a phrase needed to be executed before that is marked by an expert as a relation candidate. However, ReqTagger detects 2 relations there: needed to be and to be executed before. Relations consisting of so many tokens were not observed in the CQs from the *training set*, but they are popular among statements. To fix this kind of error, we should add new rules.

7.6 Summary

In this chapter, we described how glossary of terms extraction from textual requirements can be automated. We proposed two methods: a machine learning-based one and a handcrafted rule-based one.

The rule-based method works better for several reasons:

- It is based on human expertise rather than purely on machine-based data analysis. The CRF-based model may be unaware that phrases such as kind, category are not good candidates for entities.
- It does not depend on the context words so that it scales to processing statements with good quality.
- It uses sequences of POS tags rather than noun phrases defined as in spaCy.

However, the rule-based method is not perfect for several reasons:

- The list of rejected phrases is not exhaustive. One has to maintain it and expand if needed.

- The rule language we provide can only mark consecutive sequences of tokens as candidates for entities and relations. There are cases where nonconsecutive sequences state phrases. For example, in Do you prefer white or red wine?, ReqTagger finds red wine only and fails to detect white . . . wine as a candidate.
- The method cannot decide whether an extracted phrase should be a class or an instance. Moreover, it cannot distinguish between data and object properties.

In conclusion, considering the evaluation scores, we showed that it is possible to recommend candidates for entities and relations without human supervision.

Chapter 8

BigCQ: a synthetic dataset of CQ patterns formalized into SPARQL-OWL templates

The CQ2SPARQLOWL dataset that we introduced in Chapter 6 is a good starting point to understand how engineers construct CQs and how CQs are related to the knowledge provided in ontologies. However, its small size and the multitude of modeling styles that engineers use to express the knowledge limit the generalization capacity of the dataset as it does not cover many possible question and query forms. Moreover, in Table 6.4, we show that only 6 CQ patterns presented in CQ2SPARQLOWL are shared by more than one ontology. Considering higher-level patterns, Table 6.6 lists 8 of them observed in two or more ontologies. Furthermore, in the same table, we show that the most commonly used higher-level CQ pattern is shared by only 3 out of 5 ontologies.

Such a state of affairs may indicate that human language provides engineers with rich possibilities to formulate questions. Various CQs can differ in their grammatical forms and use different words to express the same meaning. Our analysis of competency questions provided in CORAL[1] [49] shows that CQ2SPARQLOWL does not introduce some CQ forms (e.g., questions about counting, for example, How many organizations can a partnership count?).

Regarding ontology modeling styles, the 5 ontologies included in CQ2SPARQLOWL provide knowledge formalized by different engineers and describe diverse domains, but they do not cover all possible modeling decisions.

For these reasons, in this section, we provide a method for automatic construction of CQs related to SPARQL-OWL queries. The goal of this method is to generate a large set of diverse CQ forms mapped to SPARQL-OWL queries targeting the most prevalent modeling decisions.

The BigCQ dataset generated using the method introduces 549 SPARQL-OWL query templates mapped to 77,575 CQ patterns and is motivated by a set of frequent ontology axiom forms extracted from a large group of over 300 biomedical ontologies. The pairs of generated patterns and templates can be filled with labels and IRIs extracted from a given ontology to produce even larger collections of questions and queries.

BigCQ may be used to provide silver-standard datasets for tasks such as automatic translation of CQs into SPARQL-OWL queries. The BigCQ dataset and the method implementation used to generate the dataset are published online[2]. BigCQ and the method for CQ patterns and

[1]https://coralcorpus.linkeddata.es
[2]https://github.com/dwisniewski/BigCQ/

SPARQL-OWL templates generation, which is visualized in Figure 8.1, has been described in a preprint [187], the shortened form of which was accepted for presentation at the student abstract track during AAAI'22.

This chapter is structured as follows: In Section 8.1, we discuss the datasets used to motivate the method and evaluate the coverage of BigCQ. Section 8.2 covers the analysis of the frequent ontology axioms processed into a form that helps transform them into questions and queries. In Section 8.3, we describe the workflow of the method, while in Section 8.4, we describe the dataset produced using the method. The coverage of the dataset measured on a set of previously unseen CQs is calculated in Section 8.5. We conclude, describe potential applications, and provide examples of generated questions and queries in Section 8.6.

8.1 Materials

There are four sources of information that influence the method and the BigCQ dataset described in this chapter:

1. The dataset prepared by Ławrynowicz et al. [92], providing axiom patterns that frequently recur in a large set of ontologies.
2. ACE verbalizer [80], which translates axioms stated in OWL into English sentences.
3. CQ2SPARQLOWL, which is introduced in Chapter 6, provides examples of how CQs are formalized into queries.
4. CORAL [49], which provides the largest dataset of ontological requirements.

In the remainder of this section, we describe how we use each of these information sources.

8.1.1 Modeling patterns shared among ontological axioms

Ontological axioms are used to define how entities relate to each other. As we showed in Table 8.1, each ontology in CQ2SPARQLOWL introduces a large set of various types of logical axioms, among which the class expression axioms are the most common. A sample axiom can state that Every piece of software implements an algorithm, relating classes representing `software` and `algorithm` using a relation named `implements`.

Table 8.1: The number of axioms per ontology in CQ2SPARQLOWL.

Axiom type	SWO	Stuff	AWO	Dem@Care	OntoDT
Class axioms	7,469	326	51	575	951
Object property axioms	96	118	8	55	19
Data property axioms	6	31	0	46	0
Individual axioms	133	279	0	27	0

Frequent axiom patterns

Ławrynowicz et al. [92] constructed a dataset listing frequent axiom patterns (further referenced as *frequent axiom patterns dataset*). These patterns represent the most common modeling decisions identified among 331 ontologies coming from the BioPortal repository [183]. They used the *frequent axiom patterns dataset* to identify emergent ontology design patterns (ODPs) [136].

To construct the dataset, the authors selected a subset of axioms that meet the following criteria:

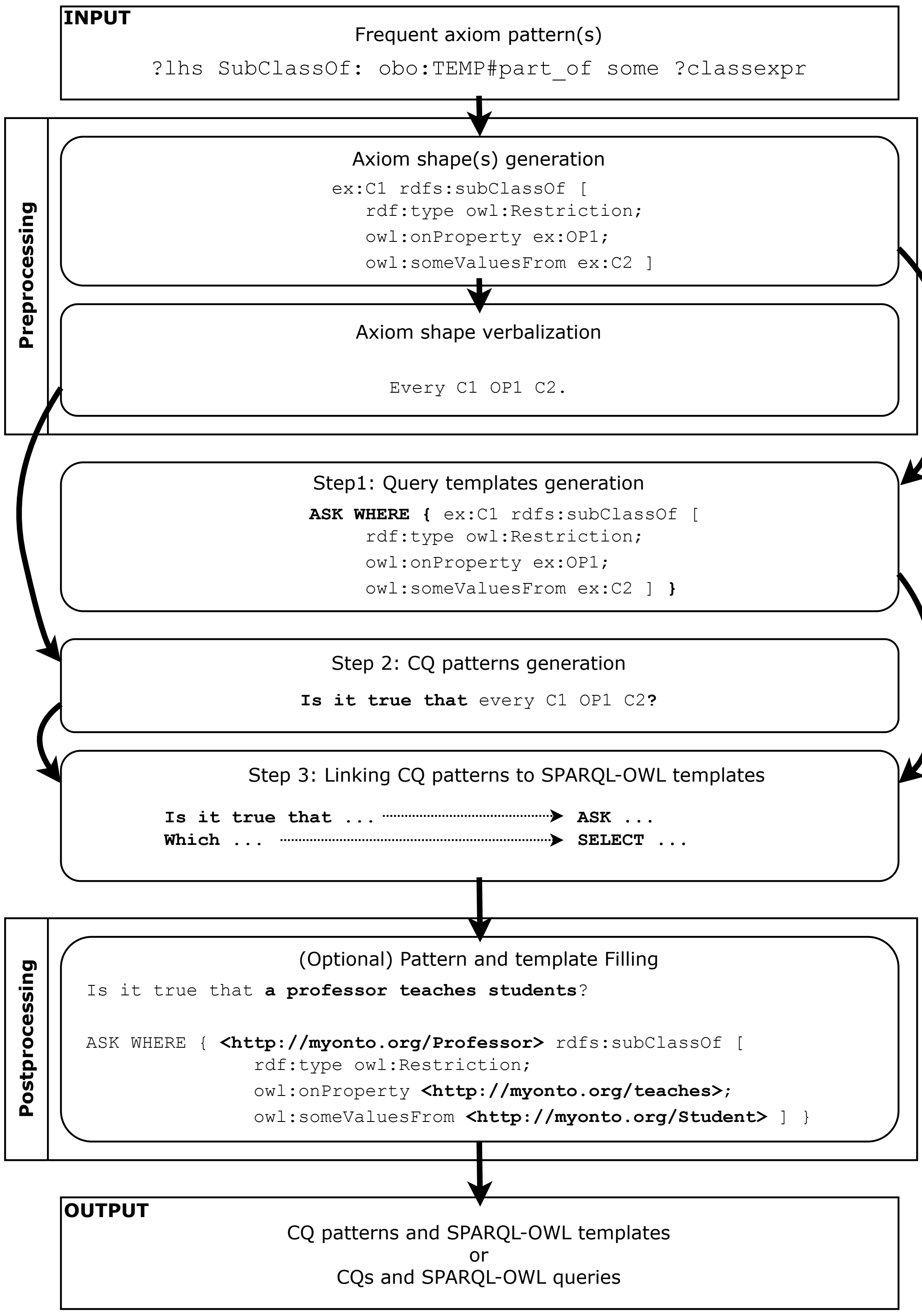

INPUT
Frequent axiom pattern(s)
?lhs SubClassOf: obo:TEMP#part_of some ?classexpr
Preprocessing
Axiom shape(s) generation
ex:C1 rdfs:subClassOf [rdf:type owl:Restriction; owl:onProperty ex:OP1; owl:someValuesFrom ex:C2]
Axiom shape verbalization
Every C1 OP1 C2.
Step1: Query templates generation
ASK WHERE { ex:C1 rdfs:subClassOf [rdf:type owl:Restriction; owl:onProperty ex:OP1; owl:someValuesFrom ex:C2] }
Step 2: CQ patterns generation
Is it true that every C1 OP1 C2?
Step 3: Linking CQ patterns to SPARQL-OWL templates
Is it true that ... → ASK ...
Which ... → SELECT ...
Postprocessing
(Optional) Pattern and template Filling
Is it true that a professor teaches students?
ASK WHERE { <http://myonto.org/Professor> rdfs:subClassOf [rdf:type owl:Restriction; owl:onProperty <http://myonto.org/teaches>; owl:someValuesFrom <http://myonto.org/Student>] }
OUTPUT
CQ patterns and SPARQL-OWL templates
or
CQs and SPARQL-OWL queries

FIGURE 8.1: The workflow of the method generating CQ and SPARQL-OWL forms that was used to construct BIGCQ.

- The axiom has to contain a named class on its left-hand side.
- The axiom has to express the subsumption or equivalence between classes.

Afterward, they applied the following procedure on those:

1. Transform all selected axioms into trees [92].
2. Use tree-mining techniques, namely SLEUTH [194] and its modification FF-SLEUTH [92] to identify the most common subtrees.
3. Serialize the most common subtrees. These are either whole axioms, axiom fragments, or axioms with elements replaced with variables.

The most popular axiom pattern extracted from BioPortal is `?lhs SubClassOf: obo:TEMP#part_of some ?classexpr`. This pattern introduces variables on both the left-hand side and the right-hand side (`?lhs`, `?classexpr`). Moreover, it introduces domain-related IRI (`obo:TEMP#part_of`).

Axiom shapes

We transformed each frequent axiom pattern into a domain-independent axiom shape using the following procedure:

1. Substitute all domain-related IRIs (outside of XSD, OWL, RDF, and RDFS namespaces), variables, and missing axiom fragments with artificial IRIs, according to forms defined in Table 8.2. The artificial IRIs encode the entity type and provide numerical identifiers for each entity of a given type.
2. Serialize the results in Turtle [138].

Table 8.2: Artifical IRIs used in axiom shapes. Reprint from [187].

IRI	Short name (local name)	Represents
`http://example.org/C{NUM}`	C{NUM}	class
`http://example.org/I{NUM}`	I{NUM}	individual
`http://example.org/OP{NUM}`	OP{NUM}	object property
`http://example.org/DP{NUM}`	DP{NUM}	data property
`http://example.org/DT{NUM}`	DT{NUM}	data type

By applying the procedure to the axiom pattern: `?lhs SubClassOf: obo:TEMP#part_of some ?classexpr`, we generate the following axiom shape:

```
<http://example.org/C1> rdfs:subClassOf [
        rdf:type owl:Restriction ;
        owl:onProperty <http://example.org/OP1> ;
        owl:someValuesFrom <http://example.org/C2> ]
```

We constructed 239 distinct axiom shapes by applying the described transformations to all axiom patterns. These provide the most common domain-independent modeling decisions among a diverse set of ontologies.

8.1.2 ACE verbalizer

Although Turtle serialization yields concise representations of axiom shapes, it is more natural for humans to interpret natural language. We decided to translate all axiom shapes into sentences expressed in English. For this purpose, we chose the ACE verbalizer [80][3] that uses Attempto Controlled English (ACE) [54] to express OWL axioms as statements in natural language.

The ACE verbalizer expects ontology properties to be labeled with verbs and ontology classes with noun phrases to generate grammatically correct verbalizations. It also provides a mapping of OWL constructs to English words.

An example verbalization generated by ACE is *Every computer that is owned by a gamer and that contains a dedicated graphics card is a gaming PC.*

8.1.3 Requirements collections

CQ2SPARQLOWL introduced in Chapter 6 is the biggest set of CQs formalized as SPARQL-OWL queries, but it is not the biggest set of ontology requirements in general. The CORAL dataset defines 834 requirements: 469 expressed as CQs and 365 as declarative sentences (e.g., A device has a unique identifier.), which make it much bigger than CQ2SPARQLOWL.

We decided to use both datasets in the following contexts:

- We used all 234 CQs and their SPARQL-OWL formalizations defined in CQ2SPARQLOWL to observe how CQs are constructed and how they relate to queries. This relation motivates the design choices behind the method presented in this chapter.
- We used all CQs present in CORAL and not included in CQ2SPARQLOWL for evaluation to verify to what extent the dataset we generate covers real, unseen CQs. As presented in Table 8.3, 324 CQs from CORAL are not included in CQ2SPARQLOWL.

Table 8.3: The size of requirement datasets involved in BigCQ construction and evaluation. Reprint from [187].

Dataset	Number of requirements
CORAL (all CQs + sentences)	834
CORAL (all CQs)	469
CORAL (all CQs that are not in CQ2SPARQLOWL)	324
CQ2SPARQLOWL (all CQs)	234
CQ2SPARQLOWL (all CQs with SPARQL-OWL queries defined)	131

8.2 Analysis of axiom shapes and their verbalizations

First, we analyze the axiom shapes collected and their verbalized forms. The observations we make in this step motivate the workflow of the method described later in this chapter.

8.2.1 Axiom shape verbalization groups

We used the ACE verbalizer [80] to process all 239 axiom shapes. As a result, 125 axiom shapes are successfully translated into sentences, while 114 failed to be translated because they represent axioms too complex for ACE verbalizer.

Using the ACE verbalizer, a sample axiom shape:

[3]https://github.com/Kaljurand/owl-verbalizer

```
<http://example.org#C1> rdfs:subClassOf [
        rdf:type owl:Restriction ;
        owl:onProperty <http://example.org#OP1> ;
        owl:someValuesFrom <http://example.org#C2> ]
```

is translated into Every C1 OP1 a C2. As can be seen, the ACE verbalizer uses local names extracted from IRIs to form sentences.

52 of the successfully verbalized axiom shapes express class subsumption using `rdfs:subClassOf`, which relates two, possibly complex, class expressions. Let us refer to the left-hand side class expression as `CE1` and the right-hand side one as `CE2`. Then, we can represent each axiom shape in this group using a common form `CE1 rdfs:subClassOf CE2`.

The remaining 73 of the successfully verbalized axiom shapes express class equivalence using `owl:equivalentClass`. The ACE verbalizer translates each such an axiom shape into two sentences with class subsumption because the form of `CE1 owl:equivalentClass CE2` is semantically equivalent to two-way subsumption: `CE1 rdfs:subClassOf CE2` and `CE2 rdfs:subClassOf CE1`.

A sample axiom shape:

```
<http://example.org#C1> owl:equivalentClass <http://example.org#C2>.
```

is translated into Every C1 is a C2. Every C2 is a C1..

We provide the list of verbalizations of all the verbalizable axiom shapes in Appendix D.

8.2.2 Mapping between fragments of axiom shapes and fragments of their verbalizations

Each verbalization can be represented using a shared form `LHS VERB RHS`, where:

- `VERB` – stands for the main verb that denotes the root of the dependency parse tree constructed over the sentence.
- `LHS` (left-hand side) – stands for the span of text ranging from the beginning of the sentence to the last non-whitespace character preceding `VERB`.
- `RHS` (right-hand side) – stands for the span of text ranging from the first non-whitespace character after `VERB` to the end of the sentence.

Since axiom shapes introducing class equivalence can be transformed into pairs of axioms with two-way subsumption, and because ACE verbalizer applies this transformation to produce verbalizations, in the remainder of this section, we focus on subsumption only.

As the verbalizations we generated are translations of axiom shapes into human language, there is a relation between `VERB`, `LHS`, `RHS` and `CE1`, `CE2`, `rdfs:subClassOf` defined. We observed that among the verbalizations listed in Appendix D, there are two groups of verbalizations. We visualized them in Figure 8.2. These are:

Verbalizations using the property label as the main verb If the (possibly complex) class expression `CE2` found at the right-hand side of the `rdfs:subClass` starts with a property restriction, the ACE verbalizer uses the label of the property as the main verb. In that case, as presented on the right-hand side of Figure 8.2:

1. Class expression `CE1` in the axiom shape relates to the LHS of the verbalization.

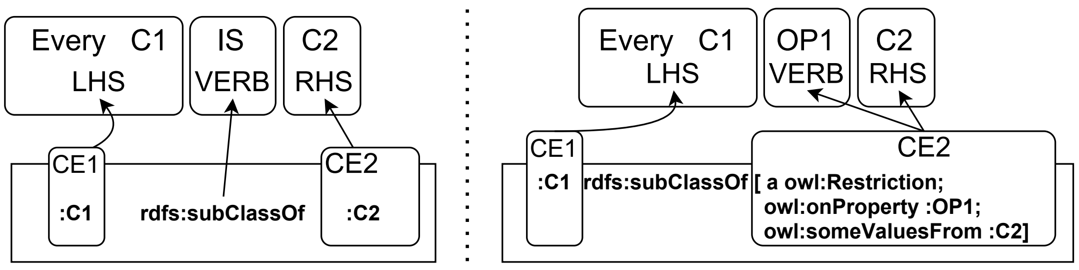


FIGURE 8.2: Relations between fragments of verbalizations and fragments of axiom shapes. Reprint from [187]

2. Class expression `CE2` relates to both RHS and the VERB of the verbalization.

A sample axiom:

```
<http://example.org#software> rdfs:subClassOf [
            rdf:type owl:Restriction ;
            owl:onProperty <http://example.org#implements> ;
            owl:someValuesFrom <http://example.org#functionality>
            ]
```

is verbalized into Every software implements a functionality so that:

- `CE1` (`<http://example.org#software>`) relates to the LHS (Every software).
- `CE2` (
```
[
    rdf:type owl:Restriction;
    owl:onProperty <http://example.org\#implements>;
    owl:someValuesFrom <http://example.org\#functionality>
])
```
  relates to both:
  - `<http://example.org#functionality>` to the RHS (a functionality).
  - `<http://example.org#implements>` to the VERB (implements).

Hereafter, we refer to this kind of verbalization as Subject-Predicate-Object (SPO).

Verbalizations using the word is as the main verb If the string representing `CE2` does not start with a property restriction, the word is is used to relate verbalizations of class expressions. In that case, LHS relates to `CE1`, RHS relates to `CE2` and the main verb is relates to `rdfs:subClassOf`.

A sample axiom:

```
<http://example.org#game> rdfs:subClassOf <http://example.org#software>
```

is verbalized into Every game is a software so that:

- `CE1` (`<http://example.org#game>`) relates to the LHS (Every game).
- `CE2` (`<http://example.org#software>`) relates to the RHS (a software).
- `rdfs:subClassOf` relates to VERB (is).

Hereafter, we refer to this case as Subclass-Superclass (SS).

Transformation of the SPO type into SS type We can observe that each verbalization of the SPO type can be simply transformed into the SS type. Considering the example mentioned above, one can transform: Every software implements a functionality into the SS form by putting the phrase is something that just before the verb representing the property label. In that case, the word is in Every software is something that implements a functionality becomes the root of the dependency parse tree (VERB), and something that implements a functionality becomes the RHS of the verbalization.

8.3 Method of translating axiom shapes into CQ patterns and query templates

In this section, we motivate and describe the method of transforming axiom shapes into SPARQL-OWL query templates and CQ patterns automatically.

8.3.1 Motivation

SPO and SS types in the context of question generation The distinction between SPO and SS types of verbalizations is helpful to generate grammatically correct questions automatically. Considering a sample verbalization of the SPO type: Every software implements an algorithm, one can ask the following question: Does every software implement an algorithm? generated by wrapping the verbalization with Does and ? strings and changing the 3rd person form of the verb implements into infinitive verb. However, verbalizations following SPO require different transformations than those following the SS type. In contrast to the example above, a sample verbalization following the SS type Every software is a piece of information cannot be transformed into a grammatically correct question by wrapping it with Does and ?.

Question targets Only 2% of CQs stated in CQ2SPARQLOWL represent questions asking for two or more entities at once. For this reason, we decided to make our method able to generate questions asking for at most a single thing. This restriction simplifies the problem, as in general, in queries, any combination of IRIs can be replaced with variables. Many of such replacements generate queries that are harder than the others to formulate as natural language questions.

In general, it is not equally simple to construct questions targeting different phrases stated in verbalized axioms. Considering a simple verbalized axiom: Every software implements an algorithm, it is easy to produce a question that asks about each ontological entity stated in the sentence:

1. What implements an algorithm? to ask about software.
2. What relates software and an algorithm? to ask about implements.
3. What does software implement? to ask about an algorithm.

However, let us consider a more complex example: Every dissertation is a piece of work that solves a problem that is analyzed by a Ph.D. student. Here, the LHS is represented by Every dissertation, the VERB is represented by is, and the RHS is a piece of work that solves a problem that is analyzed by a Ph.D. student. It is easy to ask about the LHS, as one can state the following question: How can we call a piece of work that solves a problem that is analyzed by a Ph.D. student?, but at the same time it is hard to provide a way of producing questions about some of the entities mentioned in the RHS, e.g., a Ph.D. student.

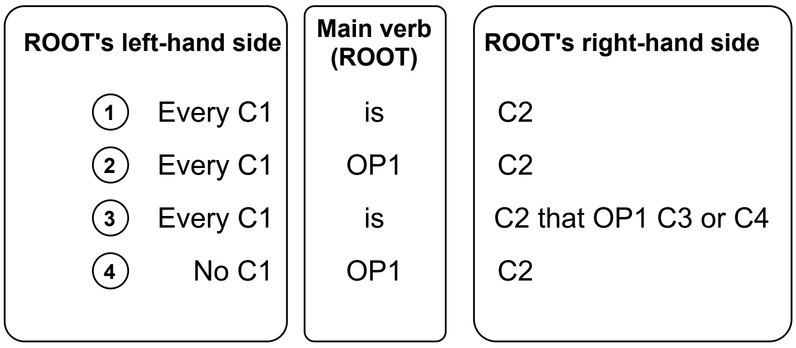


FIGURE 8.3: Verbalizations generated from example axiom shapes that are split into LHS, VERB and RHS. Reprint from [187]

In Figure 8.3, we present examples of verbalizations marked with LHS, VERB, and RHS. In general, it is easy to ask about LHS or RHS if it consists of a single named class. Considering the first example (Every C1 is a C2), we can ask about the LHS using What are the specializations of C2? and the RHS using What kind of thing is every C1?. Considering the second example (Every C1 OP1 a C2), we can ask about the LHS using What OP1 a C2? and the RHS using Which things does C1 OP1?.

However, stating questions about particular ontological entities in the LHS in the 4th example (No C1 OP1 a C2) or the RHS in the 3rd example (Every C1 is a C2 that OP1 a C3 or a C4) is very hard and would cause the questions to be complicated (e.g., a question about C4 in the 3rd example). For these reasons, we decided that in our method, we can ask for:

- LHS – if it corresponds to a single named class.
- RHS – if it corresponds to a single named class.
- VERB – if it corresponds to a property label.

8.3.2 Step 1: Query templates generation

As described in Section 6.4, there are 4 query forms that SPARQL handles: `ASK`, `SELECT`, `DESCRIBE` and `CONSTRUCT`. The first two: `ASK` and `SELECT` are useful in the context of CQs.

In the remainder of this subsection, we use the following axiom shape:

```
<http://example.org#C1> rdfs:subClassOf [
    rdf:type owl:Restriction ;
    owl:onProperty <http://example.org#OP1> ;
    owl:someValuesFrom <http://example.org#C2> ]
```

as a running example.

Generating ASK queries To construct an `ASK` query that checks if a given BGP matches in the ontology, one has to wrap a given axiom shape with `ASK WHERE { ...}` preamble and postamble. The following `ASK` query can be constructed for our running example:

```
ASK WHERE { <http://example.org#C1> rdfs:subClassOf [
                rdf:type owl:Restriction ;
                owl:onProperty <http://example.org#OP1> ;
                owl:someValuesFrom <http://example.org#C2> ] }
```

For each axiom shape, we use this procedure to produce a single ASK query.

Generating SELECT queries Similar to the *ASK* type of queries, SELECT also requires wrapping BGPs with appropriate preamble and postamble. However, it also expects a set of variables introduced in the BGP.

An example query:

```
SELECT ?x WHERE { ?x rdfs:subClassOf [
                rdf:type owl:Restriction ;
                owl:onProperty <http://example.org#OP1> ;
                owl:someValuesFrom <http://example.org#C2> ] }
```

asks about things that are in relation OP1 with instances of the class C2. The variable ?x is used to collect the matched IRIs, which are then presented to the user.

A single query can introduce multiple variables to be matched in the ontology. For example, the following query:

```
SELECT ?x ?y WHERE { ?x rdfs:subClassOf ?y }
```

simply searches for pairs of subclass and a superclass, but as we stated in the previous subsection, our method does not support queries with multiple variables.

A special case of the SELECT type are queries that are stated to count rather than list all the matches. This can be achieved by wrapping a variable listed after SELECT and WHERE keywords with (COUNT(variable_name) as count_variable_name). For example, the following query counts how many classes are subclasses of C1:

```
SELECT (COUNT(?x) as ?count) WHERE { ?x rdfs:subClassOf <http://example.org#C1> }
```

Considering the decisions made in Section 8.2, from each axiom shape, one can produce up to 7 kinds of queries:

- A single ASK query.
- A pair of SELECT and SELECT COUNT stated to ask for the LHS if it represents a single named class.
- A pair of SELECT and SELECT COUNT stated to ask for the RHS if it represents a single named class.
- A pair of SELECT and SELECT COUNT stated to ask for the VERB if it represents a property label.

In Table 8.4, we listed all kinds of query types generated from the running example.

If a given verbalization introduces a complex class expression in LHS or RHS or if the axiom shape represents the SS type, we can construct only a subset of the 7 types of queries. We produce all possible queries from a given axiom shape. Since the queries contain placeholders instead of real IRIs, we refer to the generated representations as query templates.

Table 8.4: All supported forms of query templates generated from `<C1> rdfs:subClassOf [a owl:Restriction; owl:onProperty <OP1>; owl:someValuesFrom <C2>]` axiom shape. Reprint from [187].

Query Type	Example of query template
ASK	`ASK WHERE {<C1> rdfs:subClassOf [a owl:Restriction; owl:onProperty <OP1>; owl:someValuesFrom <C2>]}`
SELECT LHS	`SELECT ?x WHERE {?x rdfs:subClassOf [a owl:Restriction; owl:onProperty <OP1>; owl:someValuesFrom <C2>]}`
SELECT RHS	`SELECT ?x WHERE {<C1> rdfs:subClassOf [a owl:Restriction; owl:onProperty <OP1>; owl:someValuesFrom ?x]}`
SELECT VERB	`SELECT ?x WHERE {<C1> rdfs:subClassOf [a owl:Restriction; owl:onProperty ?x; owl:someValuesFrom <C2>]}`
SELECT COUNT LHS	`SELECT COUNT(?x) WHERE {?x rdfs:subClassOf [a owl:Restriction; owl:onProperty <OP1>; owl:someValuesFrom <C2>]}`
SELECT COUNT RHS	`SELECT COUNT(?x) WHERE {<C1> rdfs:subClassOf [a owl:Restriction; owl:onProperty <OP1>; owl:someValuesFrom ?x]}`
SELECT COUNT VERB	`SELECT COUNT(?x) WHERE {<C1> rdfs:subClassOf [a owl:Restriction; owl:onProperty ?x; owl:someValuesFrom <C2>]}`

8.3.3 Step 2: CQ patterns generation

For each of the 7 query types defined in Section 8.3.2, we manually prepared a list of transformations used to transform each verbalization into question patterns automatically. A sample of those transformations can be found in Table 8.5. Our goal was to provide as many transformations as possible for each of the types so that large and diverse datasets of CQs could be constructed.

Table 8.5: Examples of CQ templates used to transform verbalized axiom shapes following the SPO type and expressing subsumption. Here, {VERB} stands for a property label. {LHS} and {RHS} stand for LHS and RHS, respectively. Reprint from [187].

Question type	Example of CQ template
`ASK`	`Does {LHS} {VERB} {RHS}?`
`Questions asking for LHS`	`What {VERB} {RHS}?`
`Questions asking for RHS`	`What does {LHS} {VERB} ?`
`Questions asking for VERB`	`What relates {LHS} and {RHS}?`
`Questions asking for COUNT LHS`	`How many things {VERB} {RHS}?`
`Questions asking for COUNT RHS`	`How many things does {LHS} {VERB}?`
`Questions asking for COUNT VERB`	`How many relations are there between {LHS} and {RHS}?`

For a given axiom shape verbalization, we mark LHS, VERB, and RHS and use them to fill predefined question templates. For example, let us consider the following axiom shape verbalization: Every C1 OP1 a C2 that OP2 a C3.. Here, the VERB is represented by OP1 (a label of an object property), RHS is represented by a C2 that OP2 a C3 and the LHS is represented by Every C1.

We can construct the following CQ patterns by filling CQ templates defined in Table 8.5:

- ASK: Does every C1 OP1 a C2 that OP2 a C3?
- Asking for LHS: What OP1 a C2 that OP2 a C3?
- Asking for VERB: What relates every C1 and a C2 that OP2 a C3?
- Asking for COUNT LHS: How many things OP1 a C2 that OP2 a C3?
- Asking for COUNT VERB: How many relations are there between every C1 and a C2 that OP2 a C3?

Most verbalizations start their LHSs with the word every, which can be used to fill CQ templates, but its lack does not change the meaning of the constructed question. For this reason, we decided to strip the word every when filling placeholders in CQ templates.

As can be seen, out of the 7 possible question types defined in Table 8.5, we generated CQ patterns for only 5, since RHS represents a complex class expression, querying which is not supported by the method as presented earlier in this section.

Our method takes a list of CQ templates assigned to each question type as one of the inputs. Moreover, separate sets of CQ templates should be provided for axiom shape verbalizations following SPO and SS types, as well as for verbalizations expressing subsumption or class equivalence to generate grammatically correct questions.

To understand why separate sets of CQ templates have to be provided, let us analyze how different examples of CQ templates asking for the RHS of a verbalization should look like:

- Subsumption + SPO: What does LHS VERB?
- Subsumption + SS: What kind of thing is LHS?
- Class equivalence + SPO: What is another name for things that LHS VERB?
- Class equivalence + SS: What kind of thing is another name for LHS?

For each of these 4 combinations, we provide possibly large sets of question templates categorized into question types. A sample of these for the *Subsumption + SPO* scenario is provided in Table 8.5.

Synonym sets in CQ templates In English, multiple words may share the same meaning and can be used interchangeably. For example, a CQ that starts with the word Which can have the word replaced with What and still convey the same meaning. A CQ: Which games can be regarded as open source? can be paraphrased by replacing the phrase regarded with its synonyms, e.g., Which games can be counted as open source?, Which games can be categorized as open source?, Which games can be considered as open source?.

We use that observation to provide CQ templates with synonym sets, substrings enclosed with square brackets that are substituted with all predefined synonyms to multiplicate the number of CQ templates. For example, with two predefined synonym sets:

- [kinds] representing the following synonyms: kinds, categories, classes
- [What] representing the following synonyms: What, Which

, one can transform the following CQ template using synonym sets: [What] LHS are [kinds] of RHS? to produce the following CQ templates:

- What LHS are kinds of RHS?
- Which LHS are kinds of RHS?
- What LHS are categories of RHS?
- Which LHS are categories of RHS?
- What LHS are classes of RHS?
- Which LHS are classes of RHS?

8.3.4 Step 3: Linking CQ patterns to SPARQL-OWL templates

Both SPARQL-OWL templates and CQ patterns defined above contain information about the question type they are used to express. Since both Tables: 8.4 and 8.5 define 7 kinds of queries and questions sharing their types, we can simply relate each SPARQL-OWL query template constructed for a given question type to all CQ templates created for the same question type.

8.4 BigCQ: a dataset of CQ patterns mapped to SPARQL-OWL templates

We implemented the method defined in Section 8.3 in Python and published it online[4]. Based on the 125 successfully verbalized axiom shapes collected and the forms of CQs in the CQ2SPARQLOWL dataset, we generated a comprehensive list of CQ templates, listed in Appendix G, which are used to translate axiom shape verbalizations into CQ patterns. We tried to provide as many CQ templates as possible to cover the multiple forms CQs may have. We handcrafted a list of synonym sets that is provided in Appendix F. It is used in CQ templates to multiplicate the size of the generated dataset.

As a result, as presented in Table 8.6, we created 77,575 CQ patterns assigned to 549 unique SPARQL-OWL templates. The dataset provides over 93x more CQs patterns than the total number of CQs collected in CORAL and 12x more SPARQL-OWL templates than in CQ2SPARQLOWL. It is important to note that BigCQ contains CQ patterns and SPARQL-OWL templates that should be filled with domain vocabulary to form real CQs and queries. For example, the CQ pattern: Does C1 OP1 C2?, in the context of the food-related domain, can be filled with various ontological entity labels to form CQs, e.g., Does cheese contain gluten?, Does expired food causes illness?. For this reason, BigCQ may be used to generate millions of CQs and SPARQL-OWL queries. In Table 8.6, we show that a single CQ pattern may be linked to multiple query templates, and conversely, a single SPARQL-OWL query template may be linked to many CQ patterns.

The one-to-many relation between SPARQL-OWL templates and CQ patterns comes from the multiple CQ templates with various synonym sets defined for each question type. One-to-many relations between a single CQ pattern and SPARQL-OWL templates come from ambiguous CQ templates. For example, the CQ template: What is C1 can be interpreted as a question about listing all subclasses or all superclasses of C1. In the first case, we define C1 by showing what specializations it has, while in the second case, we define C1 by showing where it lies in the taxonomy.

[4]`https://github.com/dwisniewski/BigCQ`

Table 8.6: Overall summary of BigCQ. Reprint from [187].

Measured dimension	Measured value
Number of distinct CQ patterns	77,575
Number of distinct SPARQL-OWL query templates	549
Average number of CQ patterns per SPARQL-OWL template	171.68
Average number of SPARQL-OWL templates per a CQ pattern	1.22

Figure 8.4 presents the number of CQ patterns generated for each of the 7 question types analyzed. As can be seen, the vast majority of CQ patterns express `ASK` queries. There are two reasons for this state of affairs:

1. It is always possible to state a CQ pattern and a SPARQL-OWL template to express `ASK` questions regardless of the complexity of class expressions observed in the axiom shape.
2. It is relatively easy to construct a large list of various question forms represented as CQ templates that ask if a given BGP matches in the ontology.

Frequently, RHSs of verbalizations represent complex class expressions, providing multiple entities our method cannot ask for. For this reason, many more questions are asking about LHSs rather than RHSs.

We found counting questions and questions about verbs the hardest ones to paraphrase so that they are represented with the smallest number of CQ patterns.

BigCQ introduces a subset of OWL vocabulary that is rarely used among SPARQL-OWL queries in CQ2SPARQLOWL. In Table 8.7, we listed all IRIs coming from OWL and RDFS namespaces and ranked them by the number of occurences in BigCQ. Considering those, the following ones: `owl:equivalentClass`, `owl:qualifiedCardinality`, `owl:maxQualifiedCardinality`, `owl:complementOf`, and `owl:minCardinality` are present in BigCQ but are not used in any query defined in CQ2SPARQLOWL.

Table 8.7: The number of the most frequent (present in more than 5% of BigCQ query templates) OWL and RDFS-related IRIs. Reprint from [187].

Construct	Times observed
`owl:Restriction`	504/549
`owl:onProperty`	504/549
`owl:intersectionOf`	319/549
`rdfs:subClassOf`	268/549
`owl:equivalentClass`	281/549
`owl:someValuesFrom`	273/549
`owl:qualifiedCardinality`	87/549
`owl:hasValue`	71/549
`owl:unionOf`	69/549
`owl:allValuesFrom`	56/549
`owl:maxQualifiedCardinality`	38/549
`owl:complementOf`	37/549
`owl:minCardinality`	30/549

8.5 Coverage of BigCQ measured on existing datasets

To check how well CQ patterns and SPARQL-OWL templates cover existing datasets, we evaluated BigCQ on datasets that did not influence the method‘s workflow.

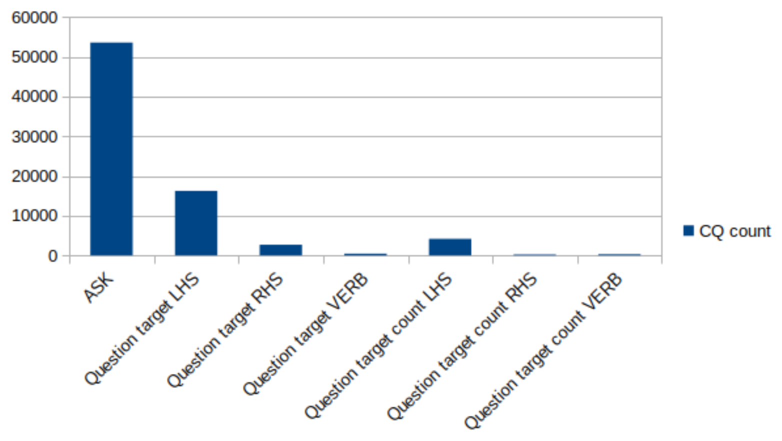


FIGURE 8.4: Number of CQ patterns generated for each of the 7 categories. Reprint from [187]

As SPARQL-OWL query templates created in BIGCQ are simple transformations of axiom shapes and are not influenced by queries found in CQ2SPARQLOWL, we used all queries introduced in CQ2SPARQLOWL to evaluate the coverage of query templates on real cases.

We evaluated BIGCQ in the following way:

- From each of the 324 CQs unique for CORAL (that is not included in CQ2SPARQLOWL), we replaced all domain-related vocabulary with placeholders. Then, an expert verified if this form is equal to any of the CQ patterns defined in BIGCQ.
- From each SPARQL-OWL query in CQ2SPARQLOWL, we replaced all domain-related IRIs with placeholders. Then, an expert verified if this form is equal to any of SPARQL-OWL templates defined in BIGCQ.

We provide the results of the evaluation in Table 8.8. Regarding SPARQL-OWL templates, the coverage of query templates varies between ontologies in CQ2SPARQLOWL. The highest one was observed for the AWO Ontology (71%), while the lowest one was observed for Stuff Ontology (9%). Considering SPARQL-OWL queries from all ontologies jointly, 45.74% of them are covered by BIGCQ.

We found that 19.38% of queries defined in CQ2SPARQLOWL share the following query template:

```
SELECT ?x WHERE {
    [] rdfs:subClassOf <C1>, [owl:onProperty ?x; owl:someValuesFrom [] ].
}
```

This template is not provided in BIGCQ. However, such a template can be found in Dem@Care ontology only. Moreover, 11.63% of queries in CQ2SPARQLOWL express the following template:

```
SELECT ?x WHERE {
    <C1> rdfs:subClassOf [
        a ow:Restriction ;
        owl:onProperty <OP1>;
        owl:someValuesFrom ?x
    ] . ?x rdfs:subClassOf <C2>
```

Table 8.8: The coverage of CQ patterns and SPARQL-OWL templates on real-world cases. Reprint from [187].

Dataset	Coverage
SPARQL-OWL queries from CQ2SPARQLOWL	45.74%
CQs unique for CORAL (that are not in CQ2SPARQLOWL)	63.89%

```
}
```

, which is also missing in BigCQ. As can be seen, adding support for these two SPARQL-OWL query templates can increase the query coverage measured on the CQ2SPARQLOWL dataset by over 30 percentage points.

The remaining SPARQL-OWL queries that are not covered can be categorized as follows:

- Queries with variables, the bindings of which are not returned to the user.
- Queries asking for ontological entities stated in complex class expressions.
- Queries with `owl:disjointWith`. These are not covered by the *frequent axiom patterns dataset*.
- Queries using `union` keyword rather than `owl:unionOf`.
- Queries asking for multiple resources at once (introducing multiple variables).

Regarding CQs stated in CORAL that are not included in CQ2SPARQLOWL, the BigCQ dataset covers almost 64% of forms presented there. Considering that human language provides rich possibilities for expressing questions, such a score can be interpreted as a success.

Some CQs that are not covered by BigCQ are:

- What are the rivers which belong to municipality x?, which is not covered because of the presence of the second which word.
- Through which autonomous community does the river x flow into?, which is not covered since no CQ pattern in BigCQ starts with the Through word.
- Where does the river X flows into?, which is not covered since no CQ pattern in BigCQ starts with the Where word.
- Who is the owner of a given device?, which is not covered since no CQ pattern in BigCQ starts with the Who word.
- What consumable items does a player have in game?, which is not covered because such a question form is not supported by BigCQ.
- After gaining an item in the game, how many players use it?, which is not covered because there is an additional context introduced before the comma.

8.6 Summary

In this section, we discuss how BigCQ can be used to generate materialized CQs and SPARQL-OWL queries. Moreover, we provide a list of areas where incorporating such a dataset may increase the quality of a given method, tool, or analysis.

8.6.1 Filling BigCQ with domain-related vocabulary

BigCQ introduces CQ patterns paired with SPARQL-OWL templates. We can materialize them by filling artificial identifiers with domain-related labels and IRIs. We provide a simple proof of concept piece of code showing how CQ patterns and SPARQL-OWL templates can be populated with labels and IRIs extracted from ontology axioms[5].

However, we have to remember that the CQ patterns filled with ontology vocabulary may require additional postprocessing. If, when materializing the CQ pattern: Which C1 is a C2?, C2 is filled with a noun in plural, the word is should be replaced with are and a should be omitted.

Similarly, CQ patterns like: Does C1 OP1 a C2? should replace Does with Do if the label used to fill C1 uses a noun in plural.

A subset of CQ patterns mapped to SPARQL-OWL templates involve cardinality restrictions. These include numeric values that are represented using {NUM} value markers shared between patterns and templates (e.g., Every C1 is a C2 that OP1 at most {NUM} thing.). In BigCQ, we replaced the numeric values with a single selected value. However, these could be treated analogously to synonym sets so that we could produce multiple patterns and templates by filling the numeric value markers with various popular numbers.

8.6.2 Potential applications of BigCQ

We found several research projects where BigCQ may be beneficial:

- CLARO by Keet et al. [86] provides a controlled natural language (CNL) guiding how to construct CQs. Since CLARO is built from a set of only 234 CQ patterns, adding over 77,000 CQ patterns from BigCQ will increase the expressivity of that tool.
- Glossary of terms taggers introduced in Chapter 7. BigCQ can provide a large dataset to train taggers and handcraft rules.
- Ren et al. [148] provided the idea of testing ontologies with competency questions. They collected CQs defined for two ontologies, then extracted and analyzed a set of 12 archetypes from them. Applying CQ templates or materialized CQs from BigCQ could result in a larger collection of archetypes analyzed.
- BigCQ could be used in a new challenge in QALD[6] [174] as a task for querying terminological parts of ontologies. The scripts transforming verbalizations (sentences) into questions that we published with BigCQ may be used to generate questions from documents automatically.

8.6.3 Examples of CQ patterns and SPARQL-OWL query templates

Below, we provide some examples of CQ patterns formalized as SPARQL-OWL templates that are included in BigCQ

Which sort of entities OP1 C2 that OP2 C3?

```
SELECT ?x WHERE {
    ?x rdfs:subClassOf [
        rdf:type owl:Restriction ;
        owl:onProperty <OP1> ;
```

[5] https://github.com/dwisniewski/BigCQ/blob/main/template_materialization_poc/materialize.py
[6] http://qald.aksw.org

```
            owl:someValuesFrom [
                owl:intersectionOf (
                    <C2> [
                        rdf:type owl:Restriction ;
                        owl:onProperty <OP2> ;
                        owl:someValuesFrom <C3>
                    ] )
            ]
        ] . }
```

Could we regard C1 the same as C2 that OP1 C3 and that OP2 nothing but C4?

```
ASK WHERE {
    <c1> owl:equivalentClass [
        owl:intersectionOf ( <C2> [
            rdf:type owl:Restriction ;
            owl:onProperty <OP1> ;
            owl:someValuesFrom <C3> ] [
            rdf:type owl:Restriction ;
            owl:onProperty <OP2> ;
            owl:allValuesFrom <C4> ]
        ) ] . }
```

How many categories of things does C1 OP1?

```
SELECT (COUNT(?x) AS ?cnt) WHERE {
    <C1> rdfs:subClassOf [
        rdf:type owl:Restriction ;
        owl:onProperty <OP1> ;
        owl:someValuesFrom ?x ]}
```

Chapter 9

SeeQuery: a recommender of SPARQL-OWL queries for CQs

In this chapter, we introduce SeeQuery – an automatic method making SPARQL-OWL query recommendations out of CQs. SeeQuery consists of a pipeline of 6 processing steps aimed to generate queries via template matching and filling.

The structure of this chapter is as follows: In Section 9.1, we introduce materials used to design and evaluate SeeQuery. In Section 9.2, we describe each of the processing steps and evaluate SeeQuery in Section 9.3. We discuss the limitations of SeeQuery and errors observed during the evaluation in Section 9.4. Finally, we conclude in Section 9.5.

9.1 Materials

9.1.1 CQs translated into SPARQL-OWL

SeeQuery is a data-driven method, developed based on existing collections of CQs translated into SPARQL-OWL that we introduced in Chapters 6 (CQ2SPARQLOWL) and 8 (BigCQ). These, as described later in this section, are transformed into a common set consisting of pairs of domain-agnostic CQ patterns and SPARQL-OWL query templates. We call that set *patterns and templates set* and use it to guide the method on how given forms of CQs map to queries (e.g., CQs starting with *can* are expressed using the `ASK` query form).

Approximation filtering CQ2SPARQLOWL contains a subset of translations, in which entities mentioned in CQs are not explicitly modeled in an ontology. Let us consider the CQ SWO_76: *Is there a publication with [it]?* and its translation in SPARQL-OWL:

```
prefix swo: <http://www.ebi.ac.uk/swo/>
ASK WHERE { $PPx1$ rdfs:subClassOf [ rdf:type owl:Restriction ;
                owl:onProperty swo:SWO_0000043 ;
                owl:hasValue ?doc ] .
filter(STRSTARTS(?doc, "http://dx.doi.org/")) }
```

We observe that even though SWO does not introduce any ontological entity representing *a publication*, it can be approximated using some operation over the entities provided in the ontology. In this query, we check whether the documentation is defined for a given piece of software and if the IRI of the documentation starts with `http://dx.doi.org/`. The placeholder `$PPx1$` –

related to placeholder *[it]* in the CQ – represents a given piece of software, while the property `swo:SWO_0000043` represents a relation labeled as *has documentation.*

In other words, the engineer who stated the query used their expertise to approximate a publication with documentation having a DOI. The need for external knowledge to construct approximations makes such pairs of CQs and queries unusable for automatic translators. For this reason, we decided to reject all approximated queries for further processing.

As a result, considering CQ2SPARQLOWL, a subset of 99 out of 131 pairs of CQs translated into SPARQL-OWL are used in SeeQuery. The problem of approximations does not occur in BigCQ, so we use all 77,575 CQ patterns and their translations.

Patterns and templates set *Patterns and templates set* is the set of domain-agnostic CQ patterns mapped to SPARQL-OWL query templates. We built it based on CQ2SPARQLOWL with approximated translations rejected and BigCQ datasets using the following procedure:

- Considering CQ2SPARQLOWL, for each CQ and its SPARQL-OWL translation, we manually related phrases stated in a CQ to IRIs in the query. Then, we replaced the related elements with common chunk identifiers shared between each CQ and its query:
  - EC{IDX} – used if the IRI represents an instance or a class.
  - PC{IDX} – used if the IRI represents an object property or a data property.

  In both cases, {IDX} is a unique numeric chunk identifier. The visualization of this step is presented in Figure 9.1.

- As BigCQ already contains domain-agnostic patterns and templates differing only in placeholder naming convention, we transformed placeholders for classes and instances into EC markers and placeholders for properties into PC markers using regular expressions.

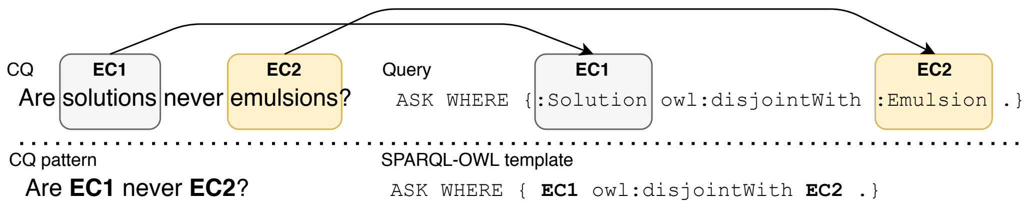

CQ
EC1
EC2
Are solutions never emulsions?
Query
EC1
EC2
ASK WHERE {:Solution owl:disjointWith :Emulsion .}
CQ pattern
Are EC1 never EC2?
SPARQL-OWL template
ASK WHERE { EC1 owl:disjointWith EC2 .}

Figure 9.1: CQ pattern to SPARQL-OWL template mapping example. Relations between phrases and IRIs are marked in the upper part of the Figure. In the lower part of the Figure, a pair of a CQ pattern and a SPARQL-OWL query template is presented. Reprint from [188].

9.1.2 Evaluation set

To construct the evaluation set, we searched the web for CQs and ontologies other than those described in Section 9.1.1 that meet the following criteria:

1. The set of CQs and the ontology have to be publicly available.
2. Either SPARQL-OWL queries for CQs should be publicly available, or the domain covered by a given ontology should be familiar to us so that it is possible to assess if the queries generated by SeeQuery are correct or not.

3. At least one CQ should ask for the terminological part of the ontology.

We found two ontologies meeting the criteria:

1. Pizza Ontology [149] – a popular ontology used in OWL teaching tutorials with 42 requirements provided.
2. TrhOnt [12] – an ontology describing rehabilitation domain with 20 CQs provided.

There are 12 requirements in Pizza Ontology that are in the form of statements rather than CQs. We rephrased each such statement into question (e.g., Find all the nut free pizzas. into Which pizzas are nut free?).

9.2 Method description

SeeQuery consists of a sequence of 6 processing steps as presented in Figure 9.2. The method requires a CQ and an ontology (in our implementation, we expect a CQ to be a string and an ontology to be provided as an OWL/XML file). SeeQuery transforms these inputs into at least one SPARQL-OWL query if it is possible or produces an error message otherwise.

While the detailed description of the method is provided later in this section, we can summarize it as:

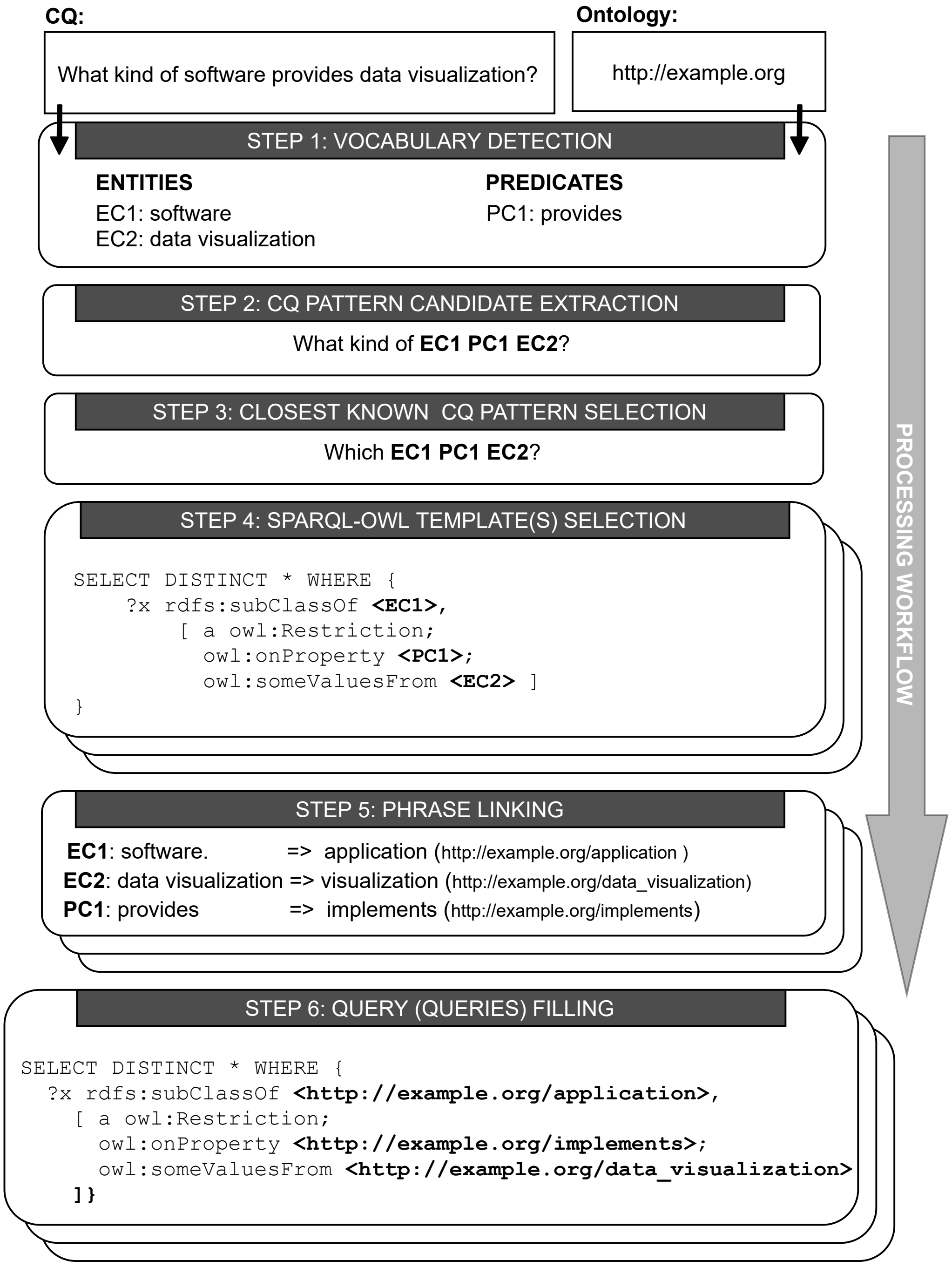
1. Analyze the CQ and search for phrases representing domain-related vocabulary that should be modeled in the ontology as classes, individuals, or properties.
2. Remove domain-related vocabulary from the CQ to form a CQ pattern candidate.
3. Identify a CQ pattern in *patterns and templates set* that is the closest one in terms of meaning to our CQ pattern candidate.
4. Select all SPARQL-OWL query templates assigned to the CQ pattern selected in the previous step.
5. Link phrases in the CQ to labels of ontological entities.
6. Fill selected SPARQL-OWL query templates with IRIs of the ontological entities linked to phrases.

9.2.1 Step 1: Vocabulary detection

IRIs, which in OWL ontologies serve as unique references to ontological entities, are used to construct graph patterns of SPARQL-OWL queries. For this reason, the first processing step focuses on parsing CQs to identify domain-related phrases in the input CQ. These phrases should relate to ontological entities.

First, let us introduce basic terms. Let us consider an ontological entity label as a string extracted from the ontological entity using:

1. `skos:prefLabel` property value if it is provided and nonempty. If multi-language labels are provided (using language tags), we choose the one written in English. If multiple labels in English are provided, we chose the first one.
2. If there is no `skos:prefLabel`, use `rdfs:label` property value using the same procedure as defined in the previous point.


CQ:
What kind of software provides data visualization?
Ontology:
http://example.org
STEP 1: VOCABULARY DETECTION
ENTITIES
EC1: software
EC2: data visualization
PREDICATES
PC1: provides
STEP 2: CQ PATTERN CANDIDATE EXTRACTION
What kind of EC1 PC1 EC2?
STEP 3: CLOSEST KNOWN CQ PATTERN SELECTION
Which EC1 PC1 EC2?
STEP 4: SPARQL-OWL TEMPLATE(S) SELECTION
STEP 5: PHRASE LINKING
EC1: software. => application (http://example.org/application)
EC2: data visualization => visualization (http://example.org/data_visualization)
PC1: provides => implements (http://example.org/implements)
STEP 6: QUERY (QUERIES) FILLING
PROCESSING WORKFLOW

```
SELECT DISTINCT * WHERE {
    ?x rdfs:subClassOf <EC1>,
        [ a owl:Restriction;
          owl:onProperty <PC1>;
          owl:someValuesFrom <EC2> ]
}
```

```
SELECT DISTINCT * WHERE {
  ?x rdfs:subClassOf <http://example.org/application>,
    [ a owl:Restriction;
      owl:onProperty <http://example.org/implements>;
      owl:someValuesFrom <http://example.org/data_visualization>
    ]}
```

FIGURE 9.2: The workflow of SEEQUERY. The outputs of the current step serve as the inputs of the subsequent step in a processing pipeline, the flow of which is represented with the grey arrow. Multiplicated boxes visualized in the background of steps 4-6 indicate that SEEQUERY can select and process more than one query template at once to produce multiple query recommendations. If that is the case, SEEQUERY processes all these templates independently.

3. If there is no `skos:prefLabel`, nor `rdfs:label`, use the local name extracted from the IRI of an entity.

Since local names encoded in IRIs are often artificial identifiers that are hard to be understood by humans (e.g., `123834` in `http://example.org/123834`), we prioritize other label sources if they are provided.

Let the normalized entity label be an ontological entity label with all non-alphanumeric characters removed and tokens normalized to a space-separated format (e.g., the snake case *some_entity_label* is normalized into *some entity label*).

The analysis of CQ2SPARQLOWL shows that it is not trivial to relate ontological entity labels to phrases in CQs. We identified two main reasons that make the problem difficult:

- Grammar-related – phrases stated in CQs are sometimes expressed using different grammar forms than labels in ontologies (e.g., singular versus plural nouns, different verb tense or person). Two examples of grammar-related differences found in CQ2SPARQLOWL are provided in the first two rows of Table 9.1.

- Synonym-related – phrases stated in CQs are sometimes expressed using different words that serve as synonyms to labels in ontologies (e.g., software – application, provides – implements). Three examples of such differences are presented in the last three rows of Table 9.1.

TABLE 9.1: Differences between phrases stated in CQs and ontological entity labels. Reprint from [188].

CQ id	CQ	Ontological entity label	Difference
Stuff_06	Are **solutions** never **emulsions**?	Solution, Emulsion	plural vs singular
AWO_04	Does a lion **eat** plants or plant parts?	eats	1st vs 3rd person
SWO_14	Which software tool **created** [this data]?	has specified data output	synonym
SWO_53	Is this software **available as** a web service?	has interface	synonym
Dem@Care_3	What types of **demographic data** are collected?	DemographicCharacteristics-Record	synonym

The grammar and synonym-related differences show that a simple search for normalized ontological entity labels in CQs (further referenced as *direct matching*) generates incomplete results. For this reason, we propose a two-step procedure for vocabulary detection consisting of *direct matching* followed by *domain phrases extraction*.

As the first step, we use *direct matching* to check which normalized entity labels are explicitly stated in the CQ. If a match is found, we classify it, depending on the ontological entity type:

- as an entity chunk – if the entity represents an individual or a class.

- as a predicate chunk – if the entity represents an object property or a data property.

Moreover, we mark where the match starts and ends in the CQ. *Direct matching* assures that every ontological label mentioned explicitly in the CQ is found.

As the second step, we use ReqTagger introduced in Chapter 7 to perform *domain phrases extraction*. ReqTagger identifies and classifies domain-related phrases independently from the ontology. We consider all phrases extracted with ReqTagger, the start and end offsets of each, and the type suggestions generated so that phrases suggested as terms and relations are classified as entity chunks and predicate chunks, respectively. The output of *domain phrases extraction* is related to ontology entities in subsequent processing steps.

9.2.2 Step 2: CQ pattern candidate extraction

The CQ provided as an input as well as domain-related vocabulary detected in the previous step are used to construct a domain-agnostic CQ pattern candidate.

A CQ pattern candidate is generated out of a CQ in the following way:

1. Replace with a unique entity chunk placeholder (`EC{NUM}`, where `{NUM}` is a unique number representing a given entity) each phrase classified as an entity chunk during *direct matching* or *domain phrases extraction*.

2. Replace with a unique predicate chunk placeholder (`PC{NUM}`, where `{NUM}` is a unique number representing a given predicate) each phrase classified as a predicate chunk during *direct matching* or *domain phrases extraction*.

If, as presented in Figure 9.3, any pair of phrases marked with *direct matching* and *domain phrases extraction* are overlapping spans of text, we merge them by considering a union of these spans and assigning a single chunk identifier. If any overlapping spans are of different chunk types (e.g., ReqTagger suggests an entity chunk, while *direct matching* a predicate chunk), the suggestion from *direct matching* has priority since it is based on the actual content of a given ontology.

As a result of this step, different similarly constructed CQs related to various domains, e.g., What kind of fruit is used in this shake?, What kind of company is led by prof. Jones?, What kind of database provides access to DBpedia? are transformed into the same CQ pattern candidate: What kind of EC1 PC1 EC2?.

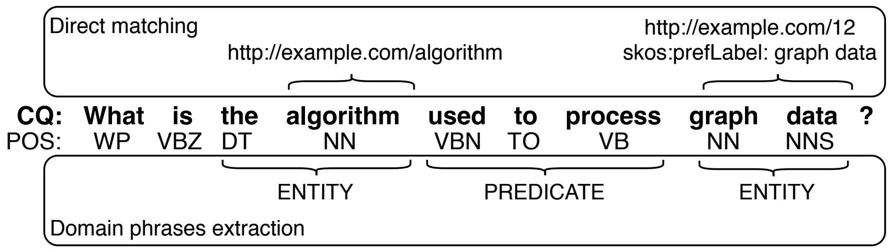

Direct matching
http://example.com/algorithm
http://example.com/12
skos:prefLabel: graph data
CQ: What is the algorithm used to process graph data ?
POS: WP VBZ DT NN VBN TO VB NN NNS
ENTITY
PREDICATE
ENTITY
Domain phrases extraction

Figure 9.3: The spans identified in a sample CQ using *direct matching* and *domain phrases extraction*.

9.2.3 Step 3: Closest known CQ pattern selection

The third step of the pipeline relates the CQ pattern candidate generated in the previous step to one of the known CQ patterns stored in *patterns and templates set* so that the CQ pattern with the closest meaning is chosen.

The natural language is so rich that it is likely to construct a CQ pattern candidate that is not included in the *patterns and templates set*. For this reason, we introduce a procedure for choosing the most similar one.

Let the *n-grams set* be a set of all possible n-grams generated from a tokenized sequence for a fixed n. Let *anygrams set* be a union of all *n-grams sets* for n between 1 and the number of tokens in a tokenized sequence.

Considering *I like her.* tokenized into: ['I', 'like', 'her', '.'], the follwing *n-grams sets* can be produced:

- 1-grams set – {'I', 'like', 'her', '.' }
- 2-grams set – {'I like', 'like her', 'her .'}
- 3-grams set – {'I like her', 'like her .'}
- 4-grams set – {'I like her .'}

As a consequence, an *anygram set* for this example is {'I', 'like', 'her', '.', 'I like', 'like her', 'her .', 'I like her', 'like her .', 'I like her .'}

Let *anygram-based Jaccard similarity* of two tokenized sequences be a measure calculating the Jaccard similarity between the *anygram sets* generated from both sequences. For the given two sets: A and B, the Jaccard similarity is calculated using the following equation:

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

.

The procedure selecting the most similar CQ pattern is defined as follows:

1. Select CQ patterns from *patterns and templates set* introducing the same number of EC and PC markers as the CQ pattern candidate constructed.
2. For every CQ pattern selected and a given CQ pattern candidate, calculate the *anygram-based Jaccard similarity* between the tokenized sequences.
3. Select the CQ pattern with the highest similarity.

This procedure can map an example CQ pattern candidate What EC1 PC1 EC2? into the CQ pattern Which EC1 PC1 EC2? being its paraphrase sharing the same meaning.

9.2.4 Step 4: SPARQL-OWL template(s) selection

In *patterns and templates set*, each CQ pattern is related to one or more SPARQL-OWL templates. These relations provide information on how formulations of CQs are realized in SPARQL-OWL queries in CQ2SPARQLOWL and BigCQ datasets. For example, the following CQ pattern starting with *does* was formalized as two kinds of `ASK` queries:

```
"Does EC1 PC1 EC2?": [
  "ASK WHERE { <EC1> rdfs:subClassOf [ rdf:type owl:Restriction ;
                                       owl:onProperty <PC1> ;
                                       owl:hasValue <EC2> ] . }",
  "ASK WHERE { <EC1> rdfs:subClassOf [ rdf:type owl:Restriction ;
                                       owl:onProperty <PC1> ;
                                       owl:someValuesFrom <EC2> ] . }"
]
```

Using *patterns and templates set* and the CQ pattern selected in the previous step, for further processing, we choose all SPARQL-OWL query templates a given CQ pattern relates to. From now on, each query template is processed independently in the subsequent processing steps. As a consequence, we can generate more than one SPARQL-OWL query recommendation.

9.2.5 Step 5: Phrase linking

This step aims to link the phrases extracted during the vocabulary detection step to appropriate ontological entities. The IRIs of the resources are then used to construct queries. As the *direct matching* already linked phrases, here, we link the output of *domain phrases extraction*.

Let a phrase vector $\mathbf{p}_i$ denote a vector calculated as an element-wise average of embeddings related to tokens in a given phrase p_i. The method of generating embedding can be chosen freely, but in our implementation, we used the BERT [38] model for this purpose. A phrase vector $\mathbf{p}_i$ can be then interpreted as a fixed-length dense representation of the meaning of p_i.‘

Let $L_{nthings}$ represent the set of all normalized labels related to classes in a given ontology. Similarly, let L_{nprop} represent the set of all normalized labels related to object and data properties.

First, we lemmatize each p_i (i.e., transform each inflected token into its base form [107]) and check if the lemmatized p_i is equal to any element from:

- $L_{nthings}$ – if p_i is classified as an entity chunk.
- L_{nprop} – if p_i is classified as a predicate chunk.

We also verify if the lemmatized p_i is equal to any lemmatized element from $L_{nthings}$ or L_{nprop}. If no match is found for a given p_i, we calculate the cosine similarity between $\mathbf{p}_i$ and all phrase vectors generated for labels in $L_{nthings}$ or L_{nprop} depending on whether p_i represents an entity or a relation. If a match is found, we link p_i with the matched entity and assign the maximum similarity value (1.0) between $\mathbf{p_i}$ and the phrase vector generated from the label of the linked entity.

If the most similar vector scores lower than the threshold hyperparameter γ, SeeQuery stops processing and informs the user that there is no ontological entity related to p_i modeled in the ontology and, as a consequence, no query can be generated.

Further actions depend on the SPARQL-OWL template used. If the query template introduces no PC markers, no properties have to be linked. In this case, we link each phrase p_i independently by choosing the ontological entity with the highest cosine similarity.

If PC markers are used in the query template, we cannot link ontological entities independently of each other, because properties may introduce domain and range restrictions. These influence the choice of ontological entities that serve as the subject and object of a given property as they should comply with the property restrictions.

Moreover, we observe that ontologies in CQ2SPARQLOWL provide rich class taxonomies and a few quite general properties. These properties have often labels that are different than phrases stated in CQs. It could be beneficial to use additional information sources such as axioms stated in the ontology to see which properties are used together with which classes or instances.

For these reasons, to handle the problem, we present a way of choosing the best translations of properties and their arguments jointly.

We assume that each predicate chunk PC representing a property in a SPARQL-OWL query template must have exactly two arguments representing the property's subject and object. Both arguments in the CQs are entity chunks, referenced to as EC_s and EC_o, respectively.

Let $translate(p_i)$ represent a function that for a given phrase p_i returns a set of ontological entities, the labels of which are possible translations of p_i. Denote by $domain(prop)$ the domain of the property $prop$ if it is specified or `owl:Thing` otherwise. Denote by $range(prop)$ the range of the property $prop$ if it is specified or `owl:Thing` otherwise. Denote by $super(C)$ the set of all superclasses of a given class C.

The following helper functions operate on the axioms defined in the ontology $\mathbb{O}$:

```
function example_score(subj,pred,obj)
    if $subj \in lhs(pred, oobj)$ return 1
    else if $super(subj) \cap lhs(pred, obj) \neq \emptyset$ return 0.75
    else if $super(obj) \cap rhs(pred, subj) \neq \emptyset$ return 0.75
    else if $rhs(pred, subj) \neq \emptyset \vee lhs(pred, obj) \neq \emptyset$ return 0.5
    else if $\exists subj' \in super(subj)\colon rhs(pred, subj') \neq \emptyset$ return 0.25
    else if $\exists obj' \in super(obj)\colon lhs(pred, obj') \neq \emptyset$ return 0.25
    else return 0
end
```

Algorithm 2: Computing an example score for a given triple of ontological entities $(subj, pred, obj)$. The score is based on the content of the ontology and rewards triples that co-occur in the axioms. Reprint from [188].

- $\tau(prop)$ returns a set of pairs of classes that are expected to be seen with property *prop*, in this sense that if $(C, D) \in \tau(prop)$ then for an individual i_1 of the class C, there should exist some (possibly anonymous) individual i_2 of the class D such that i_1 is connected to i_2 via *prop* [188].

$$\tau(prop) = \{(C, D)\colon (C \text{ SubClassOf: } prop \text{ some } D) \in \mathbb{O} \vee$$
$$\exists n \geq 1\colon [(C \text{ SubClassOf: } prop \text{ min } n\, D) \in \mathbb{O} \vee (C \text{ SubClassOf: } prop \text{ exactly } n\, D) \in \mathbb{O}]\}$$

- $lhs(prop, D) = \{C\colon (C, D) \in \tau(prop)\}$ is the set representing first elements of the pairs of τ, intuitively: these are known possible classes of the left-hand sides of assertions related to *prop* [188].
- $rhs(prop, C) = \{D\colon (C, D) \in \tau(prop)\}$ is the set of the second elements of the pairs of τ, corresponding to the set of classes of the right-hand sides of assertions related to *prop* [188].

For a given string marked as a predicate chunk PC, its subject EC_s, and object EC_o, we compute the translation candidates set CT consisting of translation triples that do not contradict the domain and range restrictions of each PC translation candidate.

$$subjects(prop) = \{s \in translate(EC_s)\colon domain(prop) \in \{s \cup super(s)\}\}$$
$$objects(prop) = \{o \in translate(EC_o)\colon range(prop) \in \{o \cup super(o)\}\}$$
$$CT = \bigcup_{prop \in translate(PC)} subjects(prop) \times \{prop\} \times objects(prop)$$

From the candidates in CT, representing all possible translations of PC, EC_s and EC_o jointly, we select the best translation according to the scoring function:

$$best = \underset{(subj,pred,obj) \in CT}{\arg\max} \{\theta \cdot \texttt{label\_sim}(subj, pred, obj) + (1 - \theta) \cdot \texttt{example\_score(}subj,\ pred,\ obj\texttt{)}\}$$

where θ represents a trade-off parameter, `example_score` is a function defined in Algorithm 2 that rewards triples co-occurring among axioms in an ontology, and `label_sim` is the average cosine similarity of the chunks and normalized entity labels of their candidate translations:

$$\texttt{sim}(entity, phrase) = \cos(\texttt{embedding}(\texttt{label}(entity)), \texttt{embedding}(phrase))$$

$$\texttt{label\_sim}(subj, pred, object, EC_s, EC_o, PC) = \frac{1}{3}\texttt{sim}(subj, EC_s) + \texttt{sim}(pred, PC) + \texttt{sim}(obj, EC_o)$$

Function `label`(e) produces the normalized entity label from the entity e and `embedding`(p) is a function generating phrase vector for a phrase p. In short, the process can be summarized as: from all possible candidates for the property and its arguments translations, consider those that do not contradict the domain and range restrictions of the property and choose top-scored translations based on a scoring function aggregating the phrase embedding similarity and co-occurrence from the axioms stated in an ontology.

9.2.6 Step 6: Query (queries) filling

In Step 4, we selected ≥ 1 query templates, each of which contains chunk markers linked with ontological entities in Step 5. In this step, we substitute the EC and PC markers with actual IRIs extracted from the linked ontological entities. We iterate over each chunk and replace it with the chosen IRI enclosed with angle brackets.

As a result, each query template is materialized into a SPARQL-OWL query representing a query recommendation for a given CQ.

9.3 Evaluation

9.3.1 Evaluation procedure

We implemented SeeQuery in Python and published it online [1] under the MIT licence.

We used the CQ2SPARQLOWL dataset to choose the best values of hyperparameters. After 30 runs of SeeQuery, using a random set of hyperparameters in each run, we observed the γ similarity threshold set to 0.82 and 0.0 for ECs and PCs, respectively, and the θ tradeoff parameter set to 0.6 gave the best quality of results.

To generate context-aware embeddings, we used BERT [38] in its base and uncased version [2]. The embeddings are generated in the following way:

- Vectorize the CQ using BERT. As a result, each token is mapped to its context-aware embedding. Then, for each phrase p_i marked with ReqTagger, collect embeddings assigned to tokens in p_i and apply mean pooling to obtain a single phrase vector $\mathbf{p_i}$.
- For each ontological entity label l, form a label in context cl by using the CQ and l to fill the template `[CQ] How about [l]?`. For example, for the ontological entity label l set to `Weka` and the CQ defined as Does Matlab provide clustering algorithms?, the following cl is produced: Does Matlab provide clustering algorithms? How about Weka?. Then, vectorize cl, collect embeddings related to tokens from l and average them using mean pooling.

The rationale behind generating cl before calculating embeddings is to add context to the label l. As BERT is pretrained on a multidomain text corpus, it can only provide good representations of polysemic words if the context disambiguates their meaning. In the example above, Weka can be a piece of software or a bird from New Zealand. If BERT sees other tokens from the CQ during vectorization, it generates an embedding of Weka that is similar to other pieces of software.

The evaluation was performed on every CQ from the evaluation set. SeeQuery processed each CQ, and an expert verified the output deciding if the query can be generated for a given CQ, and if the output produced is correct or not. We did not state the predefined golden standard to evaluate the method automatically as frequently, multiple semantically equivalent queries can be provided (e.g., queries with different filter orders).

After processing all 62 CQs from the evaluation set, we observed that 6 of them were translated with SeeQuery to exactly two query recommendations. No CQ was translated to 3 or more SPARQL-OWL queries. For this reason, we decided to consider the output of SeeQuery as correct if any of the generated recommendations are correct. The evaluation summary, which shows that SeeQuery provides correct recommendations for 46 out of 62 CQs (74%), can be found in Table 9.2.

[1] `https://github.com/dwisniewski/SeeQuery`

[2] `https://huggingface.co/bert-base-uncased`

Table 9.2: Evaluation scores on different ontologies. The columns denote (i) the fraction of generated outputs considered correct, (ii) the fraction of correctly identified untranslatable questions.

Ontology	Correct outputs	Correctly chosen untranslatable
Pizza	33/42 **(78.57%)**	25/26 **(96.15%)**
TrhOnt	13/20 **(65%)**	9/11 **(81.8%)**

9.3.2 Error analysis

There are interesting groups of errors that we observed among the 16 CQs the SeeQuery translates incorrectly.

Lack of required SPARQL-OWL template The *patterns and templates set* extracted from CQ2SPARQLOWL and BigCQ does not cover all possible queries. For example, no query template using `owl:disjointWith` is provided in *patterns and templates set*. However, 3 CQs in the Pizza ontology: Which pizzas do not have nuts?, Which pizzas contain prawns but not anchovy? and Which are the nut free pizzas? all require `owl:disjointWith` in their translations.

CQs with too many chunks Consider the CQ from TrhOnt: Which are the conditions that a patient must fulfill in order to be in a phase of a treatment protocol?. From that CQ, SeeQuery, in step 2 of the pipeline generates a CQ pattern candidate that uses 4 entity chunks (the conditions, a patient, a phase, a treatment protocol) and 2 predicate chunks (in order to, be in). However, no CQ pattern in *patterns and templates set* introduces the same number of chunks of given types, so it is not possible to choose the closest known CQ pattern. As a consequence, the method returns no query.

Phrase linking failures Consider two CQs from Pizza: Can you have a pizza with any combination of toppings? and Are different bases available?. In both cases, no queries are produced because SeeQuery assumes there is no required vocabulary in the Pizza ontology provided. However, `pizza topping` and `pizza base` are ontological entity labels represented in the ontology and these should be linked to toppings and bases, respectively. Unfortunately, they are not linked, because the similarity between `pizza topping` and toppings as well as between `pizza base` and bases is scored below the threshold γ. It is not clear, however, how to determine the equivalence between these phrases. The assumption of equivalence between `pizza topping` and toppings is reasonable in the domain of Pizza, but in general, the noun topping can also represent a distinct (top) part of a thing. Similarly, the equivalence between `pizza base` and bases can be considered reasonable in the context of the knowledge modeled in the Pizza ontology, but if one adds an entity representing the base salary of a pizza-maker, the term bases in the CQ becomes ambiguous.

Such ambiguity is observed in the case of CQ Which pizzas are spicy?. The Pizza ontology provides two classes introducing the word spicy: `Spicy Topping` and `Spicy Pizza` and the query produced wrongly asks about the subclasses of both `Pizza` and `SpicyTopping`. Such a query is incorrect, as there is no possibility of being both a pizza and a topping in this domain.

Expected relation between one chunk in a CQ and many ontological entities An example of such an error is the CQ Which body part does an auxiliary movement refer to?, where the phrase refer to maps to a chain of object properties: `has component` and `has location`. Another example is the CQ *Which are the nut free pizzas?*, where *the nut free pizzas* should be mapped to `Pizza` subclasses that are disjoint with classes with nut topping defined. SeeQuery fails to produce correct outputs for both CQs, as there is no functionality of mapping a single phrase to complex relations between entities.

Wrong interpretation of CQs Consider two CQs: Which range of movement does a movement cover? and Are anchovies and capers used together?. These introduce phrases such as range and used together. SeeQeury marks both phrases as chunks and tries to link those to ontological entities. However, it should not interpret these phrases explicitly but rather

- used together should be interpreted as a phrase that determines the form of query template instead of pointing to an ontological entity label. It should be used to construct a query asking for pizzas that contain both ingredients at once.
- range should be interpreted as a reference to a pair of properties that in the TrhOnt define the minimum and the maximum value of the movement.

9.4 Discussion

SeeQuery's limitations The limited number of examples of CQs translated into SPARQL-OWL queries in CQ2SPARQLOWL was the reason for making SeeQuery template-based. However, this approach causes several limitations:

- Since SeeQuery fills templates extracted from predefined pairs of CQs and SPARQL-OWL queries examples, it cannot express a query constructed differently from the predefined templates. Because ontologies are modeled using various modeling styles, new patterns and templates should be added to increase the coverage. For this reason, the implementation of SeeQuery makes it easy to extend the *patterns and templates set*, which is stored in a JSON file.
- As we showed in the error analysis section, SeeQuery fails to map a single phrase to an expression introducing multiple ontological entities. This task is very hard to solve as complete knowledge about a given domain is required to assess that, e.g., *a dog* is not equivalent to *a mammal that barks* because foxes also bark.
- The heuristic of searching for a translation of a predicate chunk with its subject and object jointly may fail in complex queries. If a property chain is used, the object of a property may not be a named class. Similarly, in some complex scenarios, an object of one predicate chunk may serve as a subject for another predicate chunk. Fortunately, the evaluation shows that such complex representations are rare.

Design choices made In the SeeQuery implementation, we used BERT to calculate embeddings for tokens. The rationales behind this choice are:

1. BERT achieves state-of-the-art scores on many NLP tasks.
2. BERT, as opposed to methods generating static embeddings, provides context-aware representations. They are especially useful to process polysemic words.
3. BERT uses WordPiece tokenization, which handles arbitrary texts. Even if some word was not seen by BERT during training, the vector representation of the word can be still calculated by splitting the word into subword units known by the tokenizer and aggregating embeddings calculated for these.

The delegation of entity linking to step 5 of the pipeline rather than performing it in step 1 is motivated by the need for a joint search for translations if predicate chunks and their subjects and objects are found.

Using threshold values for entity linking but not for the closest known pattern selection is motivated by the fact that if the vocabulary is provided (entities can be linked to phrases), the query can be most probably generated. In this scenario, we prefer to generate a query recommendation even if the closest known CQ pattern has a low similarity score to the CQ pattern candidate constructed from the input CQ. Even if the query recommendations are wrong, the IRIs of the linked phrases may guide the engineer on how to fix the recommendations.

9.5 Summary

The main goal of this chapter was to introduce a method for generating SPARQL-OWL query recommendations from CQs and ontologies. We described SeeQuery – a template-based method, which can assist the process of verifying if an ontology is complete and correct.

The SeeQuery is based on the biggest to date dataset of the real-world: ontologies, CQs, and their formalizations in SPARQL-OWL, as well as on the largest automatically generated silver-standard dataset of CQ patterns mapped to SPARQL-OWL templates. Those datasets introduce knowledge that generalizes well. The evaluation shows that, even for new domains, the method presents decent performance.

However, ontologies differ in the modeling styles used to construct them, so the *patterns and templates* set should be expanded as new CQs translated to SPARQL-OWL queries dataset will be published.

Chapter 10

Presuppositions and Test-Driven Development of ontologies

Questions (e.g., competency questions) hold implicit assumptions that have to be satisfied to obtain meaningful answers [148, 37]. For example, the following question: Did Dawid finish his Ph.D. thesis? implicitly assumes that Dawid was in the process of preparing his Ph.D. thesis. We refer to such implicit assumptions about the world or background beliefs that relate to an utterance whose truth is taken for granted in discourse as presuppositions [2]. These were introduced and analyzed in the area of linguistics called pragmatics.

In this chapter, we analyze how CQs and their SPARQL-OWL formalizations can be enriched with presuppositions handling. We show that we can automate presupposition testing using ASK-type SPARQL-OWL queries, which can be generated automatically via template selection and filling. Moreover, we discuss how presuppositions can be used as testing artifacts and embedded in the Test-Driven Development of ontologies pipeline [134].

The main contributions presented in this chapter are:

1. A way of formalizing presuppositions using SPARQL-OWL queries,
2. A presupposition-based testing model to verify CQs that can be formalized as SELECT queries,
3. A way of integrating presupposition tests into test-driven ontology development,
4. An extension of CQ2SPARQLOWL providing queries for automated presupposition checking.

10.1 Presuppositions among CQs

Questions that begin with one of the following words: when, where, what, who, whom, whose, which, why, and how are called wh- questions. All of them contain presuppositions that assume some objects fulfill the predicate stated in the question [104]. To generate a presupposition out of such a question, one needs to replace the wh-word with an appropriate indefinite pronoun, e.g., a question Who won the match yesterday? presupposes that Someone won the match yesterday [104]. Moreover, the presupposition can be denied, e.g., Noone won the match yesterday [104].

Considering CQs, many of them are wh- questions. In the context of CQ2SPARQLOWL, 189 out of 234 CQs represent this category. These always explicitly or implicitly assume a domain for their answers [134]. Also, there must be elements capable of fulfilling the predicate (a positive presupposition), yet the domain elements do not necessarily fulfill the predicate (a negative presupposition) [134].

10.2 Ontology testing using presuppositions

Presuppositions can be used to test the knowledge modeled in an ontology to verify if the answers obtained from SPARQL-OWL queries are meaningful. To define the testing model, let us define the following symbols:

- $\mathbb{O}$ – used to denote an ontology expressed in OWL 2 [119],
- $CE1$ and $CE2$ – used to denote class expressions (e.g., named classes),
- Q – used to represent a SPARQL-OWL query that is a formalization of a given CQ,
- PQ – used to denote a *presupposition query* – a query of the form `ASK WHERE {CE rdfs:subClassOf owl:Nothing}` that we use to check the satisfiability of CE,
- PQ^+ and PQ^- – used to denote a positive and negative presupposition query, respectively,
- $\mu(PQ)$ and $\mu(Q)$ – used to denote the answer(s) to PQ and Q, respectively.

To check the satisfiability of CE, we can query the ontology using the query PQ: `ASK WHERE {CE rdfs:subClassOf owl:Nothing}` and interpret the answer given by the ontology. If the logical consequence of the ontology $\mathbb{O}$ is `C rdfs:subClassOf owl:Nothing`, then the returned answer $\mu(PQ) =$ **true**. In this case, no instance of CE exists so that the presupposition is not satisfied. Alternatively, if $\mu(PQ) =$ **false**, there are instances of CE allowed in $\mathbb{O}$ so that the presupposition is satisfied.

These two scenarios are visualized in Figure 10.1. We refer to the process of interpreting the result of PQ as a presupposition test.

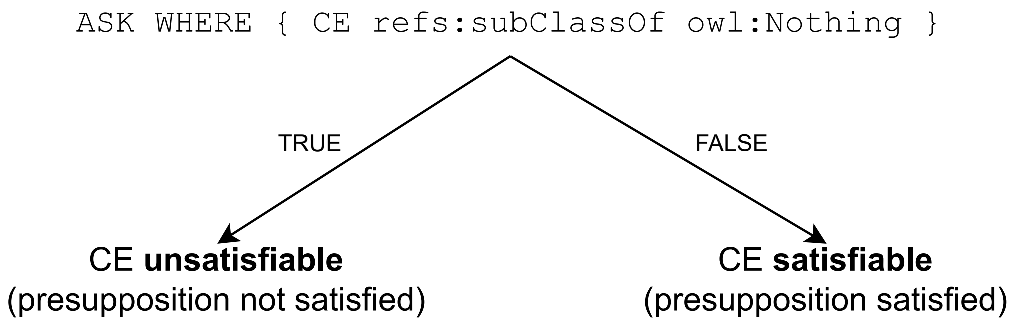

ASK WHERE { CE refs:subClassOf owl:Nothing }
TRUE
FALSE
CE **unsatisfiable**
(presupposition not satisfied)
CE **satisfiable**
(presupposition satisfied)

Figure 10.1: A presupposition query and the two possible interpretations of the result.

Ren et al. [148] noticed that ASK queries do not have presuppositions because they only check whether there exists an answer satisfying a constraint [134]. For this reason, the testing model we propose is defined over SELECT queries and open questions they are associated with.

Considering an example CQ Which software implements clustering algorithms?, two kinds of presuppositions can be identified that both should be satisfied to answer the question in a meaningful way:

- Positive presupposition: There may exist a piece of software that implements clustering algorithms.
- Negative presupposition: There may exist a piece of software that does not implement clustering algorithms.

If a CQ can be formalized as a SELECT-type SPARQL-OWL query, the query can be interpreted as restricting a class expression $CE1$ with another class expression $CE2$ [134]. In that interpretation, if the positive presupposition is satisfied, it indicates that objects that are both $CE1$ and $CE2$ may exist (e.g., software that implements clustering algorithms). Similarly, the negative presupposition is satisfied if there may exist objects that are $CE1$ and not $CE2$ (e.g., software that does not implement clustering algorithms) [134].

If the positive or negative presupposition is not satisfied, it means that the ontology determines the answer:

- If the positive presupposition is not satisfied, the intersection of $CE1$ and $CE2$ is necessarily empty (e.g., pieces of software that implement clustering algorithms cannot exist).
- If the negative presupposition is not satisfied, the intersection of $CE1$ and $CE2$ is equal to $CE1$ (e.g., each piece of software implements clustering algorithms).

If both positive and negative presuppositions are satisfied, it is reasonable to query $\mathbb{O}$ with Q.

Formally, we can define the model for testing SELECT queries as follows: Given an ontology $\mathbb{O}$ that is coherent (i.e., all named classes in $\mathbb{O}$ are satisfiable) and consistent (i.e., $\mathbb{O}$ has a model – there is an interpretation where all axioms hold), a CQ formalized as a SPARQL-OWL query Q asking about the intersection of class expressions $CE1_i$ and $CE2_i$ and its corresponding positive and negative presupposition queries PQ^+ and PQ^-,the result of testing PQ^+, PQ^- and Q against the ontology $\mathbb{O}$ can be expressed as:

$$test_O(Q) = \begin{cases} \mu(Q) = \emptyset & \text{if } \mu(PQ^+) = \text{true (i.e., unsatisfiable)} \\ \mu(Q) = CE1_i & \text{if } \mu(PQ^-) = \text{true (i.e., unsatisfiable)} \\ \text{compute the answer to Q} & \text{if } \mu(PQ^+) = \text{false and } \mu(PQ^-) = \text{false} \end{cases}$$

10.3 Integration of presupposition tests into TDD

The test-driven ontology authoring was proposed as a promising implementation of a test-first approach that is already well established in the field of software engineering [10]. It was intended to reduce authoring time and increase authoring efficiency [34]. We propose an extension of the TDD workflow proposed in [34] to handle question-answerability checking by incorporating presupposition tests and providing an automatic formalization of CQs into SPARQL-OWL. In Figure 10.2, we present the TDD workflow, including additional steps marked with grey boxes. The first and fourth of them can be handled automatically using SeeQuery introduced in Chapter 9. Considering CQs related to SELECT queries, the steps in the second and third boxes can be automated using positive and negative presupposition tests.

If SeeQuery identifies the required vocabulary (box named vocabulary is present), both positive (box named positive presupposition is satisfied), and negative (box named negative presupposition is satisfied) presupposition tests pass, and SeeQuery produces a query (box named translate CQ into SPARQL-OWL), the subsequent steps may be performed.

To make presupposition tests automated, we enriched SPARQL-OWL query templates collected in CQ2SPARQLOWL [189, 135]. For every template related to the SELECT query, we constructed SPARQL-OWL query templates related to presupposition tests. This way, we can fill the presupposition query templates similarly to "main" templates used by SeeQuery, since both kinds share the same placeholders. The dataset of presupposition query templates assigned to query templates from CQ2SPARQLOWL is provided in Appendix E. We listed only those SPARQL-OWL templates that require one or more presupposition tests.

An example of a SPARQL-OWL query template from CQ2SPARQLOWL enriched with presupposition query templates is:

```
SELECT DISTINCT * WHERE {
    ?x rdfs:subClassOf <EC1>, [
        a owl:Restriction ;
        owl:onProperty <PC1> ;
        owl:someValuesFrom <EC2> ] },
```

for which the positive presupposition query is:

```
ASK WHERE {
    [a owl:Class; owl:intersectionOf(
        <EC1>
        [a owl:Restriction;
         owl:onProperty <PC1>;
         owl:someValuesFrom <EC2>
        ]
     )] rdfs:subClassOf owl:Nothing }
```

and the negative presupposition query is:

```
ASK WHERE {
    [a owl:Class; owl:intersectionOf(
        <EC1>
        [a owl:Restriction;
         owl:onProperty <PC1>;
         owl:allValuesFrom [a owl:Class; owl:complementOf <EC2>]
      ]
     )] rdfs:subClassOf owl:Nothing }
```

10.4 Conclusions

The idea of incorporating presupposition tests into the test-driven development of ontologies helps to understand the reasons behind the answers provided by the ontology tested with queries. In consequence, the outputs returned by the queries can be interpreted according to the engineer's intents. The CQs can be meaningfully answered if the appropriate vocabulary is present in the ontology and if the presuppositions are satisfied.

The extension of the TDD pipeline and a way to automate each of the added steps may improve the ontology authoring speed and make the ontology quality assurance easier.

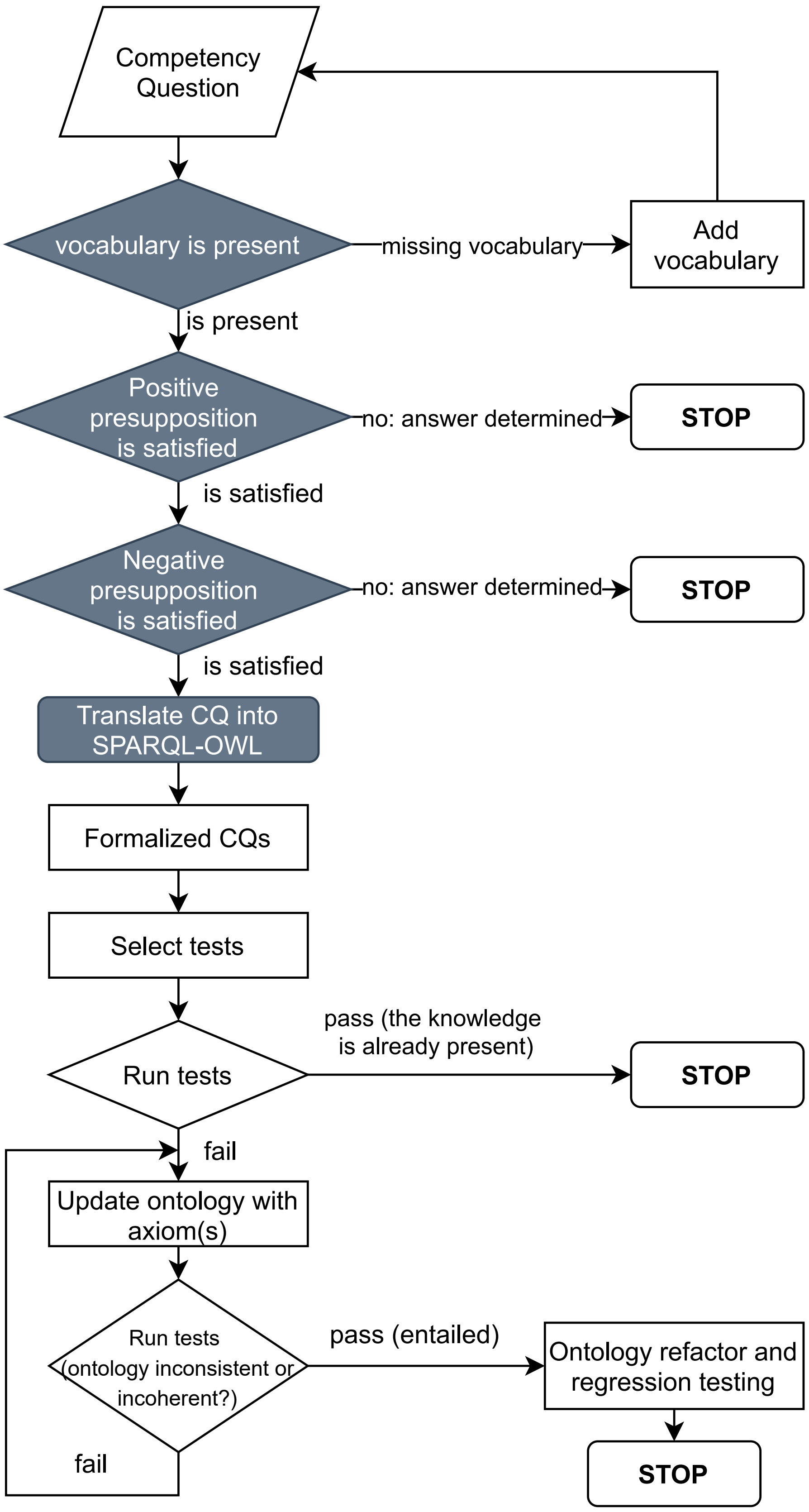

Competency Question
vocabulary is present
missing vocabulary
Add vocabulary
is present
Positive presupposition is satisfied
no: answer determined
STOP
is satisfied
Negative presupposition is satisfied
no: answer determined
STOP
is satisfied
Translate CQ into SPARQL-OWL
Formalized CQs
Select tests
Run tests
pass (the knowledge is already present)
STOP
fail
Update ontology with axiom(s)
Run tests (ontology inconsistent or incoherent?)
pass (entailed)
Ontology refactor and regression testing
fail
STOP

Figure 10.2: The TDD workflow with presupposition tests included. The white shapes represent the simplified, preexisting steps and the grey ones represent our contribution.

Chapter 11

Summary

11.1 Answers to the research questions

In this section, we provide the answers to the research questions defined in Chapter 1.

RQ1 – Are there recurring patterns among CQs, SPARQL-OWL queries, and between CQs and SPARQL-OWL queries? Based on our research, we can answer this question positively. In Chapter 6, we analyzed a dataset of 234 CQs, 131 of which are translated into SPARQL-OWL queries. We showed that among the CQs, there are 106 domain-independent CQ patterns, 6 of which are shared among more than one ontology. Using CQ pattern normalization, we generated 81 higher-level CQ patterns, 8 of which are shared by multiple ontologies. We also constructed domain-independent SPARQL-OWL signatures to show that there are recurring patterns among the queries as well. We found 6 SPARQL-OWL signatures shared among multiple ontologies and 9 of them among more than one query. The mapping between CQ patterns and SPARQL-OWL templates constructed from CQ2SPARQL-OWL and BigCQ was used as the main component of SeeQuery. As that mapping determined the forms of recommended queries and the evaluation scores show that SeeQuery generalizes well to new ontologies and CQs, we conclude that there are recurring patterns between CQs and SPARQL-OWL queries.

RQ2 – How to automate the glossary of terms extraction? In Chapter 7, we proposed two methods for the glossary of terms extraction. One of them is a CRF-based model trained over a set of CQs annotated with candidates for classes and relations, and the other uses hand-crafted rules. The evaluation of the methods shows that the rule-based approach gives better results and can generalize to handle requirements expressed as sentences. As the rule-based tagger achieves promising evaluation scores, we recommend it as a tool for an automatic glossary of terms extraction.

RQ3 – How to construct pairs of CQs and SPARQL-OWL queries automatically based on ontology axioms? In Chapter 8, we showed a method based on axiom verbalization and linguistic transformations that can transform axiom shapes into pairs of CQ patterns and SPARQL-OWL templates automatically. We used the method to construct BigCQ – a dataset of 77,575 CQ patterns mapped to 549 SPARQL-OWL templates. These patterns and templates can be filled with labels and IRIs extracted from a given ontology to provide CQs and SPARQL-OWL queries.

RQ4 – How to construct SPARQL-OWL query recommendations from CQs automatically? In Chapter 9, we proposed a method consisting of a pipeline defining 6 processing steps used to transform CQs into SPARQL-OWL query recommendations automatically. The evaluation procedure showed that this method, based on pattern matching and template filling, generalizes to ontologies and CQ sets that are different than those used to build the pipeline.

RQ5 – How to integrate the automatic translation of CQs into SPARQL-OWL with Test-Driven Development of ontologies? In Chapter 10, we described how our method for translating CQs into SPARQL-OWL queries could be integrated into Test-Driven Development of ontologies and extended with so-called presupposition tests. We visualized how to integrate our approach with TDD in Figure 10.2.

11.2 Conclusions

In this dissertation, we focused on the problems of automated competency question handling in ontology development. We analyzed how CQ handling can be automated to:

- construct glossaries of terms that provide terms to model in an ontology,
- verify the quality of a given ontology by counting how many CQs, formalized using SPARQL-OWL, can be correctly answered.

We proposed two datasets. The first – smaller one – CQ2SPARQLOWL, consists of real-world ontologies, and CQs translated into SPARQL-OWL. The second – much larger – BigCQ, which is inspired by CQ2SPARQLOWL, is constructed automatically from frequent axiom patterns.

We focused on automating both described scenarios involving CQs:

1. Automating glossary of terms construction – There are two methods introduced in Chapter 7 that can be used to extract candidates for classes, instances, and properties without human intervention. We showed that the rule-based approach provides candidates of better quality. These methods allow engineers to limit the time spent during the tedious process of extracting vocabulary from CQs.
2. Automating formalization of CQs into SPARQL-OWL – The method introduced in Chapter 9 recommends SPARQL-OWL translations for CQs. If more than one query is recommended, an engineer can select the most appropriate one. Assuming that a comprehensive set of CQs is provided, if all CQs can be formalized as queries and the queries return correct answers, one can consider the ontology correct and complete. With the use of SeeQuery, the engineer does not have to be specialized in SPARQL-OWL, as instead of constructing the query, they have only to select an appropriate recommendation when needed. Moreover, the engineer does not have to check which ontological entities should be used in the query as the method can determine if the expected vocabulary is modeled and link phrases extracted from CQs to appropriate entities to obtain their IRIs.

Finally, we propose the integration of the translation method into the existing Test-Driven Development approach to ontology development.

We hope that our tools and analyses will make the competency question-based ontology authoring easier and quicker, and the idea of using CQs in ontology development will become more popular.

11.3 Future work

There are several directions for further research we would like to investigate in the future:

1. As more ontologies with CQs stated for their terminological parts become available, our goal is to collect them to expand CQ2SPARQLOWL. This way, we make the *patterns and templates set* cover more CQ formulations and more query forms.

2. The large-scale datasets of ontologies with CQs translated into queries that are created using the method introduced in Chapter 8 may fuel deep learning-based translators that may be an interesting alternative for a template-based SeeQuery.

3. Currently, the glossary of terms extractors we propose process each CQ independently. However, different CQs may mention the same entity using various grammatical forms (plurals, different verb tenses) or synonyms. It would be interesting to group those forms and suggest only a single, canonical entity form that should be modeled in the ontology.

4. The entity linking procedure defined for SeeQuery works well for the most common graph patterns. However, there are cases (e.g., property chains) that cannot be handled properly now. In the future, we should focus on creating a more robust entity linking method to support more complex BGPs.

5. SeeQuery recommends queries based on the way knowledge was modeled in other ontologies. However, it could use the information on how axioms are constructed in the queried ontology to rank the recommendations so that the most probable ones are high on the list.

6. In CQ2SPARQLOWL, we found that approximations relating a single phrase extracted from a CQ to a complex class expression are frequently used. In this dissertation, we rejected such cases. However, we would like to investigate the problem of discovering equivalence between phrases and complex class expressions.

7. As the machine learning-based tagger proposed in Chapter 7 was created before models generating contextualized word representations (e.g., BERT) became widely used, it would be interesting to extend the tagger to include such representations as additional features.

Bibliography

[1] Amir D Aczel and Jayavel Sounderpandian. *Complete business statistics.* Irwin/McGraw Hill Boston, MA, 1999.

[2] Adrian Akmajian, Richard A. Demers, and Robert M. Harnish. *Linguistics: An Introduction to Language and Communication.* MIT Press, Cambridge, Massachusetts, 1979.

[3] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *CoRR*, abs/2102.13027, 2021.

[4] Tahani Alsubait. *Ontology-based multiple-choice question generation.* PhD thesis, University of Manchester, UK, 2015.

[5] Takatura Ando. The origin of the concept of metaphysics. In *Metaphysics: A Critical Survey of its Meaning*, pages 3–16. Springer Netherlands, Dordrecht, 1974.

[6] Grigoris Antoniou and Frank van Harmelen. Web ontology language: OWL. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 67–92. Springer, 2004.

[7] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, USA, 2003.

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[9] GH. Bakir, T. Hofmann, B. Schölkopf, AJ. Smola, B. Taskar, and SVN. Vishwanathan. *Predicting Structured Data.* Advances in neural information processing systems. MIT Press, Cambridge, MA, USA, September 2007.

[10] Beck. *Test Driven Development: By Example.* Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[11] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[12] Idoia Berges, David Antón, Jesús Bermúdez, Alfredo Goñi, and Arantza Illarramendi. Trhont: building an ontology to assist rehabilitation processes. *Journal of Biomedical Semantics*, 7:60, 2016.

[13] Amaia Bernaras, Iñaki Laresgoiti, and Jose Manuel Corera. Building and reusing ontologies for electrical network applications. In Wolfgang Wahlster, editor, *12th European Conference on Artificial Intelligence, Budapest, Hungary, August 11-16, 1996, Proceedings*, pages 298–302. John Wiley and Sons, Chichester, 1996.

[14] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[15] Camila Bezerra, Filipe Santana, and F. Freitas. CQChecker: A tool to check ontologies in OWL-DL using competency questions written in controlled natural language. *Learning and Nonlinear Models*, 12:115–129, 2014.

[16] Steven Bird and Edward Loper. NLTK: The natural language toolkit. In *Proceedings of the ACL Interactive Poster and Demonstration Sessions*, pages 214–217, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[17] Christopher M. Bishop. *Pattern recognition and machine learning, 5th Edition*. Information science and statistics. Springer, 2007.

[18] Eva Blomqvist, Azam Seil Sepour, and Valentina Presutti. Ontology testing - methodology and tool. In Annette ten Teije, Johanna Völker, Siegfried Handschuh, Heiner Stuckenschmidt, Mathieu d'Aquin, Andriy Nikolov, Nathalie Aussenac-Gilles, and Nathalie Hernandez, editors, *Knowledge Engineering and Knowledge Management - 18th International Conference, EKAW 2012, Galway City, Ireland, October 8-12, 2012. Proceedings*, volume 7603 of *Lecture Notes in Computer Science*, pages 216–226. Springer, 2012.

[19] Bernd Bohnet, Ryan T. McDonald, Gonçalo Simões, Daniel Andor, Emily Pitler, and Joshua Maynez. Morphosyntactic tagging with a meta-BiLSTM model over context sensitive token encodings. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 2642–2652. Association for Computational Linguistics, 2018.

[20] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

[21] Gemma Boleda. Distributional semantics and linguistic theory. *Annual Review of Linguistics*, 6(1):213–234, 2020.

[22] Dan Brickley and Ramanathan Guha. RDF schema 1.1. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[23] Charles George Broyden. The convergence of a class of double-rank minimization algorithms 1. general considerations. *IMA Journal of Applied Mathematics*, 6(1):76–90, 1970.

[24] Elena Cabrio, Julien Cojan, Fabien Gandon, and Amine Hallili. Querying multilingual DBpedia with QAKiS. In Philipp Cimiano, Miriam Fernández, Vanessa López, Stefan Schlobach, and Johanna Völker, editors, *The Semantic Web: ESWC 2013 Satellite Events - ESWC 2013 Satellite Events, Montpellier, France, May 26-30, 2013, Revised Selected Papers*, volume 7955 of *Lecture Notes in Computer Science*, pages 194–198. Springer, 2013.

[25] Nicola De Cao, Gautier Izacard, Sebastian Riedel, and Fabio Petroni. Autoregressive entity retrieval. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[26] Gavin Carothers and Eric Prud'hommeaux. RDF 1.1 turtle. W3C recommendation, W3C, February 2014. http://www.w3.org/TR/2014/REC-turtle-20140225/.

[27] Eugene Charniak, Curtis Hendrickson, Neil Jacobson, and Mike Perkowitz. Equations for part-of-speech tagging. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, AAAI'93, page 784–789. AAAI Press, 1993.

[28] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734. ACL, 2014.

[29] Philipp Cimiano, Peter Haase, Jörg Heizmann, Matthias Mantel, and Rudi Studer. Towards portable natural language interfaces to knowledge bases - the case of the ORAKEL system. *Data and Knowledge Engineering*, 65(2):325–354, 2008.

[30] Philipp Cimiano, Vanessa Lopez, Christina Unger, Elena Cabrio, Axel-Cyrille Ngonga Ngomo, and Sebastian Walter. Multilingual question answering over linked data (qald-3): Lab overview. In Pamela Forner, Henning Müller, Roberto Paredes, Paolo Rosso, and Benno Stein, editors, *Information Access Evaluation. Multilinguality, Multimodality, and Visualization*, pages 321–332, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[31] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yangqiu Song, Seung-won Hwang, and Wei Wang. KBQA: learning question answering over QA corpora and knowledge bases. *Proceedings of the VLDB Endowment*, 10(5):565–576, 2017.

[32] Enrico Daga, Eva Blomqvist, Aldo Gangemi, Elena Mon-tiel, Nadejda Nikitina, Valentina Presutti, and Boris Villazón-Terrazas. NeOn D2. 5.2 pattern based ontology design: Methodology and software support. Technical Report D2. 5.2, NeOn Consortium, 2010.

[33] Danica Damljanovic, Milan Agatonovic, Hamish Cunningham, and Kalina Bontcheva. Improving habitability of natural language interfaces for querying ontologies with feedback and clarification dialogues. *Journal of Web Semantics*, 19:1–21, 2013.

[34] Kieren Davies, C. Maria Keet, and Agnieszka Lawrynowicz. More effective ontology authoring with test-driven development and the TDDonto2 tool. *International Journal on Artificial Intelligence Tools*, 28(7):1950023:1–1950023:25, 2019.

[35] Marie-Catherine de Marneffe, Christopher D. Manning, Joakim Nivre, and Daniel Zeman. Universal Dependencies. *Computational Linguistics*, 47(2):255–308, 07 2021.

[36] María del Carmen Suárez-Figueroa, Eva Blomqvist, Mathieu d'Aquin, Mauricio Espinoza, Asunción Gómez-Pérez, Holger Lewen, Igor Mozetic, Raul Palma, Maria Poveda, Margherita Sini, Boris Villazon, Fouad Zablith, and Martin Dzbor. D5.4.2 revision and extension of the neon methodology for building contextualized ontology networks, February 2009.

[37] Matt Dennis, Kees van Deemter, Daniele Dell'Aglio, and Jeff Z. Pan. Computing authoring tests from competency questions: Experimental validation. In Claudia d'Amato, Miriam Fernández, Valentina A. M. Tamma, Freddy Lécué, Philippe Cudré-Mauroux, Juan F. Sequeda, Christoph Lange, and Jeff Heflin, editors, *The Semantic Web - ISWC 2017 - 16th International Semantic Web Conference, Vienna, Austria, October 21-25, 2017, Proceedings, Part I*, volume 10587 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2017.

[38] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[39] Dennis Diefenbach, Vanessa López, Kamal Deep Singh, and Pierre Maret. Core techniques of question answering systems over knowledge bases: a survey. *Knowledge and Information Systems volume*, 55(3):529–569, 2018.

[40] Xinya Du, Junru Shao, and Claire Cardie. Learning to ask: Neural question generation for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1342–1352, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[41] Nan Duan, Duyu Tang, Peng Chen, and Ming Zhou. Question generation for question answering. In Martha Palmer, Rebecca Hwa, and Sebastian Riedel, editors, *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 866–874. Association for Computational Linguistics, 2017.

[42] Mohnish Dubey, Sourish Dasgupta, Ankit Sharma, Konrad Höffner, and Jens Lehmann. AskNow: A framework for natural language query formalization in SPARQL. In Harald Sack, Eva Blomqvist, Mathieu d'Aquin, Chiara Ghidini, Simone Paolo Ponzetto, and Christoph Lange, editors, *The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Proceedings*, volume 9678 of *Lecture Notes in Computer Science*, pages 300–316. Springer, 2016.

[43] M. Duerst and M. Suignard. Internationalized resource identifiers (iris). RFC 3987, RFC Editor, January 2005. `http://www.rfc-editor.org/rfc/rfc3987.txt`.

[44] Mikel Egaña, Robert Stevens, and Erick Antezana. Transforming the axiomisation of ontologies: The ontology pre-processor language. In Kendall Clark and Peter F. Patel-Schneider, editors, *Proceedings of the Fourth OWLED Workshop on OWL: Experiences and Directions, Washington, DC, USA, 1-2 April 2008*, volume 496 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[45] Paola Espinoza-Arias, María Poveda-Villalón, Raúl García-Castro, and Oscar Corcho. Ontological representation of smart city data: From devices to cities. *Applied Sciences*, 9(1), 2019.

[46] European Telecommunications Standards Institute (ETSI). SmartM2M; SAREF extension investigation; requirements for industry and manufacturing domains. Technical report, ETSI, October 2018.
https://www.etsi.org/deliver/etsi_TR/103500_103599/103507/01.01.01_60/tr_103507v010101p.pdf.

[47] Alexander R. Fabbri, Patrick Ng, Zhiguo Wang, Ramesh Nallapati, and Bing Xiang. Template-based question generation from retrieved sentences for improved unsupervised question answering. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 4508–4513. Association for Computational Linguistics, 2020.

[48] Alba Fernández-Izquierdo and Raúl García-Castro. Themis: a tool for validating ontologies through requirements. In Angelo Perkusich, editor, *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10-12, 2019*, pages 573–753. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019.

[49] Alba Fernández-Izquierdo, María Poveda-Villalón, and Raúl García-Castro. CORAL: A corpus of ontological requirements annotated with lexico-syntactic patterns. In Pascal Hitzler, Miriam Fernández, Krzysztof Janowicz, Amrapali Zaveri, Alasdair J. G. Gray, Vanessa López, Armin Haller, and Karl Hammar, editors, *The Semantic Web - 16th International Conference, ESWC 2019, Portorož, Slovenia, June 2-6, 2019, Proceedings*, volume 11503 of *Lecture Notes in Computer Science*, pages 443–458. Springer, 2019.

[50] Mariano Fernandez-Lopez, Asuncion Gomez-Perez, and Natalia Juristo. Methontology: from ontological art towards ontological engineering. In *Proceedings of the AAAI97 Spring Symposium*, pages 33–40, Stanford, USA, March 1997.

[51] J. R. Firth. A synopsis of linguistic theory 1930-55. In *Studies in Linguistic Analysis (special volume of the Philological Society)*, volume 1952-59, pages 1–32, Oxford, 1957. The Philological Society.

[52] R. Fletcher. A new approach to variable metric algorithms. *The Computer Journal*, 13(3):317–322, 1970.

[53] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Ace 6.0 construction rules. Technical report, Attempto project, 2007. http://attempto.ifi.uzh.ch/site/docs/ace/6.0/ace_constructionrules.html.

[54] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto controlled english for knowledge representation. In Cristina Baroglio, Piero A. Bonatti, Jan Maluszynski, Massimo Marchiori, Axel Polleres, and Sebastian Schaffert, editors, *Reasoning Web, 4th International Summer School 2008, Venice, Italy, September 7-11, 2008, Tutorial Lectures*, volume 5224 of *Lecture Notes in Computer Science*, pages 104–124. Springer, 2008.

[55] Norbert E. Fuchs, Kaarel Kaljurand, and Gerold Schneider. Attempto controlled English meets the challenges of knowledge representation, reasoning, interoperability and user interfaces. In Geoff Sutcliffe and Randy Goebel, editors, *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference, Melbourne Beach, Florida, USA, May 11-13, 2006*, pages 664–669. AAAI Press, 2006.

[56] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202, 1980.

[57] Philip Gage. A new algorithm for data compression. *The C Users Journal archive*, 12:23–38, 1994.

[58] Fabien Gandon and Guus Schreiber. RDF 1.1 XML syntax. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/.

[59] Aldo Gangemi. Ontology design patterns for semantic web content. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005, Proceedings*, volume 3729 of *Lecture Notes in Computer Science*, pages 262–276. Springer, 2005.

[60] Giuseppe De Giacomo and Maurizio Lenzerini. TBox and ABox reasoning in expressive description logics. In Lin Padgham, Enrico Franconi, Manfred Gehrke, Deborah L. McGuinness, and Peter F. Patel-Schneider, editors, *Proceedings of the 1996 International Workshop on Description Logics, November 2-4, 1996, Cambridge, MA, USA*, volume WS-96-05 of *AAAI Technical Report*, pages 37–48. AAAI Press, 1996.

[61] Birte Glimm and Chimezie Ogbuji. SPARQL 1.1 entailment regimes. W3C recommendation, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/.

[62] Rudolphus Goclenius. Lexicon philosophicum, quo tanquam clave philosophiae fores aperiuntur. *Les Etudes Philosophiques*, 20(1):88–88, 1965.

[63] Donald Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of Computation*, 24:23–26, 1970.

[64] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[65] M Gruninger and Mark S Fox. Methodology for the design and evaluation of ontologies. 1995. In *Proceedings of the Workshop on Basic Ontological Issues in Knowledge Sharing, IJCAI-95, Montreal, Canada*, 1995.

[66] Zellig Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[67] Shizhu He, Yuanzhe Zhang, Kang Liu, and Jun Zhao. Casia@v2: A mln-based question answering system over linked data. In Linda Cappellato, Nicola Ferro, Martin Halvey, and Wessel Kraaij, editors, *Working Notes for CLEF 2014 Conference, Sheffield, UK, September 15-18, 2014*, volume 1180 of *CEUR Workshop Proceedings*, pages 1249–1259. CEUR-WS.org, 2014.

[68] Benjamin Heinzerling and Michael Strube. Sequence tagging with contextual and non-contextual subword representations: A multilingual evaluation. In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 273–291. Association for Computational Linguistics, 2019.

[69] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

[70] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[71] A. E. Hoerl and R. W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12:55–67, 1970.

[72] Johannes Hoffart, Mohamed Amir Yosef, Ilaria Bordino, Hagen Fürstenau, Manfred Pinkal, Marc Spaniol, Bilyana Taneva, Stefan Thater, and Gerhard Weikum. Robust disambiguation of named entities in text. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, EMNLP 2011, 27-31 July 2011, John McIntyre Conference Centre, Edinburgh, UK, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 782–792. ACL, 2011.

[73] Konrad Höffner, Jens Lehmann, and Ricardo Usbeck. CubeQA - question answering on RDF data cubes. In Paul Groth, Elena Simperl, Alasdair J. G. Gray, Marta Sabou, Markus Krötzsch, Freddy Lécué, Fabian Flöck, and Yolanda Gil, editors, *The Semantic Web - ISWC 2016 - 15th International Semantic Web Conference, Kobe, Japan, October 17-21, 2016, Proceedings, Part I*, volume 9981 of *Lecture Notes in Computer Science*, pages 325–340, 2016.

[74] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python, 2020. version, 3.0.1, https://doi.org/10.5281/zenodo.1212303.

[75] Matthew Horridge, Nick Drummond, John Goodwin, Alan L. Rector, Robert Stevens, and Hai Wang. The manchester OWL syntax. In Bernardo Cuenca Grau, Pascal Hitzler, Conor Shankey, and Evan Wallace, editors, *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions, Athens, Georgia, USA, November 10-11, 2006*, volume 216 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[76] Ian Horrocks, Achille Fokoue, Bernardo Cuenca Grau, Zhe Wu, and Boris Motik. OWL 2 web ontology language profiles (second edition). W3C recommendation, W3C, December 2012. https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/.

[77] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 328–339. Association for Computational Linguistics, 2018.

[78] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional LSTM-CRF models for sequence tagging. *CoRR*, abs/1508.01991, 2015.

[79] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., USA, 2009.

[80] Kaarel Kaljurand. *Attempto controlled english as a semantic web language*. PhD thesis, University of Tartu, 2007.

[81] Kaarel Kaljurand and Norbert E. Fuchs. Bidirectional mapping between OWL DL and attempto controlled english. In José Júlio Alferes, James Bailey, Wolfgang May, and Uta Schwertel, editors,

Principles and Practice of Semantic Web Reasoning, 4th International Workshop, PPSWR 2006, Budva, Montenegro, June 10-11, 2006, Revised Selected Papers, volume 4187 of *Lecture Notes in Computer Science*, pages 179–189. Springer, 2006.

[82] C. Maria Keet. Open world assumption. In *Encyclopedia of Systems Biology*, pages 1567–1567. Springer New York, New York, NY, 2013.

[83] C. Maria Keet. A core ontology of macroscopic stuff. In Krzysztof Janowicz, Stefan Schlobach, Patrick Lambrix, and Eero Hyvönen, editors, *Knowledge Engineering and Knowledge Management - 19th International Conference, EKAW 2014, Linköping, Sweden, November 24-28, 2014. Proceedings*, volume 8876 of *Lecture Notes in Computer Science*, pages 209–224. Springer, 2014.

[84] C. Maria Keet. The African wildlife ontology tutorial ontologies. *Journal of Biomedical Semantics*, 11(1):4, 2020.

[85] C. Maria Keet and Agnieszka Lawrynowicz. Test-driven development of ontologies. In Harald Sack, Eva Blomqvist, Mathieu d'Aquin, Chiara Ghidini, Simone Paolo Ponzetto, and Christoph Lange, editors, *The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Proceedings*, volume 9678 of *Lecture Notes in Computer Science*, pages 642–657. Springer, 2016.

[86] C. Maria Keet, Zola Mahlaza, and Mary-Jane Antia. Claro: a data-driven CNL for specifying competency questions. *CoRR*, abs/1907.07378, 2019.

[87] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[88] S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[89] Ilianna Kollia, Birte Glimm, and Ian Horrocks. SPARQL query answering over OWL ontologies. In Grigoris Antoniou, Marko Grobelnik, Elena Paslaru Bontas Simperl, Bijan Parsia, Dimitris Plexousakis, Pieter De Leenheer, and Jeff Z. Pan, editors, *The Semantic Web: Research and Applications - 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29-June 2, 2011, Proceedings, Part I*, volume 6643 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2011.

[90] Sandra Kübler, Ryan T. McDonald, and Joakim Nivre. *Dependency Parsing*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2009.

[91] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In Carla E. Brodley and Andrea Pohoreckyj Danyluk, editors, *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001), Williams College, Williamstown, MA, USA, June 28 - July 1, 2001*, pages 282–289. Morgan Kaufmann, 2001.

[92] Agnieszka Lawrynowicz, Jedrzej Potoniec, Michal Robaczyk, and Tania Tudorache. Discovery of emerging design patterns in ontologies using tree mining. *Semantic Web*, 9(4):517–544, 2018.

[93] Andrew Layman, Dave Hollander, and Tim Bray. Namespaces in XML. W3C recommendation, W3C, January 1999. https://www.w3.org/TR/1999/REC-xml-names-19990114/.

[94] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[95] Dong Bok Lee, Seanie Lee, Woo Tae Jeong, Donghwan Kim, and Sung Ju Hwang. Generating diverse and consistent QA pairs from contexts with information-maximizing hierarchical

conditional vaes. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 208–224. Association for Computational Linguistics, 2020.

[96] Douglas B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems; Representation and Inference in the Cyc Project.* Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1989.

[97] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7871–7880. Association for Computational Linguistics, 2020.

[98] Xiao Ling, Sameer Singh, and Daniel S. Weld. Design challenges for entity linking. *Transactions of the Association for Computational Linguistics*, 3:315–328, 2015.

[99] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45(1-3):503–528, 1989.

[100] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[101] Vanessa López, Miriam Fernández, Enrico Motta, and Nico Stieler. Poweraqua: Supporting users in querying and exploring the semantic web. *Semantic Web*, 3(3):249–265, 2012.

[102] Vanessa Lopez, Michele Pasin, and Enrico Motta. Aqualog: An ontology-portable question answering system for the semantic web. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *The Semantic Web: Research and Applications*, pages 546–562, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[103] Jacob Lorhard, Sara Uckelman, et al. Diagraph of metaphysic or ontology. https://eprints.illc.uva.nl/id/eprint/668/1/X-2008-04.text.pdf, 2008.

[104] John Lyons. *Semantics*, volume 2. Cambridge University Press, 1977.

[105] Mads Holten Rasmussen, Pieter Pauwels, Maxime Lefrançois, Georg Ferdinand Schneider. Building topology ontology (draft community group report). `https://w3c-lbd-cg.github.io/bot/#Requirements`, 2021. Accessed: 2021-11-24.

[106] James Malone, Andy Brown, Allyson L. Lister, Jon C. Ison, Duncan Hull, Helen E. Parkinson, and Robert Stevens. The software ontology (SWO): a resource for reproducibility in biomedical data analysis, curation and digital preservation. *Journal of Biomedical Semantics*, 5:25, 2014.

[107] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval.* Cambridge University Press, 2008.

[108] Christopher D. Manning and Hinrich Schütze. *Foundations of statistical natural language processing.* MIT Press, 2001.

[109] Maria Poveda-Villalón, Raúl García Castro, Fernando Serena. Vicinity project. `http://vicinity.iot.linkeddata.es/vicinity/`, 2020. Accessed: 2021-11-24.

[110] Nicolas Matentzoglu, Jared Leo, Valentino Hudhra, Uli Sattler, and Bijan Parsia. A survey of current, stand-alone OWL reasoners. In Michel Dumontier, Birte Glimm, Rafael S. Gonçalves, Matthew Horridge, Ernesto Jiménez-Ruiz, Nicolas Matentzoglu, Bijan Parsia, Giorgos B. Stamou, and Giorgos Stoilos, editors, *Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation (ORE-2015) co-located with the 28th International Workshop on Description*

Logics (DL 2015), Athens, Greece, June 6, 2015, volume 1387 of *CEUR Workshop Proceedings*, pages 68–79. CEUR-WS.org, 2015.

[111] John McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pages 75–91, London, 1959. Her Majesty's Stationary Office.

[112] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13(1):27–39, 1980. Special Issue on Non-Monotonic Logic.

[113] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[114] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[115] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Yoshua Bengio and Yann LeCun, editors, *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013.

[116] Eric Miller and Frank Manola. RDF primer. W3C recommendation, W3C, February 2004. https://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[117] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[118] Jonathan Mortensen, Matthew Horridge, Mark A. Musen, and Natalya Fridman Noy. Modest use of ontology design patterns in a repository of biomedical ontologies. In Eva Blomqvist, Aldo Gangemi, Karl Hammar, and Mari Carmen Suárez-Figueroa, editors, *Proceedings of the 3rd Workshop on Ontology Patterns, Boston, USA, November 12, 2012*, volume 929 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[119] Boris Motik, Peter Patel-Schneider, and Bijan Parsia. OWL 2 web ontology language structural specification and functional-style syntax (second edition). W3C recommendation, W3C, December 2012. https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

[120] Boris Motik, Peter Patel-Schneider, and Bijan Parsia. OWL 2 web ontology language XML serialization (second edition). W3C recommendation, W3C, December 2012. https://www.w3.org/TR/2012/REC-owl2-xml-serialization-20121211/.

[121] Kevin P. Murphy. *Machine learning - a probabilistic perspective*. Adaptive computation and machine learning series. MIT Press, 2012.

[122] Nils J. Nilsson. *The Quest for Artificial Intelligence*. Cambridge University Press, USA, 1st edition, 2009.

[123] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35:773–782, 1980.

[124] N. Noy and Deborah Mcguinness. Ontology development 101: A guide to creating your first ontology. *Knowledge Systems Laboratory*, 32, 01 2001.

[125] Chimezie Ogbuji and Birte Glimm. SPARQL 1.1 entailment regimes. W3C recommendation, W3C, March 2013. https://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/.

[126] Naoaki Okazaki. CRFsuite: a fast implementation of conditional random fields (CRFs), 2007. version 0.12, http://www.chokkan.org/software/crfsuite/, Accessed: 2021-05-25.

[127] Pance Panov, Larisa N. Soldatova, and Saso Dzeroski. Generic ontology of datatypes. *Information Sciences*, 329:900–920, 2016.

[128] Paola Espinoza, Miguel Angel Garcia, Oscar Corcho. btn100 ontology – requirements. `https://github.com/oeg-upm/ontology-BTN100/tree/master/requirements`, 2020. Accessed: 2021-11-24.

[129] Seonyeong Park, Hyosup Shim, and Gary Geunbae Lee. ISOFT at QALD-4: semantic similarity-based question answering system over linked data. In Linda Cappellato, Nicola Ferro, Martin Halvey, and Wessel Kraaij, editors, *Working Notes for CLEF 2014 Conference, Sheffield, UK, September 15-18, 2014*, volume 1180 of *CEUR Workshop Proceedings*, pages 1236–1248. CEUR-WS.org, 2014.

[130] Bijan Parsia et al. OWL 2 web ontology language primer (second edition). Technical report, W3C, December 2012. http://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[131] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL, 2014.

[132] Silvio Peroni. A simplified agile methodology for ontology development. In Mauro Dragoni, María Poveda-Villalón, and Ernesto Jiménez-Ruiz, editors, *OWL: - Experiences and Directions - Reasoner Evaluation - 13th International Workshop, OWLED 2016, and 5th International Workshop, ORE 2016, Bologna, Italy, November 20, 2016, Revised Selected Papers*, volume 10161 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2016.

[133] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In Marilyn A. Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, pages 2227–2237. Association for Computational Linguistics, 2018.

[134] Jedrzej Potoniec, Dawid Wisniewski, and Agnieszka Ławrynowicz. Incorporating presuppositions of competency questions into test-driven development of ontologies. In *SEKE 2021 : Proceedings of the 33rd International Conference on Software Engineering and Knowledge Engineering*, pages 437–440. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2021.

[135] Jedrzej Potoniec, Dawid Wiśniewski, Agnieszka Ławrynowicz, and C. Maria Keet. Dataset of ontology competency questions to SPARQL-OWL queries translations. *Data in Brief*, 29:105098, 2020.

[136] Valentina Presutti, Eva Blomqvist, Enrico Daga, and Aldo Gangemi. Pattern-based ontology design. In Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, Enrico Motta, and Aldo Gangemi, editors, *Ontology Engineering in a Networked World*, pages 35–64. Springer, 2012.

[137] Valentina Presutti, Enrico Daga, Aldo Gangemi, and Eva Blomqvist. extreme design with content ontology design patterns. In Eva Blomqvist, Kurt Sandkuhl, François Scharffe, and Vojtech Svátek, editors, *Proceedings of the Workshop on Ontology Patterns (WOP 2009) , collocated with the 8th International Semantic Web Conference (ISWC-2009), Washington D.C., USA, 25 October, 2009*, volume 516 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[138] Eric Prud'hommeaux and Gavin Carothers. RDF 1.1 turtle. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-turtle-20140225/.

[139] W. V. Quine. On what there is. In W. V. Quine, editor, *From a Logical Point of View*, pages 1–19. Cambridge, Mass.: Harvard University Press, 1953.

[140] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[141] Lance Ramshaw and Mitch Marcus. Text chunking using transformation-based learning. In *Third Workshop on Very Large Corpora*, 1995.

[142] Delip Rao, Paul McNamee, and Mark Dredze. Entity linking: Finding extracted entities in a knowledge base. In Thierry Poibeau, Horacio Saggion, Jakub Piskorski, and Roman Yangarber, editors, *Multi-source, Multilingual Information Extraction and Summarization*, Theory and Applications of Natural Language Processing, pages 93–115. Springer, 2013.

[143] Lila Rao, Han Reichgelt, and Kweku-Muata Osei-Bryson. Knowledge elicitation techniques for deriving competency questions for ontologies. In José Cordeiro and Joaquim Filipe, editors, *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume ISAS-2, Barcelona, Spain, June 12-16, 2008*, pages 105–110, 2008.

[144] Manoj Prabhakar Kannan Ravi, Kuldeep Singh, Isaiah Onando Mulang, Saeedeh Shekarpour, Johannes Hoffart, and Jens Lehmann. CHOLAN: A modular approach for neural entity linking on wikipedia and wikidata. In Paola Merlo, Jörg Tiedemann, and Reut Tsarfaty, editors, *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, EACL 2021, Online, April 19 - 23, 2021*, pages 504–514. Association for Computational Linguistics, 2021.

[145] Alan L. Rector, Nick Drummond, Matthew Horridge, Jeremy Rogers, Holger Knublauch, Robert Stevens, Hai Wang, and Chris Wroe. OWL pizzas: Practical experience of teaching OWL-DL: common errors & common patterns. In Enrico Motta, Nigel Shadbolt, Arthur Stutt, and Nicholas Gibbins, editors, *Engineering Knowledge in the Age of the Semantic Web, 14th International Conference, EKAW 2004, Whittlebury Hall, UK, October 5-8, 2004, Proceedings*, volume 3257 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2004.

[146] Raymond Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, pages 55–76, New York, 1977. Plemum Press.

[147] Raymond Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, pages 191–238. Springer New York, New York, NY, 1984.

[148] Yuan Ren, Artemis Parvizi, Chris Mellish, Jeff Z. Pan, Kees van Deemter, and Robert Stevens. Towards competency question-driven ontology authoring. In Valentina Presutti, Claudia d'Amato, Fabien Gandon, Mathieu d'Aquin, Steffen Staab, and Anna Tordai, editors, *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*, volume 8465 of *Lecture Notes in Computer Science*, pages 752–767. Springer, 2014.

[149] Robert Stevens. Competency questions for ontologies. `http://studentnet.cs.manchester.ac.uk/pgt/2014/COMP60421/slides/Week2-CQ.pdf`, 2014. Accessed: 2021-05-25.

[150] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536, 1986.

[151] Vasile Rus, Brendan Wyse, Paul Piwek, Mihai Lintean, Svetlana Stoyanchev, and Christian Moldovan. The first question generation shared task evaluation challenge. In *Proceedings of the 6th International Natural Language Generation Conference*. Association for Computational Linguistics, July 2010.

[152] Stefan Ruseti, Alexandru Mirea, Traian Rebedea, and Stefan Trausan-Matu. Qanswer - enhanced entity matching for question answering over linked data. In Linda Cappellato, Nicola Ferro, Gareth J. F. Jones, and Eric SanJuan, editors, *Working Notes of CLEF 2015 - Conference and Labs of the Evaluation forum, Toulouse, France, September 8-11, 2015*, volume 1391 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

[153] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010.

[154] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[155] Michael Schneider. OWL 2 web ontology language RDF-based semantics (second edition). W3C recommendation, W3C, December 2012. https://www.w3.org/TR/2012/REC-owl2-rdf-based-semantics-20121211/.

[156] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2012, Kyoto, Japan, March 25-30, 2012*, pages 5149–5152. IEEE, 2012.

[157] Andy Seaborne and Gavin Carothers. RDF 1.1 n-triples. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-n-triples-20140225/.

[158] Andy Seaborne and Steven Harris. SPARQL 1.1 query language. W3C recommendation, W3C, March 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[159] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The semantic web revisited. *IEEE Intell. Syst.*, 21(3):96–101, 2006.

[160] David F Shanno. Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656, 1970.

[161] Saeedeh Shekarpour, Edgard Marx, Axel-Cyrille Ngonga Ngomo, and Sören Auer. SINA: semantic interpretation of user queries for question answering on interlinked data. *Journal of Web Semantics*, 30:39–51, 2015.

[162] Evren Sirin and Bijan Parsia. SPARQL-DL: SPARQL query for OWL-DL. In Christine Golbreich, Aditya Kalyanpur, and Bijan Parsia, editors, *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions, Innsbruck, Austria, June 6-7, 2007*, volume 258 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[163] Barry Smith and Christopher A. Welty. FOIS introduction: Ontology - towards a new synthesis. In *2nd International Conference on Formal Ontology in Information Systems, FOIS 2001, Ogunquit, Maine, USA, October 17-19, 2001, Proceedings*, pages iii–ix. ACM, 2001.

[164] Katherine Stasaski and Marti A. Hearst. Multiple choice question generation utilizing an ontology. In *Proceedings of the 12th Workshop on Innovative Use of NLP for Building Educational Applications*, pages 303–312, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[165] Thanos G. Stavropoulos, Georgios Meditskos, Ioannis Kompatsiaris, and Stelios Andreadis. Dem@Care: Ambient sensing and intelligent decision support for the care of dementia. In James Malone, Robert Stevens, Kerstin Forsberg, and Andrea Splendiani, editors, *Proceedings of the 8th Semantic Web Applications and Tools for Life Sciences International Conference, Cambridge UK, December 7-10, 2015*, volume 1546 of *CEUR Workshop Proceedings*, pages 229–230. CEUR-WS.org, 2015.

[166] Simon Steyskal, Benedict Whittam Smith, and Michael Steidl. POE use cases and requirements. W3C working draft, W3C, February 2017. `https://www.w3.org/TR/2017/WD-poe-ucr-20170223/`.

[167] Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, and Mariano Fernández-López. The NeOn methodology framework: A scenario-based methodology for ontology development. *Appl. Ontology*, 10(2):107–145, 2015.

[168] Mari Carmen Suárez-Figueroa, Asunción Gómez-Pérez, and Boris Villazón-Terrazas. How to write and use the ontology requirements specification document. In Robert Meersman, Tharam S. Dillon, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2009, Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009, Vilamoura, Portugal, November 1-6, 2009, Proceedings, Part II*, volume 5871 of *Lecture Notes in Computer Science*, pages 966–982. Springer, 2009.

[169] York Sure, Steffen Staab, and Rudi Studer. On-to-knowledge methodology (otkm). In *Handbook on Ontologies*, pages 117–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[170] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[171] Charles Sutton and Andrew McCallum. An introduction to conditional random fields. *Foundations and Trends in Machine Learning*, 4(4):267–373, 2012.

[172] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society (Series B)*, 58:267–288, 1996.

[173] Christina Unger, Lorenz Bühmann, Jens Lehmann, Axel-Cyrille Ngonga Ngomo, Daniel Gerber, and Philipp Cimiano. Template-based question answering over RDF data. In Alain Mille, Fabien Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab, editors, *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, pages 639–648. ACM, 2012.

[174] Ricardo Usbeck, Ria Hari Gusmita, Axel-Cyrille Ngonga Ngomo, and Muhammad Saleem. 9th challenge on question answering over linked data (QALD-9) (invited paper). In Key-Sun Choi, Luis Espinosa Anke, Thierry Declerck, Dagmar Gromann, Jin-Dong Kim, Axel-Cyrille Ngonga Ngomo, Muhammad Saleem, and Ricardo Usbeck, editors, *Joint proceedings of the 4th Workshop on Semantic Deep Learning (SemDeep-4) and NLIWoD4: Natural Language Interfaces for the Web of Data (NLIWOD-4) and 9th Question Answering over Linked Data challenge (QALD-9) co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, California, United States of America, October 8th - 9th, 2018*, volume 2241 of *CEUR Workshop Proceedings*, pages 58–64. CEUR-WS.org, 2018.

[175] Mike Uschold and Michael Gruninger. Ontologies: principles, methods and applications. *Knowledge Engineering Review*, 11(2):93–136, 1996.

[176] Mike Uschold and Martin King. Towards a methodology for building ontologies. In *In Workshop on Basic Ontological Issues in Knowledge Sharing, held in conjunction with IJCAI-95*, 1995.

[177] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[178] Denny Vrandecic and Aldo Gangemi. Unit tests for ontologies. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, COMINF, IS, KSinBIT, MIOS-CIAO, MONET, OnToContent, ORM, PerSys, OTM Academy*

Doctoral Consortium, RDDS, SWWS, and SeBGIS 2006, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part II, volume 4278 of *Lecture Notes in Computer Science*, pages 1012–1020. Springer, 2006.

[179] Taowei David Wang, Bijan Parsia, and James A. Hendler. A survey of the web ontology landscape. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *The Semantic Web - ISWC 2006, 5th International Semantic Web Conference, ISWC 2006, Athens, GA, USA, November 5-9, 2006, Proceedings*, volume 4273 of *Lecture Notes in Computer Science*, pages 682–694. Springer, 2006.

[180] Xinyu Wang and Kewei Tu. Second-order neural dependency parsing with message passing and end-to-end training. In Kam-Fai Wong, Kevin Knight, and Hua Wu, editors, *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing, AACL/IJCNLP 2020, Suzhou, China, December 4-7, 2020*, pages 93–99. Association for Computational Linguistics, 2020.

[181] Ralph Weischedel, Eduard Hovy, Mitchell Marcus, Martha Palmer, Robert Belvin, Sameer Pradhan, Lance Ramshaw, and Nianwen Xue. Ontonotes: A large training corpus for enhanced processing. *Handbook of Natural Language Processing and Machine Translation. Springer*, 3(3):3–4, 2011.

[182] Christopher Welty and Deborah McGuinness. OWL web ontology language guide. W3C recommendation, W3C, February 2004. https://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[183] Patricia L. Whetzel, Natalya Fridman Noy, Nigam H. Shah, Paul R. Alexander, Csongor Nyulas, Tania Tudorache, and Mark A. Musen. Bioportal: enhanced functionality via new web services from the national center for biomedical ontology to access and use ontologies in software applications. *Nucleic Acids Research*, 39(Web-Server-Issue):541–545, 2011.

[184] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, nov 1977.

[185] Dawid Wisniewski. Automatic translation of competency questions into SPARQL-OWL queries. In Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis, editors, *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, pages 855–859. ACM, 2018.

[186] Dawid Wisniewski and Agnieszka Lawrynowicz. A tagger for glossary of terms extraction from ontology competency questions. In Pascal Hitzler, Sabrina Kirrane, Olaf Hartig, Victor de Boer, Maria-Esther Vidal, Maria Maleshkova, Stefan Schlobach, Karl Hammar, Nelia Lasierra, Steffen Stadtmüller, Katja Hose, and Ruben Verborgh, editors, *The Semantic Web: ESWC 2019 Satellite Events - ESWC 2019 Satellite Events, Portorož, Slovenia, June 2-6, 2019, Revised Selected Papers*, volume 11762 of *Lecture Notes in Computer Science*, pages 181–185. Springer, 2019.

[187] Dawid Wisniewski, Jedrzej Potoniec, and Agnieszka Lawrynowicz. BigCQ: A large-scale synthetic dataset of competency question patterns formalized into SPARQL-OWL query templates. *CoRR*, abs/2105.09574, 2021.

[188] Dawid Wisniewski, Jedrzej Potoniec, and Agnieszka Lawrynowicz. SeeQuery: An automatic method for recommending translations of ontology competency questions into SPARQL-OWL. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, CIKM '21, page 2119–2128, New York, NY, USA, 2021. Association for Computing Machinery.

[189] Dawid Wiśniewski, Jedrzej Potoniec, Agnieszka Ławrynowicz, and C. Maria Keet. Analysis of ontology competency questions and their formalizations in SPARQL-OWL. *Journal of Web Semantics*, 59:100534, 2019.

[190] Ledell Wu, Fabio Petroni, Martin Josifoski, Sebastian Riedel, and Luke Zettlemoyer. Scalable zero-shot entity linking with dense entity retrieval. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 6397–6407. Association for Computational Linguistics, 2020.

[191] Dongling Xiao, Han Zhang, Yu-Kun Li, Yu Sun, Hao Tian, Hua Wu, and Haifeng Wang. ERNIE-GEN: an enhanced multi-flow pre-training and fine-tuning framework for natural language generation. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 3997–4003. ijcai.org, 2020.

[192] Mohamed Yahya, Klaus Berberich, Shady Elbassuoni, and Gerhard Weikum. Robust question answering over the web of linked data. In Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi, editors, *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 1107–1116. ACM, 2013.

[193] Ikuya Yamada, Akari Asai, Hiroyuki Shindo, Hideaki Takeda, and Yuji Matsumoto. LUKE: deep contextualized entity representations with entity-aware self-attention. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP 2020, Online, November 16-20, 2020*, pages 6442–6454. Association for Computational Linguistics, 2020.

[194] Mohammed Javeed Zaki. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, 66(1-2):33–52, 2005.

[195] Leila Zemmouchi-Ghomari and Abdessamed Réda Ghomari. Translating natural language competency questions into sparqlqueries: a case study. In *The First International Conference on Building and Exploring Web Based Environments*, pages 81–86. sn, 2013.

[196] Ruqing Zhang, Jiafeng Guo, Lu Chen, Yixing Fan, and Xueqi Cheng. A review on question generation from natural language text. *ACM Transactions on Information Systems*, 40(1), sep 2021.

[197] Yu Zhang, Zhenghua Li, and Min Zhang. Efficient second-order TreeCRF for neural dependency parsing. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3295–3305, Online, July 2020. Association for Computational Linguistics.

[198] Weiguo Zheng, Lei Zou, Xiang Lian, Jeffrey Xu Yu, Shaoxu Song, and Dongyan Zhao. How to build templates for RDF question/answering: An uncertain graph similarity join approach. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1809–1824. ACM, 2015.

[199] Junru Zhou, Zuchao Li, and Hai Zhao. Parsing all: Syntax and semantics, dependencies and spans. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 4438–4449. Association for Computational Linguistics, 2020.

[200] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society, Series B*, 67:301–320, 2005.

[201] Arnold M. Zwicky. Heads. *Journal of Linguistics*, 21(1):1–29, 1985.

Appendix A

Algorithms used to group and analyze CQ2SPARQLOWL

A.1 Extraction of a CQ pattern candidate from a given CQ

The following pseudo-codes describe the main algorithm (Algorithm 3) along with helper procedures that are used to transform CQs into CQ pattern candidates.

```
Input: A competency question cq
Output: A pattern extracted from cq
cq ← normalizeCQ(cq)
pattern ← markPlaceholders(cq)
pattern ← extractEcChunks(pattern)
pattern ← extractPcChunks(pattern)
return pattern
```

Algorithm 3: An algorithm defining the pattern candidate generation procedure out of a given CQ *cq*. Reprint from [189].

```
function normalizeCQ(cq)
    cq ← lowercase(cq)
    foreach regex ∈ {[-"’‘], (.*?)}
        cq ← cq.regexReplace(regex,””)
    cq ← cq.regexReplace(” \?”,”?”)
    cq ← cq.regexReplace(”[ \t]+”,” ”)
    return cq
end
```

Algorithm 4: An algorithm defining the CQ normalization procedure involving CQ lowercasing, removing redundant spaces and removing dashes as well as texts in brackets. Reprint from [189].

```
function markChunk(cq, startEndPositions, typeID, counter)
    chunkID = typeID + string(counter)
    cq = cq.spanReplace(startEndPositions, chunkID)
    return cq
end
```

Algorithm 5: An algorithm defining the procedure substituting a given span of characters with a given chunk identifier (e.g. PC1 representing the first predicate chunk). Reprint from [189].

```
function normalizeNounChunk($nounChunkTokens$)
    if $nounChunkTokens.firstToken \in \{any, some, many, well, its, much, few\}$
        $nounChunkTokens \leftarrow nounChunkTokens$.removeFirstToken()
    return $nounChunkTokens$
end
```

Algorithm 6: An algorithm rejecting a predefined list of tokens from noun chunks extracted by spaCy. Reprint from [189].

```
function extractEcChunks($cq$)
    $tokens \leftarrow$ tokenize($cq$)
    $counter \leftarrow 1$
    $disallowedPhrases = \{$type,types,kind,kinds,category,categories$\}$
    $disallowedPhrases+ = \{$difference,differences,extent,i,we,there$\}$
    $disallowedPhrases+ = \{$respect,the main types,the possible types$\}$
    $disallowedPhrases+ = \{$the types,the difference,the differences$\}$
    $disallowedPhrases+ = \{$the main categories$\}$
    if $tokens.first =$ "how" $\wedge$ postag($tokens.second$) = "ADJ" $\wedge$ postag($tokens.third =$ "VERB"
        $cq =$ markChunk($cq, getSpan(token.second)$, "EC", $counter$)
        $counter = counter + 1$
    foreach $nounChunk \in$ getNounChunks($tokens$)
        $chunkTokens \leftarrow$ normalizeNounChunk($nounChunk$)
        if $chunkTokens \in disallowedPhrases$
            continue
        $cq \leftarrow$ markChunk($cq$, getSpan($chunkTokens$), "EC", $counter$)
        $counter = counter + 1$
    $tokensNum =$ length($tokens$)
    if $tokens[tokensNum - 1] =$ "?" $\wedge$ postag($tokens[tokensNum - 2] =$ "VERB" $\wedge$ postag($tokens[tokenNum - 3]$) $\in \{$are, is, were, was, will$\}$
        $cq \leftarrow$ markChunk($cq$, getSpan($tokens[tokensNum - 2]$), "EC", $counter$)
        $counter = counter + 1$
    if $tokens[tokensNum - 1] =$ "?" $\wedge$ postag($tokens[tokensNum - 2] \in \{$ADJ, ADV$\}$
        $cq \leftarrow$ markChunk($cq$, getSpan($tokens[tokensNum - 2]$), "EC", $counter$)
        $counter = counter + 1$
    return $cq$
end
```

Algorithm 7: An algorithm providing entity chunks (EC) extraction from the CQ *cq*. Reprint from [189].

```
function extractPcChunks($cq$)
    $tokens =$ tokenize(cq)
    $counter = 1$
    $rejectingPhrases = \{$is,'s,are,was,do,does,did,were,have,had,can$\}$
    $rejectingPhrases+ = \{$could,categorise,regarding,is of,are of$\}$
    $rejectingPhrases+ = \{$are in,given,is there$\}$
    $rules = \{$(PART|VERB)*VERB$\}$
    $rules+ = \{$(PART|VERB)+ (ADJ|ADV)+ADP$\}$
    $rules+ = \{$(PART|VERB)+ADP$\}$
    foreach $chunkTokens \in$ getMatches($tokens, rules$)
        $chunkTokens = chunkTokens +$ addAuxilary($tokens, chunkTokens$)
        if $chunkTokens \in rejectingPhrases$
            continue
        $cq \leftarrow$ markChunk($cq$, getSpan($chunkTokens$), "PC", $counter$)
        $counter = counter + 1$
    return $cq$
end
```

Algorithm 8: An algorithm providing predicate chunks (PC) extraction from the CQ *cq*.

```
1 function addAuxilary(cqTokens, pcTokens)
2     $auxilaries = \{\}$
3     foreach $token \in cqTokens$
4         if $token.head \in pcTokens \wedge token.depLabel = $ "aux" $\wedge token \notin pcTokens$"
5             $auxilaries+ = token$
6     return $auxilaries$
7 end
```

Algorithm 9: An algorithm providing the procedure to attach auxiliary verbs to PC chunks. It allows to include modal verbs to predicates so that discontinuous PC chunks can be provided (e.g. Do animals eat impalas? contains a discontinuous PC chunk do . . . eat). Reprint from [189].

Function name	Description
`tokenize`(*text*)	Transform *text* into a sequence of tokens.
`postag`(*token*)	Obtain the part-of-speech token assigned to a given token.
`string`(*variable*)	Return the string representing *variable*.
`getSpan`(x)	Find at which positions phrase x begins and ends.
`text.regexReplace`(*re*, *filler*)	Use *filler* to replace matches of *re* in *text*.
`text.spanReplace`(*span*, *filler*)	Replace text at a given *span* with *filler* in given *text*.
`length`(*tokens*)	Calculate the number of tokens in *tokens* sequence.
`tokens.removeFirstToken`()	Remove the first token in *tokens* sequence.
`getMatches`(*tokens*, *rules*)	Return tokens matching one or more rules from *rules*.

Table A.1: A list of popular natural language processing methods that are used in CQ pattern extraction algorithms provided. Reprint from [189].

Rule expression	Meaning
`ADJ,ADV,VERB,PART,ADP`	POS-tags: adjective, adverb, verb, particle and adposition, respectively.
|	Elements alternative.
()	Grouping operator, used together with + or ∗ operators.
+	Operator requiring at least one occurrence of preceding group/POS-tag.
∗	Operator requiring at least one occurrence of preceding group/POS-tag.

Table A.2: Rule language components. Reprint from [189].

A.2 Extraction of a SPARQL-OWL signature from a given SPARQL-OWL query

```
Input: A SPARQL query query
Output: A signature of the query
1 root ← SPARQLToAlgebra(query)
2 root ← ExtractSignature(root)
3 return AlgebraToSPARQL(root)
```

Algorithm 10: An algorithm extracting a signature of a given SPARQL query *query* . Reprint from [189].

```
function ExtractSignature(op)
    newsubops ← empty vector
    foreach subop ∈ root.subops
        newsubops.append(ExtractSignature(subop))
    op.subops = newsubops
    return Process(op)
end
```

Algorithm 11: A function traversing a SPARQL algebra tree rooted in *op* and recursively processing it. Reprint from [189].

```
function Process(op)
    SolutionModifiers ← {ToList, OrderBy, Project, Distinct, Reduced, Slice,
     ToMultiSet}
    if op.name ∈ SolutionModifiers
        return op.subop
    if op.name = BGP
        // op.args is a list of triple patterns
        tps ← ∅
        foreach s, p, o ∈ op.args
            if ¬ (p = rdf:type ∧ o ∈ {owl:Restriction, owl:Class})
                tps ← tps ∪ {(Rename(s), Rename(p), Rename(o)}
        return BGP(tps)
    if op.name = Filter
        op.filterexpr = ProcessFilter(op.filterexpr)
        return op
    if op.name = Path
        op.p ← DropStarFromTransitive(op.p)
        if op.p.name = seq ∧ op.p.args[0] = rdf:type ∧ op.p.args[1] = rdfs:subClassOf
            op.p ← rdf:type
        return op
    if op.name = Group
        triplepatterns ← empty vector
        other ← empty vector
        foreach subop ∈ op.subops
            if subop.name = BGP
                triplepatterns.append(subop.args)
            else
                others.append(subop)
        // triplepatterns contains all triple patterns that occured in the BGPs
           in this group
        bgp ← BGP(triplepatterns)
        if others is empty
            return bgp
        else
            return Group(others, bgp)
    return op
end
```

Algorithm 12: A function processing a single node of a SPARQL algebra tree. Reprint from [189].

```
function DropStarFromTransitive(path)
    if $path.name \in \{ZeroOrMorePath, OneOrMorePath\} \wedge path.subpath \in \{rdfs{:}subClassOf, rdfs{:}subPropertyOf\}$
        return $path.subpath$
    $subpaths \leftarrow$ empty vector
    foreach $p \in path.subpaths$
        $subpaths.append($DropStarFromTransitive($p$)$)$
    $p.subpaths = subpaths$
    return $p$
end
```

Algorithm 13: A function to remove `*` and `+` from property paths containing known transitive properties. Reprint from [189]

```
function ProcessFilter(filterexpr)
    if $filterexpr.name \in \{logical\text{-}and, logical\text{-}or$
        {
    $filterexpr.left \leftarrow$ ProcessFilter($filterexpr$)
    $filterexpr.right \leftarrow$ ProcessFilter($filterexpr$)
    if $filterexpr.left$ is None return $filterexpr.right$
    if $filterexpr.right$ is None return $filterexpr.left$
    return $filterexpr$
    } if $filterexpr.name = RDFterm\text{-}equal$
        if $filterexpr.left = owl{:}Nothing \vee filterexpr.right = owl{:}Nothing$
            return None
    return $filterexpr$
end
```

Algorithm 14: A function to remove comparison with `owl:Nothing` in filter expressions. Reprint from [189]

```
function Rename(node)
    if node is an IRI $\wedge namespace(node) \notin \{RDF, RDFS, OWL, XSD\}$
        $id \leftarrow SafeBase64(node)$
        return BlankNode(id)
    return node
end
```

Algorithm 15: A function to consistently replace all IRIs from namespaces other than RDF, RDFS, OWL and XSD with blank nodes. Reprint from [189]

Appendix B

CQ patterns in CQ2SPARQLOWL

B.1 CQ patterns

What is EC1 PC1 EC2

What are EC1 to EC2

What EC1 to EC2 are there

Which of EC1 PC1 EC2 PC1

Are there any EC1 to EC2 EC3 PC1

PC1 EC1 PC1 EC2

What type of EC1 is EC2

What EC1 PC1 EC2

Is EC1 EC2 for EC3

What are EC1 and EC2 of EC3

Which EC1 is there for EC2 and what PC1 EC1 PC1

Which EC1 PC1 EC2

What EC1 PC1 I PC1 EC2 PC1 EC3

What are EC1 and EC2 for EC3

What EC1 from EC2 PC1 EC3, EC4

What are EC1 for EC2

What is EC1 for EC2

PC1 EC1 PC1 EC2 to EC3

PC1 EC1 PC1 EC2 that are EC3 from EC4

To what extent PC1 EC1 PC1 EC2

What EC1 PC1 I PC1 EC2 in EC3

Is EC1 of EC2 EC3

PC1 I PC1 EC1 if EC2 PC2 EC3

Given EC1, what are EC2 for EC3 of EC4

Where PC1 I PC1 EC1

Is there EC1 for EC2

How PC1 I PC1 EC1

How PC1 I PC1 EC1 with EC2

How PC1 I PC1 EC1 with EC2 PC1

Are there any EC1 PC1 EC2 PC1

Where PC1 I PC1 EC1 for EC2

Who PC1 EC1

What is EC1 of EC2
PC1 we PC1 EC1 of EC2
Where PC1 I PC1 EC1 PC1
Which EC1 PC1 I PC1 EC2 PC1
Which is EC1 PC1 EC2
Do I know EC1 who PC1 EC2 or PC2 EC3
How and where PC1 EC1 PC1 in the past
How long PC1 EC1 PC1
How EC1 is EC2
What do EC1 PC1 EC2 EC3
What EC1 PC1 EC2 given EC3
Who are EC1 of EC2
Who else PC1 EC1 EC2
How many EC1 PC1 I PC1 EC2
What EC1 are in EC2 of EC3
What are the differences between EC1 of EC2
When PC1 EC1 of EC2 PC1
Is EC1 EC2
What EC1 does EC2 have, and what is its EC3
Is EC1 EC2 or not
At what EC1 PC1 EC2 of EC3 PC1
Who PC1 EC1 for EC2
How many EC1 PC1 we PC1 EC2 EC3
PC1 I PC1 EC1 PC1 EC2
Does EC1 of EC2 PC1 EC3
Are there any EC1 for EC2
Is there any EC1 for EC2 and where PC1 I PC1 EC3
Does EC1 have EC2
Where is EC1 of EC2
Where's EC1 of EC2
How well PC1 is EC1 for EC2
Is there EC1 with EC2
How PC1 I PC1 EC1 PC1 EC2
PC1 I PC1 some EC1 of EC2 for EC3
What EC1 PC1 I PC1 EC2
What EC1 PC1 EC2 PC1
In what EC1 PC1 EC2 PC1
PC1 I PC1 EC1 on EC2
What EC1 PC1 I PC1 EC2 on EC3
Is EC1 EC2 or EC3
What is the difference between EC1 and EC2
In which EC1 are EC2 in EC3
Which kind of EC1 are EC2
What kind of EC1 is EC2
Where do I categorise EC1 like EC2
Which EC1 PC1 EC2 PC1
Which EC1 are EC2 of EC3

Are there EC1 in EC2
Which EC1 PC1 I PC1 PC2 EC2
In what kind of EC1 PC1 EC2 PC1
Which EC1 are EC2
PC1 EC1 and EC2 PC1 EC3
What types of EC1 are EC2
What are the main types of EC1
What are the types of EC1
Which are EC1
What PC1 EC1
What PC1 EC1 of EC2
What EC1 are of EC2 with respect to EC3
What EC1 is of EC2 regarding EC3
What EC1 PC1 EC2 or EC3 that PC2 EC4
What EC1 is of EC2 regarding EC3 and EC4
What are the main categories of EC1
What EC1 are EC2
What are the main types of EC1 EC2 PC1
What types of EC1 PC1 EC2
What are the possible types of EC1
What is EC1 of EC2 for EC3
What is EC1 of EC2 that have EC3
What is EC1 of EC2 that have EC3 and EC4
What are EC1 that have EC2
What is EC1 of EC2 that have EC3 as EC4
What is EC1 of EC2 that PC1 EC3
What EC1 of EC2 PC1 EC3

B.2 Higher-level CQ patterns

What is EC1 PC1 EC2
What is EC1 to EC2
What EC1 to EC2 is there
What of EC1 PC1 EC2 PC1
Is there any EC1 to EC2 EC3 PC1
PC1 EC1 PC1 EC2
What type of EC1 is EC2
What EC1 PC1 EC2
Is EC1 EC2
What is EC1 and EC2
What EC1 is there for EC2 and what PC1 EC1 PC1
What EC1 PC1 I PC1 EC2 PC1 EC3
What is EC1
PC1 EC1 PC1 EC2 to EC3
PC1 EC1 PC1 EC2 that is EC3 from EC4
To what extent PC1 EC1 PC1 EC2
What EC1 PC1 I PC1 EC2

PC1 I PC1 EC1 if EC2 PC2 EC3
Given EC1, what is EC2
Where PC1 I PC1 EC1
Is there EC1
How PC1 I PC1 EC1
How PC1 I PC1 EC1 PC1
Is there any EC1 PC1 EC2 PC1
Who PC1 EC1
PC1 I PC1 EC1
Where PC1 I PC1 EC1 PC1
What EC1 PC1 I PC1 EC2 PC1
Which is EC1
Do I know EC1 who PC1 EC2 or PC2 EC3
How and where PC1 EC1 PC1 in the past
How long PC1 EC1 PC1
How EC1 is EC2
What do EC1 PC1 EC2
What EC1 PC1 EC2 given EC3
Who is EC1
Who else PC1 EC1 EC2
How many EC1 PC1 I PC1 EC2
What EC1 is in EC2
What is the difference between EC1
When PC1 EC1 PC1
What EC1 do EC2 have, and what is its EC3
Is EC1 EC2 or not
At what EC1 PC1 EC2 PC1
How many EC1 PC1 I PC1 EC2 EC3
PC1 I PC1 EC1 PC1 EC2
Do EC1 PC1 EC2
Is there any EC1
Is there any EC1 and where PC1 PC1 EC2
Do EC1 have EC2
Where is EC1
Where's EC1
How EC1 PC1 is EC2
How PC1 I PC1 EC2
How PC1 I PC1 EC1 PC1 EC2
PC1 I PC1 some EC1
What EC1 PC1 EC2 PC1
In what EC1 PC1 EC2 PC1
PC1 I PC1 EC1
Is EC1 EC2 or EC3
What is the difference between EC1 and EC2
In which EC1 is EC2
Where do I categorise EC1 like EC2
What EC1 is EC2

In what type of EC1 PC1 EC2 PC1
PC1 EC1 and EC2 PC1 EC3
What is the main type of EC1
What is the type of EC1
What EC1 PC1 EC2 and EC3
What PC1 EC1
What EC1 is of EC2 with respect to EC3
What EC1 is of EC2 regarding EC3
What EC1 PC1 EC2 or EC3 that PC2 EC4
What EC1 is of EC2 regarding EC3 and EC4
What is the main type of EC1 EC2 PC1
What type of EC1 PC1 EC2
What is the possible type of EC1
What is EC1 that have EC2
What is EC1 that have EC2 and EC3
What is EC1 that PC1 EC2
How well PC1 is EC1

Appendix C

Phrases rejected by ReqTagger

C.1 Entities

- a kind
- the kind
- kind
- kinds
- the kinds
- category
- a category
- the category
- categories
- the categories
- type
- a type
- the type
- types
- the types

C.2 Relations

- is
- ’s
- are
- was
- do
- does
- did
- were
- have
- had
- has
- can
- could
- regarding
- is of
- are of
- are in
- given
- is there

Appendix D

Verbalized axiom shapes used in BigCQ

```
Every c1 dp1 dt1.

Every c1 is a c2 that dp1 dt1 or that dp2 dt2. Every c2 that dp1 dt1 or
that dp2 dt2 is a c1.

Every c1 is a c2 that dp1 dt1. Every c2 that dp1 dt1 is a c1.

Every c1 is a c2 that is a c3. Every c2 that is a c3 is a c1.

Every c1 is a c2 that op1 a c3 and that op2 a c4 and that op3 a c5.
Every c2 that op1 a c3 and that op2 a c4 and that op3 a c5 is a c1.

Every c1 is a c2 that op1 a c3 and that op2 a c4 and that op3 i1.
Every c2 that op1 a c3 and that op2 a c4 and that op3 i1 is a c1.

Every c1 is a c2 that op1 a c3 and that op2 a c4 and that op3 nothing but c5 and
that op4 nothing but c6.
Every c2 that op1 a c3 and that op2 a c4 and that op3 nothing but c5 and
that op4 nothing but c6 is a c1.

Every c1 is a c2 that op1 a c3 and that op2 a c4.
Every c2 that op1 a c3 and that op2 a c4 is a c1.

Every c1 is a c2 that op1 a c3 and that op2 i1
and that op3 something that op4 a c4.
Every c2 that op1 a c3 and that op2 i1
and that op3 something that op4 a c4 is a c1.

Every c1 is a c2 that op1 a c3 and that op2 nothing but c4.
Every c2 that op1 a c3 and that op2 nothing but c4 is a c1.

Every c1 is a c2 that op1 a c3 that op2 a c4.
Every c2 that op1 a c3 that op2 a c4 is a c1.
Every c1 is a c2 that op1 a c3.

Every c1 is a c2 that op1 a c3. Every c2 that op1 a c3 is a c1.

Every c1 is a c2 that op1 at least {NUM} c3.
Every c2 that op1 at least {NUM} c3 is a c1.

Every c1 is a c2 that op1 at most {NUM} thing and that
op2 at least {NUM} thing.

Every c1 is a c2 that op1 at most {NUM} thing.

Every c1 is a c2 that op1 exactly 2 c3 and that op2
a c4 that op3 a c5 that op4 something that op5 a c6.
Every c2 that op1 exactly 2 c3 and that op2
a c4 that op3 a c5 that op4 something that op5 a c6 is a c1.

Every c1 is a c2 that op1 exactly 2 c3 and that op2
a c4 that op3 something that op4 a c5 that op5 a c6.
Every c2 that op1 exactly 2 c3 and that op2
a c4 that op3 something that op4 a c5 that op5 a c6 is a c1.

Every c1 is a c2 that op1 nothing but c3 and that op2 exactly {NUM} c4.
Every c2 that op1 nothing but c3 and that op2 exactly {NUM} c4 is a c1.

Every c1 is a c2 that op1 nothing but c3.

Every c1 is a c2 that op1 nothing but c3.
Every c2 that op1 nothing but c3 is a c1.

Every c1 is a c2 that op1 something and that op2 a c3 and that op3 a c4.
Every c2 that op1 something and that op2 a c3 and that op3 a c4 is a c1.

Every c1 is a c2 that op1 something and that op2 a c3.

Every c2 that op1 something and that op2 a c3 is a c1.

Every c1 is a c2 that op1 something and that op2 something that op3 a c3.
Every c2 that op1 something and that op2 something that op3 a c3 is a c1.

Every c1 is a c2 that op1 something that op2 a c3.
Every c2 that op1 something that op2 a c3 is a c1.

Every c1 is a c2. Every c2 is a c1.

Every c1 is something that op1 a c2 and that op2 a c3 and that dp1 dt1.
Every c1 is something that op1 a c2 and that op2 a c3 and that op3
a c4 and that op4 a c5 and that op5 a c6.
Everything that op1 a c2 and that op2 a c3 and that op3
a c4 and that op4 a c5 and that op5 a c6 is a c1.

Every c1 is something that op1 a c2 and that op2
a c3 and that op3 a c4 and that op4 a c5.
Everything that op1 a c2 and that op2
a c3 and that op3 a c4 and that op4 a c5 is a c1.

Every c1 is something that op1 a c2 and that op2 a c3 and that op3
a c4 and that op4 c5 and that op5 c6.
Everything that op1 a c2 and that op2 a c3 and that op3 a c4 and
that op4 c5 and that op5 c6 is a c1.

Every c1 is something that op1 a c2 and that op2 a c3 and that op3
a c4 and that op4 c5.
Everything that op1 a c2 and that op2 a c3 and that op3 a c4 and
that op4 c5 is a c1.

Every c1 is something that op1 a c2 and that op2 a c3 and that op3 a c4.

Every c1 is something that op1 a c2 and that op2 a c3 and that op3 a c4.
Everything that op1 a c2 and that op2 a c3 and that op3 a c4 is a c1.

Every c1 is something that op1 a c2 and that op2 a c3 and that op3
c4 and that op4 something that op5 a c5.
Everything that op1 a c2 and that op2 a c3 and that op3 c4 and
that op4 something that op5 a c5 is a c1.

Every c1 is something that op1 a c2 and that op2 a c3 and that op3
c4 and that op4 something that op5 i1.
Everything that op1 a c2 and that op2 a c3 and that op3 c4 and
that op4 something that op5 i1 is a c1.

Every c1 is something that op1 a c2 and that op2 a c3 and that op3 c4.

Every c1 is something that op1 a c2 and that op2 a c3 and that op3
something that op4 a c4.
Everything that op1 a c2 and that op2 a c3 and that op3 something
that op4 a c4 is a c1.

Every c1 is something that op1 a c2 and that op2 a c3.

Every c1 is something that op1 a c2 and that op2 a c3.
Everything that op1 a c2 and that op2 a c3 is a c1.
```

```
Every c1 is something that op1 a c2 and that op2 nothing but c3.
Everything that op1 a c2 and that op2 nothing but c3 is a c1.

Every c1 is something that op1 a c2 and that op2 something that is a c3
or that is a c4.
Everything that op1 a c2 and that op2 something that is a c3
or that is a c4 is a c1.

Every c1 is something that op1 a c2 and that op2 something that op3 a c3.

Every c1 is something that op1 a c2 and that op2 something that op3 a c3.
Everything that op1 a c2 and that op2 something that op3 a c3 is a c1.

Every c1 is something that op1 a c2 or that is something and that op2 a c3.
Everything that op1 a c2 or that is something and that op2 a c3 is a c1.

Every c1 is something that op1 a c2 or that op2 a c3.

Every c1 is something that op1 at least 2 things and that op2 a c2.

Every c1 is something that op1 at least {NUM} things and that op2
at least {NUM} things and that op3 at least {NUM} things and
that op4 at least {NUM} things.

Every c1 is something that op1 c2 and that op2 c3 and
that op3 c4 and that op4 c5.

Every c1 is something that op1 c2 or that op2 c3 or that op3 c4 or that op4 c5.
Everything that op1 c2 or that op2 c3 or that op3 c4 or that op4 c5 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 1
c6 and that op6 exactly {NUM} c7.
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and that op3
exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly {NUM} c6 and
that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 1
c6 and that op6 exactly {NUM} c7.
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and that op3
exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly {NUM} c6 and
that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 999
c6 and that op6 exactly {NUM} c7.
```

```
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and that op3
exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly {NUM} c6 and
that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 999
c6 and that op6 exactly {NUM} c7.
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and that op3
exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly {NUM} c6 and
that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 1
c6 and that op6 exactly {NUM} c7.
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and that op3
exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly {NUM} c6 and
that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 999
c6 and that op6 exactly {NUM} c7.
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and that op3
exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly {NUM} c6 and
that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 999
c6 and that op6 exactly {NUM} c7.
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and that op3
exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly {NUM} c6 and
that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 1
c6 and that op6 exactly {NUM} c7.
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and that op3
exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly {NUM} c6 and
that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 1
c6 and that op6 exactly {NUM} c7.
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and that op3
exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly {NUM} c6 and
that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
```

```
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 999
c6 and that op6 exactly {NUM} c7.
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 999
c6 and that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and
that op3 exactly {NUM} c4 and that op4 exactly {NUM} c5 and that op5 exactly 999
c6 and that op6 exactly {NUM} c7.
Everything that op1 exactly {NUM} c2 and that op2 exactly {NUM} c3 and that op3
exactly {NUM} c4 and that op4 exactly {NUM} c5 and that
op5 exactly {NUM} c6 and that op6 exactly {NUM} c7 is a c1.

Every c1 is something that op1 exactly 2 c2 and that op2 a c3 that op3 a c4
that op4 something that op5 a c5. Everything that op1 exactly 2 c2 and that op2
a c3 that op3 a c4 that op4 something that op5 a c5 is a c1.

Every c1 is something that op1 exactly 2 c2 and that op2 a c3 that op3 something
that op4 a c4 that op5 a c5.
Everything that op1 exactly 2 c2 and that op2 a c3 that op3 something
that op4 a c4 that op5 a c5 is a c1.

Every c1 is something that op1 exactly {NUM} things and that op2 a c2
and that op3 a c3 and that op4 a c4.
Everything that op1 exactly {NUM} things and that op2 a c2
and that op3 a c3 and that op4 a c4 is a c1.

Every c1 is something that op1 i1 and that op2 something that op3 i2.
Everything that op1 i1 and that op2 something
that op3 i2 is a c1.

Every c1 is something that op1 nothing but c2 and that is not a c3
and that op2 at least 2 c4.
Everything that op1 nothing but c2 and that is not a c3
and that op2 at least 2 c4 is a c1.

Every c1 is something that op1 nothing but c2 and that op2 nothing but c3.

Every c1 is something that op1 something and that op2 a c2 and that op3 a c3.
Everything that op1 something and that op2 a c2 and that op3 a c3 is a c1.

Every c1 is something that op1 something and that op2 a c2 and that op3 c3.
Everything that op1 something and that op2 a c2 and that op3 c3 is a c1.

Every c1 is something that op1 something and that op2 a c2.
Everything that op1 something and that op2 a c2 is a c1.
```

```
Every c1 is something.

Every c1 is something. Everything is a c1.

Every c1 op1 a c2 that dp1 dt1.

Every c1 op1 a c2 that is not a c3 and that is not a c4.

Every c1 op1 a c2 that op2 a c3 and that op3 a c4.
Everything that op1 a c2 that op2 a c3 and that op3 a c4 is a c1.

Every c1 op1 a c2 that op2 a c3 and that op3 something that op4 a c4.
Everything that op1 a c2 that op2 a c3 and that op3 something that op4
a c4 is a c1.

Every c1 op1 a c2 that op2 a c3 that op3 a c4 and that op4 a c5.

Every c1 op1 a c2 that op2 a c3 that op3 a c4.

Every c1 op1 a c2 that op2 a c3.

Every c1 op1 a c2 that op2 a c3. Everything that op1 a c2 that op2 a c3 is a c1.

Every c1 op1 a c2 that op2 nothing but c3.

Every c1 op1 a c2.

Every c1 op1 a c2. Everything that op1 a c2 is a c1.

Every c1 op1 at least {NUM} c2.

Every c1 op1 at least {NUM} thing.

Every c1 op1 at least 2 c2.

Every c1 op1 at least 2 things.

Every c1 op1 at least {NUM} things.

Every c1 op1 at least {NUM} things.
Everything that op1 at least {NUM} things is a c1.

Every c1 op1 at most {NUM} c2.

Every c1 op1 at most {NUM} thing.

Every c1 op1 c2.
Every c1 op1 exactly {NUM} c2.

Every c1 op1 exactly {NUM} c2. Everything that op1 exactly {NUM} c2 is a c1.

Every c1 op1 exactly {NUM} thing.

Every c1 op1 exactly 2 c2.

Every c1 op1 exactly {NUM} c2.

Every c1 op1 exactly {NUM} c2. Everything that op1 exactly {NUM} c2 is a c1.

Every c1 op1 exactly {NUM} things.

Every c1 op1 i1.

Every c1 op1 something that is a c2 or that is a c3.

Every c1 op1 something that op2 a c2 and that op3 a c3 and that op4 a c4.
Everything that op1 something that op2 a c2 and that op3 a c3 and that op4
a c4 is a c1.

Every c1 op1 something that op2 a c2 and that op3 a c3.
Everything that op1 something that op2 a c2 and that op3 a c3 is a c1.

Every c1 op1 something that op2 a c2 and that op3 something that op4 a c3.
Everything that op1 something that op2 a c2 and that op3 something that op4
a c3 is a c1.

Every c1 op1 something that op2 a c2.

Every c1 op1 something that op2 a c2.
Everything that op1 something that op2 a c2 is a c1.

Every c1 op1 something that op2 c2.

Every c1 op1 something that op2 nothing but c2.

Every c1 op1 something that op2 something that op3 a c2.
Everything that op1 something that op2 something that op3 a c2 is a c1.

Every c1 op1 something that op2 something.

Every c1 op1 something.

Every c1 op1 something. Everything that op1 something is a c1.
Every c2 is a c2.

Everything that is dp1 by a c1 is something that is not something.

Everything that is op1 by a c1 is a c2.

Everything that is op1 by a c1 is a c2.
Everything that op1 nothing but c2 is a c1.

Everything that is op1 by a c1 is something that is a c2 or that is a c3.

Everything that is op1 by a c1 is something that is a c2 or that is a c3.
Everything that op1 nothing but things that are a c2 or that are a c3 is a c1.

Everything that is op1 by a c1 is something that is not something.

Everything that is op1 by a c1 is something.

Everything that op1 a c1 and that does not op2 a c2 op3 a c3.
Everything that op3 a c3 is something that op1 a c1 and that does not op2 a c2.

No c1 is a c2. Everything that is not a c2 is a c1.

No c1 op1 a c2.

No c1 op1 something.

Nothing is a c1. Everything that is not something is a c1.
```

Appendix E

Presupposition tests for query templates

```
Query template 1: SELECT ?x WHERE {
  ?x rdfs:subClassOf <EC1>, [a owl:Restriction; owl:onProperty <PC1>; owl:someValuesFrom <EC2>]}

ASK WHERE {
  [a owl:Class; owl:intersectionOf(
    <EC1> [a owl:Restriction; owl:onProperty <PC1>; owl:someValuesFrom <EC2>])] rdfs:subClassOf owl:Nothing }
ASK WHERE {
  [a owl:Class; owl:intersectionOf(
    <EC1>
    [a owl:Restriction; owl:onProperty <PC1>; owl:allValuesFrom [a owl:Class; owl:complementOf <EC2>]] )]
  rdfs:subClassOf owl:Nothing }
```

```
Query template 2: SELECT DISTINCT * WHERE {
  <E3> rdfs:subClassOf [a owl:Restriction; owl:onProperty <IS_EC2>; owl:someValuesFrom ?x].
  ?x rdfs:subClassOf <EC1>. filter(?x != <EC1>) }

ASK WHERE {
  [a owl:Restriction; owl:onProperty <IS_EC2>; owl:someValuesFrom <EC1>] rdfs:subClassOf owl:Nothing }
ASK WHERE {
  [a owl:Restriction; owl:onProperty <IS_EC2>; owl:allValuesFrom [a owl:Class; owl:complementOf <EC1>]]
  rdfs:subClassOf owl:Nothing }
```

```
Query template 3: SELECT DISTINCT * WHERE {
  <EC1> rdfs:subClassOf [a owl:Restriction ; owl:onProperty <PC1> ; owl:hasValue ?x ] }

ASK WHERE {[a owl:Restriction ; owl:onProperty <PC1> ; owl:hasValue owl:Thing ] rdfs:subClassOf owl:Nothing}
```

```
Query template 4: SELECT DISTINCT * WHERE {
  <EC1> rdfs:subClassOf [ a owl:Restriction ; owl:onProperty <PC1> ; owl:someValuesFrom ?x ]

ASK WHERE {
  [a owl:Restriction; owl:onProperty <PC1>; owl:someValuesFrom owl:Thing] rdfs:subClassOf owl:Nothing }
ASK WHERE {
  [a owl:Restriction; owl:onProperty <PC1>; owl:allValuesFrom owl:Nothing] rdfs:subClassOf owl:Nothing }
```

```
Query template 5: SELECT DISTINCT * WHERE {
  <EC2> rdfs:subClassOf [a owl:Restriction ; owl:onProperty <HAS_EC1> ; owl:hasValue ?x ] }

ASK WHERE {
  [a owl:Restriction ; owl:onProperty <HAS_EC1> ; owl:hasValue owl:Thing ] rdfs:subClassOf owl:Nothing}
```

```
Query template 6: SELECT DISTINCT * WHERE {
  <EC2> rdfs:subClassOf [ a owl:Restriction ; owl:onProperty <PC1> ; owl:someValuesFrom ?x ]
  . ?x rdfs:subClassOf <EC1> . filter(?x != <EC1>) }

ASK WHERE {
  [a owl:Restriction; owl:onProperty <PC1>; owl:someValuesFrom <EC1>] rdfs:subClassOf owl:Nothing }
ASK WHERE {
  [a owl:Restriction; owl:onProperty <PC1>; owl:allValuesFrom [a owl:Class; owl:complementOf <EC1>]]
  rdfs:subClassOf owl:Nothing }
```

```
Query template 7: SELECT DISTINCT * WHERE {
  <EC2> rdfs:subClassOf [ a owl:Restriction ; owl:onProperty <PC1> ; owl:someValuesFrom ?x ]

ASK WHERE {
  [a owl:Restriction; owl:onProperty <PC1>; owl:someValuesFrom owl:Thing] rdfs:subClassOf owl:Nothing }
ASK WHERE {
  [a owl:Restriction; owl:onProperty <PC1>; owl:allValuesFrom owl:Nothing] rdfs:subClassOf owl:Nothing }
```

```
Query template 8: SELECT DISTINCT * WHERE {
  ?x rdfs:subClassOf <EC1>, [a owl:Restriction ; owl:onProperty <IS_EC2>; owl:someValuesFrom <EC3> ] .
  filter(?x != owl:Nothing) }

ASK WHERE {
  [a owl:Class; owl:intersectionOf(
    <EC1>
    [a owl:Restriction; owl:onProperty <IS_EC2>; owl:someValuesFrom <EC3>])] rdfs:subClassOf owl:Nothing }
ASK WHERE {
  [a owl:Class; owl:intersectionOf(
    <EC1>
    [a owl:Restriction; owl:onProperty <IS_EC2>; owl:allValuesFrom [a owl:Class; owl:complementOf <EC3>]] )]
  rdfs:subClassOf owl:Nothing }
```

```
Query template 9: SELECT DISTINCT ?x WHERE {
  ?x rdfs:subClassOf <EC1>, [a owl:Restriction; owl:onProperty <PC1>; owl:someValuesFrom <EC2>]}

ASK WHERE {
  [a owl:Class; owl:intersectionOf(
    <EC1>
    [a owl:Restriction; owl:onProperty <PC1>; owl:someValuesFrom <EC2>])] rdfs:subClassOf owl:Nothing }
ASK WHERE {
  [a owl:Class; owl:intersectionOf(
    <EC1>
    [a owl:Restriction; owl:onProperty <PC1>; owl:allValuesFrom [a owl:Class; owl:complementOf <EC2>]] )]
  rdfs:subClassOf owl:Nothing }
```

```
Query template 10: SELECT DISTINCT * WHERE {
  ?x rdfs:subClassOf
    [rdf:type owl:Restriction; owl:onProperty <HAS_EC4>; owl:someValuesFrom <EC4>].
  ?x  rdfs:subClassOf
      [ rdf:type owl:Restriction ; owl:onProperty <HAS_EC3> ; owl:someValuesFrom <EC3>] }

ASK WHERE { owl:intersectionOf(
    [rdf:type owl:Restriction; owl:onProperty <HAS_EC4>; owl:someValuesFrom <EC4>]
    [rdf:type owl:Restriction; owl:onProperty <HAS_EC3>; owl:someValuesFrom <EC3>]
) rdfs:subClassOf owl:Nothing}

ASK WHERE { owl:intersectionOf(
  [rdf:type owl:Restriction; owl:onProperty <HAS_EC4>; owl:someValuesFrom <EC4>]
  [rdf:type owl:Restriction; owl:onProperty <HAS_EC3>; owl:allValuesFrom [
    a owl:Class; owl:complementOf <EC3>]]
) rdfs:subClassOf owl:Nothing}

ASK WHERE { owl:intersectionOf(
  [rdf:type owl:Restriction; owl:onProperty <HAS_EC3>; owl:someValuesFrom <EC3>]
  [rdf:type owl:Restriction; owl:onProperty <HAS_EC4>; owl:allValuesFrom [
    a owl:Class; owl:complementOf <EC4>]]
) rdfs:subClassOf owl:Nothing}
```

```
Query template 11: SELECT DISTINCT * WHERE {?x rdfs:subClassOf <EC1> . ?x rdfs:subClassOf <EC2>}

ASK WHERE {[a owl:Class; owl:intersectionOf(<EC1> <EC2>] rdfs:subClassOf owl:Nothing }
ASK WHERE {
  [a owl:Class; owl:intersectionOf(<EC1> [a owl:Class; owl:complementOf <EC2>]] rdfs:subClassOf owl:Nothing }
```

Appendix F

BigCQ synonym sets

The following lines present how synonym sets are expanded to produce multiple CQ patterns. Each line consists of a synonym set id enclosed in square brackets, which is followed by a colon and a list of phrases that are used to materialize synonym set ids.

```
[WHAT]: [what, which],
[THE SAME AS]: [equal to, identical with, the same as],
[REGARD]: [regard, count, classify, categorize, consider],
[CATEGORY]: [category, class, type, group],
[DOES]: [does, did, do],
[EVERY]: [every, all, each],
[CAN]: [can, could, may],
[REGARDED]: [regarded, counted, classified, categorized, considered],
[IS THERE]: [is there, is there any],
[IT]: [it, that],
[THINGS]: [things, objects, entities, categories],
[IS]: [is, are],
[A KIND]: [a kind, a type, a subset, an example, a sample, a sort, a form,
           a category, a specialization],
[KIND]: [kind, type, subset, example, sort, form, specialization],
[KINDS]: [kinds, types, subsets, examples, sorts, forms, specializations],
[I]: [i, one, we],
[CATEGORIES]: [categories, classes, types],
[THING]: [thing, object, entity],
[THE TYPES]: [the types, the kinds, examples, the forms, the categories,
          the specializations],
[TYPES]: [types, kinds, examples, forms, categories, specializations],
[CLASSIFIED]: [classified, categorized, assigned],
[ARE THERE]: [are there, are available, are proposed, are described, exist]
```

Appendix G

CQ templates used to construct BigCQ

G.1 SPO + Subsumption

List of CQ templates used to generate questions related to the SPO type and class subsumption.

ASK

- `[DOES] {LHS} {VERB} {RHS}?`
- `[DOES] [EVERY] {LHS} {VERB} {RHS}?`
- `[CAN] {LHS} {VERB} {RHS}?`
- `[CAN] [EVERY] {LHS} {VERB} {RHS}?`
- `[CAN] {LHS} be [REGARDED] as something that {VERB} {RHS}?`
- `[IS THERE] {LHS} that [CAN] {VERB} {RHS}?`
- `[IS THERE] {LHS} that {VERB} {RHS}?`
- `[IS THERE] {LHS} that [CAN] be [REGARDED] as {VERB} {RHS}?`
- `Is [IT] true that {LHS} {VERB} {RHS}?`

SELECT LHS

- `[WHAT] {VERB} {RHS}?`
- `[WHAT] [THINGS] {VERB} {RHS}?`
- `[WHAT] [KIND] of [THINGS] {VERB} {RHS}?`

SELECT COUNT LHS

- `How many [CATEGORIES] of [THINGS] {VERB} {RHS}?`

SELECT VERB

- `What {LHS} [CAN] do with {RHS}?`
- `What {RHS} [CAN] do with {LHS}?`

- `What {LHS} [DOES] with {RHS}?`
- `What {RHS} [DOES] with {LHS}?`
- `What relates {LHS} and {RHS}?`
- `What can {LHS} do to {RHS}?`
- `What relation between {LHS} and {RHS} exists?`
- `[WHAT] is the relation between {LHS} and {RHS}?`

SELECT COUNT VERB

- `How many relations [ARE THERE] between {LHS} and {RHS}?`
- `How many things [CAN] {LHS} do with {RHS}?`

SELECT RHS

- `[WHAT] [DOES] {LHS} {VERB}?`
- `[WHAT] {LHS} [CAN] [I] {VERB}?`
- `[WHAT] [THINGS] [DOES] {LHS} {VERB}?`
- `[WHAT] [KIND] of [THINGS] [DOES] {LHS} {VERB}?`
- `[WHAT] is {VERB} by {LHS}?`

SELECT COUNT RHS

- `How many [CATEGORIES] of [THINGS] [DOES] {LHS} {VERB}?`

G.2 SS + Subsumption

List of CQ templates used to generate questions related to the SS type and class subsumption.

ASK

- `[IS] [EVERY] {LHS} [A KIND] of {RHS}?`
- `[IS] [EVERY] {LHS} {RHS}?`
- `[CAN] {LHS} be [A KIND] of {RHS}?`
- `[CAN] {LHS} be {RHS}?`
- `[CAN] {LHS} be [REGARDED] as {RHS}?`
- `[CAN] {LHS} be [REGARDED] as [A KIND] of {RHS}?`
- `[CAN] [I] [REGARD] {LHS} as {RHS}?`
- `[IS THERE] {LHS} that [CAN] be [A KIND] of {RHS}?`
- `[IS THERE] {LHS} that [CAN] be {RHS}?`

- `[IS THERE] {LHS} that [CAN] be [REGARDED] as [A KIND] of {RHS}?`
- `[IS THERE] {LHS} that [CAN] be [REGARDED] as {RHS}?`
- `[DOES] {LHS} being [A KIND] of {RHS} exist?`
- `[DOES] {LHS} being {RHS} exist?`
- `Is [IT] true that {LHS} is [A KIND] of {RHS}?`
- `Is [IT] true that {LHS} is {RHS}?`

SELECT LHS

- `[WHAT] [THINGS] are {RHS}?`
- `[WHAT] are [CATEGORIES] of {RHS}?`
- `[WHAT] are the [CATEGORIES] of {RHS}?`
- `[WHAT] are the main [CATEGORIES] of {RHS}?`
- `[WHAT] are the main [TYPES] of {RHS}?`
- `[WHAT] [IS] {RHS}?`
- `[WHAT] [CATEGORIES] of {RHS} [ARE THERE]?`
- `[WHAT] main [CATEGORIES] of {RHS} [ARE THERE]?`
- `[WHAT] [KINDS] of {RHS} [ARE THERE]?`
- `[WHAT] {RHS} [ARE THERE]?`
- `[WHAT] main [KINDS] of {RHS} [ARE THERE]?`

SELECT COUNT LHS

- `How many [KINDS] [DOES] {RHS} have?`
- `How many [KINDS] of {RHS} [ARE THERE]?`

SELECT RHS

- `[WHAT] [KIND] of [THING] [IS] [EVERY] {LHS}?`
- `[WHAT] [THING] [IS] [EVERY] {LHS}?`
- `[WHAT] is the supertype of {LHS}?`
- `To [WHAT] [CATEGORY] [CAN] {LHS} be [CLASSIFIED] to?`
- `[WHAT] [CATEGORY] [DOES] {LHS} belong to?`
- `[WHAT] [IS] {LHS}?`

SELECT COUNT RHS

- `How many supertypes does {LHS} belong to?`
- `How many supertypes does {LHS} have?`
- `How many supertypes of of {LHS} exists?`
- `To how many [CATEGORIES] [CAN] {LHS} be [CLASSIFIED] to?`

G.3 SPO + Equivalence

List of CQ templates used to generate questions related to the SPO type and class equivalence.

ASK

- `[CAN] [EVERY] {LHS} be considered [THE SAME AS] a thing that {VERB} {RHS}?`
- `[DOES] [EVERY] {LHS} equal a thing that {VERB} {RHS}?`
- `[CAN] {LHS} be [THE SAME AS] something that {VERB} {RHS}?`
- `[IS THERE] {LHS} that [CAN] be [THE SAME AS] things that can {VERB} {RHS}?`
- `[IS THERE] {LHS} that is [THE SAME AS] things that {VERB} {RHS}?`
- `[IS THERE] {LHS} that [CAN] be [REGARDED] as [THE SAME AS] things that {VERB} {RHS}?`
- `Is [IT] true that {LHS} [IS] [THE SAME AS] things that {VERB} {RHS}?`

SELECT LHS

- `[WHAT] is [THE SAME AS] things that {VERB} {RHS}?`
- `[WHAT] [THINGS] are [THE SAME AS] things that {VERB} {RHS}?`
- `[WHAT] [KIND] of [THINGS] are [THE SAME AS] to things that {VERB} {RHS}?`

SELECT COUNT LHS

- `How many [CATEGORIES] of [THINGS] are [THE SAME AS] things that {VERB} {RHS}?`

SELECT VERB

- `What {LHS} [CAN] do with {RHS}?`
- `What {RHS} [CAN] do with {LHS}?`
- `What {LHS} [DOES] with {RHS}?`
- `What {RHS} [DOES] with {LHS}?`
- `What relates {LHS} and {RHS}?`
- `What can {LHS} do to {RHS}?`

- `What relation between {LHS} and {RHS} exists?`
- `[WHAT] is the relation between {LHS} and {RHS}?`

SELECT COUNT VERB

- `How many relations [ARE THERE] between {LHS} and {RHS}?`
- `How many things [CAN] {LHS} do with {RHS}?`

SELECT RHS

- `[WHAT] is [THE SAME AS] things that {LHS} {VERB}?"`
- `[WHAT] [THINGS] are [THE SAME AS] things that {LHS} {VERB}?`

SELECT COUNT RHS

- `How many [CATEGORIES] of [THINGS] are [THE SAME AS] things that {LHS} {VERB}?`

G.4 SS + Equivalence

List of CQ templates used to generate questions related to the SS type and class equivalence.

ASK

- `[IS] [EVERY] {LHS} [THE SAME AS] {RHS}?`
- `[CAN] {LHS} be [THE SAME AS] {RHS}?`
- `[CAN] {LHS} be [REGARDED] [THE SAME AS] {RHS}?`
- `[CAN] [I] [REGARD] {LHS} [THE SAME AS] {RHS}?`
- `[IS THERE] {LHS} that [IS] [THE SAME AS] {RHS}?`
- `[DOES] {LHS} being [THE SAME AS] {RHS} exist?`
- `Is [IT] true that {LHS} is [THE SAME AS] {RHS}?`
- `Is [IT] true that {LHS} is {RHS}?`

SELECT LHS

- `[WHAT] [THINGS] are [THE SAME AS] {RHS}?`
- `[WHAT] [IS] [THE SAME AS] {RHS}?`

SELECT COUNT LHS

- `How many [KINDS] of {RHS} are equivalent to it?`

SELECT RHS

- `[WHAT] [KIND] of [THING] [IS] [THE SAME AS] {LHS}?`
- `[WHAT] [THING] [IS] [THE SAME AS] {LHS}?`
- `[WHAT] [IS] [THE SAME AS] {LHS}?`

SELECT COUNT RHS

- `How many [KINDS] of {RHS} are equivalent to it?`

List of Figures

List of Tables

Index