Separate-and-Conquer-Lernen von SPARQL-Abfragen

Diplomarbeit von Hossein Rabighomi aus Darmstadt
Oktober 2015

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering

Separate-and-Conquer-Lernen von SPARQL-Abfragen

Vorgelegte Diplomarbeit von Hossein Rabighomi aus Darmstadt

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Dr. Frederik Janssen, Prof. Dr. Heiko Paulheim

Tag der Einreichung:

Erklärung zur Diplomarbeit

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 5. Oktober 2015

(Hossein Rabighomi)

Zusammenfassung

Das Semantic Web bietet einen Ansatz, mit dem die Informationen so strukturiert sind, dass sie sowohl von Menschen als auch von Maschinen interpretiert werden können. Allerdings sind die Information in einem Dataset nicht immer ausreichend. Um dies zu beheben, müssen aus vorhandenen Daten neue Informationen gewonnen werden. Dieser Arbeit beschäftigt sich damit, wie man ein Dataset nutzt, um aus Daten mit Separate-and-Conquer-Algorithmen zu lernen, welche Objekte zu einer Klasse gehören.

Um dies zu realisieren, wurde das Tool `db2seco` implementiert. Das Tool benutzt eine SPARQL-Abfrage, um Daten aus Datasets zu holen, die zwecks Lernens dann klassifiziert werden müssen. Dann werden die Daten an das *SeCo-Tool* weitergereicht. Das *SeCo-Tool* lernt eine Regelmenge, die die zu lernende Klasse repräsentiert. Mit dieser Regelmenge werden die SPAQRL-Abfrage und die Trainingsmenge erweitert.

Eine Evaluation zeigt, dass das Lernen von linked open data mit Seco funktioniert, solange die Daten sinnvoll markiert sind. Bei der Evaluation mit unterschiedlichen Heuristiken zeigt sich, dass die Heuristiken *Accuracy* und *Precision* zu sehr langen SPARQL-Abfragen führen, während mit anderen Heuristiken die SPARQL-Abfragen nur wenig größer werden.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Menge der Daten im Internet wächst sehr schnell, und viele dieser Daten können nicht automatisiert erfasst und verarbeitet werden. Oft sind diese Daten im Web hauptsächlich für Menschen konzipiert und von Ihnen als Informationen nutzbar. Maschinelles Lernen auf solchen Daten ist schwieriger als auf strukturierten Daten. Entsprechend müssen diese Daten aufbereitet werden, bevor sie von Maschinen automatisiert weiter verarbeitet werden können, etwa um neue Informationen zu generieren.

Die Linked Open Data Initiative versucht, Daten in maschinenlesbarer Form bereitzustellen. Dazu wurden Standards definiert, die regeln wie Daten dargestellt und abgefragt werden. So ist im Resource Description Framework [1] festgelegt, wie Daten repräsentiert werden sollen. Dabei kommen eindeutige Resource Identifiers (URIs) zum Einsatz, auf die sich verschiedene Datensätze beziehen. Die Abfragesprache SPARQL wiederum ist der Standard für die Abfrage von RDF-Daten. Mit SPARQL können Abfragen nach Ressourcen und ihren Attributen gestellt werden.

Beide Technologien richten sich allerdings an Experten. Deshalb werden Werkzeuge benötigt, mit denen es Benutzern einfach möglich ist, linked open data abzufragen und zu verändern. Dazu gehört der interaktive Umgang mit SPARQL-Abfragen. Wenn ein Benutzer mit SPARQL eine Anfrage auf RDF auf bestimmte Kriterien hin anpassen will, muss er von Hand diese Kriterien korrekt in SPARQL formulieren. Deshalb werden Werkzeuge benötigt, die den Umgang mit SPARQL erleichtern.

Um mit linked open data zu arbeiten, kann es außerdem sinnvoll sein, neue Daten zu einem Datensatz hinzuzufügen, die sich aus bekannten Daten ableiten lassen. Diese Daten manuell einzutragen wäre zu aufwändig. Deshalb ist es besser, sie automatisch zu lernen. So kann man beispielsweise passend zu einer Abfrage nach Sportlern in einem RDF-Graph mit Personeninformationen erkennen, welche Sportler aus einem bestimmten Land stammen. Ein entsprechendes Attribut für Personen könnte automatisch hinzugefügt werden.

1.2 Zielsetzung und Vorgehen

Dieser Arbeit beschäftigt sich damit, wie man ein Dataset, z.B. DBpedia, nutzt, um aus Daten mit einem Lernverfahren, in diesem Fall Separate-and-Conquer-Algorithmen, bestimme Klassen von Objekten zu lernen. Es können dabei verschieden Daten aus unterschiedlichen Domänen miteinander verbunden werden. Der Ansatz schließt somit die Lücke zwischen maschinellem Lernen und semantischen Technologien.

Um dies zu realisieren, wurde das Tool `db2seco` entwickelt. Das Tool fragt Daten mittels SPARQL-Abfrage aus Datasets ab und stellt sie als Beispieldaten zur Verfügung. Als nächstes werden dem Nutzer die Trainingsbeispiele zwecks Lernen angezeigt und dieser nimmt eine Klasseneinteilung vor. Nachdem die Trainingsbeispiele vom Nutzer klassifiziert worden sind, werden sie an einen Regellerner (SeCo-Algorithmus) weiter gereicht. Die gewonnen Regeln werden in die ursprüngliche SPARQL-Abfrage integriert.

1.3 Aufbau der Arbeit

Kapitel 2 erklärt die Grundlagen zu Semantic Web Technologien und maschinellem Lernen. In Kapitel 3 wird beschrieben, wie die beide Technologien verbunden werden, um aus semantischen Daten anhand eines Regellerners zu lernen. In Kapitel 4 wird die Implementierung des Tools `db2seco` beschrieben. Anschließend wird in Kapitel 5 das Lernen auf Semantic Web Daten evaluiert. Kapitel 6 beschäftigt sich mit verwandten Arbeiten und in Kapitel 7 wird die Arbeit zusammengefasst.

2 Grundlagen

2.1 Semantic Web

Grundsätzlich steht Semantik aus dem Griechischen abgeleitet für „zum Zeichen gehörig“ und ist die Bedeutung eines oder mehrerer Zeichen. Semantik stellt also die Beziehung zwischen den Zeichen und deren Bedeutung dar. Das semantische Web ist ein Konzept der Wissensrepräsentation mit logischen Zusammenhängen und basiert auf von Tim Berners-Lees Idee[1] „*Das Semantische Web ist eine Erweiterung des herkömmlichen Webs, in der Informationen mit eindeutigen Bedeutungen versehen werden, um die Arbeit zwischen Mensch und Maschine zu erleichtern* “ [2].

Das Problem des Web besteht darin, dass eine unüberschaubare Menge an Daten existiert und die Informationen nur für Endnutzer ausgerichtet sind. Somit ist die Bedeutung von Informationen auf einer Webseite für Menschen sehr einfach begreifbar. Anderseits sind ist Bedeutung von Informationen für Maschinen nicht leicht zu erkennen. Wenn zum Beispiel ein Nutzer auf die Wikipedia Seite von Deutschland[2] geht, und auf der rechte Seite sich die Flagge anschaut, ist er in der Lage zu erkennen, dass das Bild die Flagge von Deutschland repräsentiert. Im Gegensatz dazu kann die Maschine nur feststellen, dass sich dort ein Bild befindet. Die Fähigkeit der Maschine bezieht sich auf das Speichern und die exakte Wiedergabe von Informationen. Die Maschine kann von der Bedeutung den Informationen keine Zusammenhänge erkennen und daraus keine Schlussfolgerungen ziehen.

Als Lösung für dieses Problem bietet sich an, die Informationen mit Wissen zu versehen. Es werden Wissensbereiche, Beziehungen sowie Eigenschaften oder auch Ableitungen solcher Bereiche dargestellt und auf einer semantischen Ebene verknüpft. Das Semantic Web bietet der Maschine die Fähigkeit, Daten in einem gegebenen Rahmen interpretieren zu können und daraus Schlüsse zu zielen. Um solche Wissensrepräsentation zu realisieren, werden verschiedene Bereiche der Wissenschaft miteinander verknüpft. Logik stellt die formale Struktur und daraus resultierende Rückschlüsse dar, allerdings ist es nicht möglich, festzustellen, ob eine Aussage überflüssig, redundant oder inkonsistent ist. Ontologien definieren Objekte, ohne die Aussagen nicht klar ausformuliert werden können.

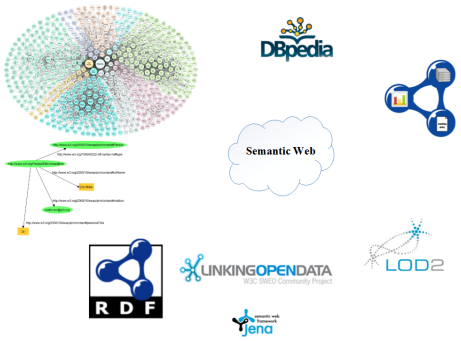


Abbildung 1: RDF, XML und digitale Signaturen als Baustein des Semantic Webs

Die wichtigste Komponenten des semantischen Webs sind in Abbildung 1 dargestellt. Viele davon wurden vom W3C[3] entwickelt, Die Basistechnologien des Semantic Webs sind die Extensible Markup Language (XML), das Resource Description Framework (RDF) und die Web Ontology Language (OWL)[3].

Struktur des Semantic Web

Das Struktur des Semantic Web ist in Abbildung 2 als Schichtenmodell dargestellt. Anhand dieser Abbildung sollen dann im weiteren Verlauf die wichtigsten Techniken vorgestellt werden. Das Modell besteht aus einzelnen aufeinander aufbauenden Schichten. Es wurde von W3C-Konsortium[4] empfohlen und besteht aus 7 Schichten, die aufeinander aufbauen. Das W3C hat als Markup- und Meta-Sprache *XML* (eXtensible Markup Language)[5] empfohlen, mit dem man

1 http://www.w3.org/People/Berners-Lee/
2 *https://de.wikipedia.org/wiki/Deutschland*
3 www.w3.org
4 http://www.w3.org/
5 http://www.w3.org/TR/xml/

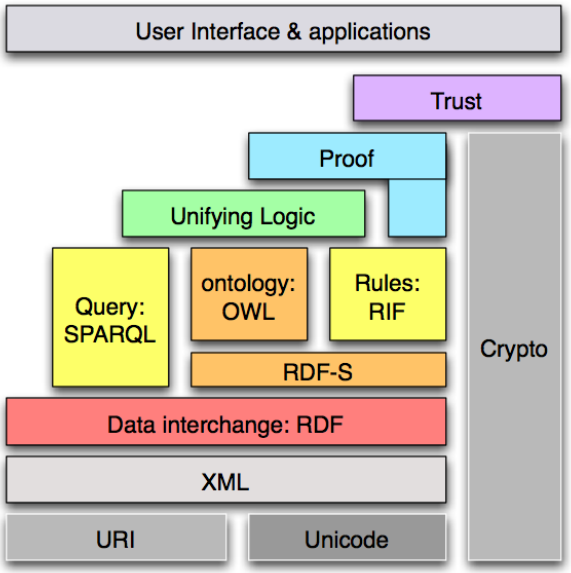


Abbildung 2: Schichtenmodell des Semantic Web (Berners-Lee, 2009) [4]

eine logische Struktur von Dokumenten festlegen kann. Eine einheitliche Formatierung und Darstellung von Dokumenten mittels XML im Gegensatz zu HTML erlaubt die Dokumentstruktur so zu gestalten, dass man sie für verschiedene Applikation dynamisch generieren und beliebig erweitern kann. Das Semantic Web baut wie das klassisches Web auf Internet-Protokolle wie HTTP. Unicode [6] ist als Kodierungsstandard für Daten vorgesehen. Unicode ist systemunabhängig, programmunabhängig und sprachunabhängig und bietet die Grundlage für eine weltweit einheitliche Kodierung. *URI*[7] (Uniform Resource Identifier) dient für eindeutige Bezeichnung einer abstrakten oder physischen Ressource. *URL*(Uniform Resource Locator) ist eine Untermenge von URI und dient zur Lokalisierung und eindeutige Benennung von Ressourcen im Internet z.B. *http://www.w3.org/People/Berners-Lee/* . Spezifikationssprachen wie *RDF* (Resource Description Framework)[8] und *OWL* (Web Ontology Language)[3] sind weitere Standards. RDF baut auf der XML-Schicht auf und ist eine plattformunabhängige Beschreibungssprache. Sie definiert den semantischen Zusammenhang zwischen Daten. Um die Bedeutung von Ressourcen sowie mögliche Beziehungen zwischen ihnen zu definieren, wird *RDF-Schema* eingesetzt. Die *OWL* basiert auf RDF/RDF-Schema. OWL definiert eine Sprache für Beschreibung von Ontologien. Mit *Proof* kann man Inferieren einer Maschine genauer nachvollziehen. Proof untersucht das Web, ob Regeln aus Logik-Schicht sich bestätigten oder widerlegen. Die Regeln beschreiben die Beziehungen wie einzelne Klassen und eigenschaften im RDF, RDF-Schema und OWL zueinander stehen. *Trust* " Vertrauens-Schicht " entscheidet, ob eine Aussage veröffentlicht werden darf. Dabei wichtig ist, dass jeder was aussagen kann und es veröffentlichen darf. Es geht darum, die Maschine zu unterstützen, indem Werkzeuge bereitgestellt werden, mit denen die Maschine entscheiden kann, welchen Quellen man vertrauen kann oder nicht[5, 6]. *SPARQL* ist die Anfragesprache für RDF (vgl. Abschnitt 2.1.1) subsubsectionXML *XML* ist die Abkürzung für „ Extensible Markup Language "[9], was ins Deutsche übersetzt „erweiterbare Auszeichnungssprache " bedeutet. Sie ist eine Dokumentauszeichnungssprache und wird in allen Gebieten der Text- oder Datenverarbeitung verwendet. XML wird als Metasprache verstanden, um andere Sprachen zu realisieren, die alle eine gemeinsame Syntax vereint. Die Bedeutung der Sprachelemente hängt nur von der jeweiligen Definition der Domäne ab. Die Benennung eines Elements gibt keine semantische Information hervor, daher muss die Bedeutung explizit zugeordnet werden. Ein XML-Dokument besteht aus dem Prolog, genau einem Element und beliebig vielen Kommentaren. Der Prolog, enthält die Deklaration des Dokumentes, danach folgen die eigentlichen Daten. Die Deklaration beinhaltet die XML-Version, Codierungsart sowie weitere Informationen. Z.B :

```
<?xml version="1.0" encoding="UTF-8"?>
```

Elemente : Elemente werden auch Tags oder Knoten genannt. Jedes XML-Dokument besitzt exakt ein Wurzelelement. Das Wurzelelement umfasst alle andere Elemente. Elemente beginnen mit dem "<"-Zeichen und enden mit dem ">"-Zeichen. Ein Element beschreibt die in ihm enthaltenen Daten. Elemente können auch andere Elemente und Attribute enthalten.

6 http://unicode.org/
7 http://tools.ietf.org/html/rfc3986
8 http://www.w3.org/TR/rdf11-mt/
9 http://www.edition-w3.de/TR/2000/REC-xml-20001006/

Diese Elemente können Text oder Attribute enthalten. Das folgende Beispiel zeigt ein Element das folgende Beispiel zeigt die Definition eines Elements mit der Bezeichnung CurrentWorldRanking mit dem einfachen Typwert Integer.

```
<xs:element name="CurrentWorldRanking" type="xs:integer" />
```

Attribute : Ein Attribut ist eine Definition einfachen Typs, die keine anderen Elemente enthalten kann. Attributen kann auch ein optionaler Standardwert zugewiesen werden. Das folgende Beispiel zeigt die Deklaration eines Attributs mit der Bezeichnung NumberOfGoldMedals, das mit dem einfachen Typ number definiert ist.

```
<xs:attribute name="NumberOfGoldMedals" type="xs:number" />
```

XML-Schema : XML-Schema ist eine eigene Sprache basiert auf XML um eine XML-Dokument zu validieren. Darin sind die Anordnung den Tags festgehalten. Es ist festgelegt, welche Tags notwendig und welche optional sind. XML-Dokumente können gegen solche XML-Schema in Betracht gezogen werden und wenn die XML-Schema per URI erreichbar sind, bedeutet es, dass das XML-Dokument unabhängig von einer speziellen Anwendung ist.

DTD : „Doctype-Definition“ beschreibt die Syntax eines XML-Dokuments. Die DTD wird benötigt um ein XML-Dokument validieren zu können. Sie beschreibt die Definition von Elementen, Attributen, Entities und Notationen. Anhand DTD wird festgelegt, welche Elemente und Attribute an welcher Stelle des Dokuments gültig sind. Ein XML-Dokument kann auf “ Wohlgeformtheit “ und auf “ Gültigkeit“ überprüft werden’, wobei die “Wohlgeformtheit“ sich auf die Verständlichkeit eines Dokuments bezieht und “Gültigkeit“ auf Validierung eines XML-Dokuments gegen eine DTD. Das folgende Beispiel stellt eine wohlgeformte XML-Datei dar. Am Beginn steht die XML-Deklaration, anschließend folgen drei Datenelemente: <Person>...</Person>,<Vorname>...</Vorname> und <Nachname>...</Nachname>. Das Dokument-Element (<Person>...</Person>) umschließt alle anderen Datenelemente. Das Beispiel ist wohlgeformt aber nicht gültig, es da keinen Bezug auf DTD enthält.[7]

```
<?xml version="1.0"?>
  <Person>
    <Vorname>Boris</Vorname>
    <Nachname>Becker</Nachname>
  </Person>
```

2.1.1 RDF

RDF Ressource Description Framework ist eine formale Sprache zur Beschreibung von semantische Informationen, wie etwa die folgende Aussage „Marc Zwiebler spielt Badminton“. Es gibt ein allgemeines RDF-Modell, indem die Beziehungen zwischen Entitäten ausgedrückt sind. RDF ist ein Standard von W3C [1] und wurde dann im Jahr 1999 vom W3C veröffentlicht [8]. RDF ermöglicht es Anwendungen, die über das Semantic Web in einer strukturierten Weise zu kommunizieren. Die Vereinigung von verschiedenen Beschreibungen ist durch diese Struktur ebenfalls möglich. Was auf die gleiche URI zeigt, ist auch das gleiche Konzept. Was mit so einer Ressource in Verbindung steht kann gemeinsam ausgegeben werden. So kann z.B. ursprünglich vorgesehen sein, dass zu einem „BadmintoPlayer“ Name und Gewicht angegeben werden, verweist man auf die gleiche URI und macht Aussagen über Trainer oder Ladensverband, können diese Informationen problemlos verbunden werden. RDF existiert als zusätzliche Schicht über der XML-Schicht in der Semantic Web Struktur. XML ist ein Standard für Strukturierung von Daten, während RDF als Standard zur Darstellung von Semantik konzipiert ist. Man kann sie im Semantic Web kombinieren. Im RDF sind Daten strukturiert und bei der Übertragung werden die Informationen bezüglich Domänen auch weitergeleitet. Infolgedessen bleibt ihre Semantik bestehen. Man kann Daten aus unterschiedlichen Quellen mischen, die sich auf die gleiche Ressourcen beziehen. Durch RDF können Ressourcen in einer maschinenlesbaren und -interpretierbaren Form gespeichert werden. RDF besteht aus einem RDF-Modell(grafisches Modell) zur Repräsentation der erzeugten Metadaten, als einer RDF-Syntax und RDF-Schema, das ein Vokabular zur Definition von Metadaten ist.

2.1.1.1 RDF-Modell

Das Konzept von RDF bezieht sich darauf, dass Aussagen (Statements) über Ressourcen in einem standardisierten Form *„ s hat die Eigenschaft P mit Wert O “* beschrieben sind.

Statement (Aussage): besteht aus Tripeln. Das oben genanntes Beispiel „Marc Zwiebler spielt Badminton “ ist ein Statement in dem Form eines Tripels [9, S.35 ff].

Ein Statement in RDF sieht folgendermaßen aus:

```
Statement = ( Ressource , Property (Ressource) , Wert(Ressource oder Literal) )
Statement = ( Subjekt , Prädikat , Objekt )
```

Das Subjekt und das Prädikat eines Statements müssen eine Ressource oder ein anonymer Knoten(Blank Node) sein. Das Objekt kann eine Ressource, ein anonymer Knoten oder ein Literal sein. Die Abbildung 3 stellt zwei Statements mit jeweils einem Literal als Objekt und einer Ressource als Objekt dar. Die Abbildung 4 zeigt die Schlüsselelemente eines RDF-Tripel.

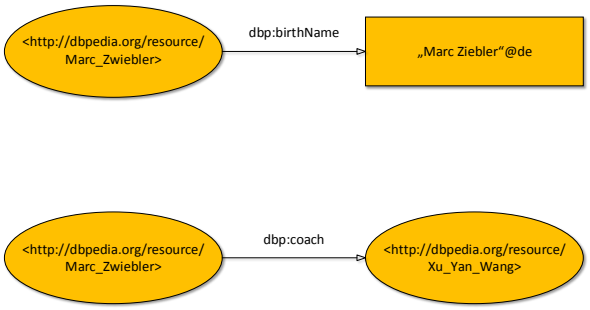


Abbildung 3: Zwei Statements mit unterschiedlichen Objekten

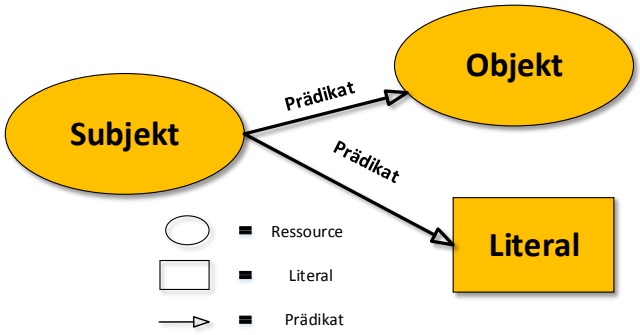


Abbildung 4: RDF-Tripel [10, S.85]

- *Subjekt* (Subject) : Das Subjekt ist allgemein gesprochen die Ressource, die beschrieben wird. Diese ist durch einen URI beschrieben. Zum Beispiel „ Marc Zwiebler " mit URI *http://dbpedia.org/resource/Marc_Zwiebler*

- *Prädikat* (Property): Das Prädikat benennt eine Eigenschaft (z. B. ein Attribut, eine Beziehung oder ein spezifisches Kennzeichen) der Ressource, auf die sich das Triple bezieht und ist ebenfalls durch einen URI identifiziert. Im RDF-Modell stellt das Prädikat die Verbindung zwischen Subjekt und Objekt her. Im RDF-Modell müsste für die Bedeutung des Wortes "'spielt" eine eindeutige URI definiert werden, die beispielsweise so aussehen könnte:

  http://www.example.org/ontologien/spielt = (Aussage) "Marc Zwiebler spielt Badminton"

- *Objekt* (Object) : Das Objekt entspricht dem Wert des Prädikats des beschriebenen Subjekts und kann entweder durch ein *Literal* oder Ressource angegeben werden. Zum Beispiel ist „Badminton" ein Objekt mit URI:

  http://dbpedia.org/resource/Badminton

Ressourcen im Semantic Web bezieht sich sowohl auf reale Objekte (Buch, Personen, HTML-Seiten etc.) als auch auf abstrakte Begriffe und Konzepte (Sport, Wissen, Freiheit etc.)[11, S.10]. Sehr wichtig dabei ist, dass diese Ressourcen sich eindeutig über die URIs identifizieren können. Denn nur so kann eine Idee, ein Konzept, die das Subjekt oder Objekt beschreibt, global definiert werden und einheitlich benutzt werden. Zum Beispiel in der natürlichen Sprache gibt es sehr häufig Homonyme. Die eindeutige Identifizierung der Objekte und Subjekte durch URIs löst so das Problem. Zum Beispiel sind

" Der König wohnt in einem Schloss " und " Das Schloss an meiner Haustür ist kaputt "

zwei Sätze mit dem jeweils selben Wort "‘ Schloss ’", aber mit unterschiedlichen Bedeutungen. Im RDF-Modell ist es möglich, dem Ort „ Schloss “ eine global eindeutige URI zuzuteilen und der Schließvorrichtung „ Schloss “ eine andere globale eindeutige URI, wodurch das Problem behoben wird. Diese URI könnte in diesem Fall so aussehen:

Ort: URI "http://www.example.org/ontology/Ort/Schloss"
Schließvorrichtung: URI "http://www.example.org/ontology/Schliessvorrichtung/Schloss"

Literale : Literale repräsentieren in RDF die Datenwerte und bestehen aus einer Zeichenkette und einen Datentyp. Damit die Bedeutung von Datenwerte unabhängig von der Anwendung immer gleich bleibt, sind für konkrete Datentypen wie Zahlen, Zeitangaben, Wahrheitswerte eine eindeutige Bezeichner in Form einer URI vorgesehen. Dmit ist die Bedeutung von Datenwerte unabhängig von der Anwendung eindeutig bestimmt. Zum Beispiel stellt

"Integer" mit URI "http://www.w3.org/2001/XMLSchema#Integer"

jedem Kontext immer eine Integerzahl dar. Im Gegenteil zu URI einer Person kann Alter und Gewicht hinzugefügt werden, abhängig davon welche Anwendung die Ressource Person in Betracht zieht.[9, S.38]

Es gibt zwei Arten von Literalen: *Plain Literals* und *Typed Literals*.

- *Plain Literals(Untypisierte Literale)* bestehen aus einer Zeichenkette (engl. string) und einer Sprachangabe (engl. language tag). Die Sprachangabe ist gefolgt von „@“Zeichen und immer in Kleinbuchstaben geschrieben. Zum Beispiel

  "Marc Zwiebler"@de

  Wenn zwei Literale mit einer ansonsten identischen Zeichenkette unterschiedliche oder gar keine Sprachangabe haben, gelten sie nicht als gleich. Zum Beispiel:

  "Marc Zwiebler"@de != "Marc Zwiebler"

- *Typed Literals(typisierte Literale)* bestehen aus einer Zeichenkette und einem Datentyp. Zum Beispiel repräsentiert

  "25"^^<http://www.w3.org/2001/XMLSchema#Integer>

  die Zahl „25“ und den zugehörigen Datentyp „Integer“.„ In der Praxis sind allerdings vor allem solche Datentyp-URIs sinnvoll, die allgemein bekannt sind und von vielen Programmen erkannt und unterstützt werden. Daher empfiehlt RDF die Verwendung der Datentypen von XML Schema.“ [9, S.50]

2.1.1.2 RDF-Graph

Ein RDF-Dokument ist eine endliche Menge von RDF-Tripeln und beschreibt einen gerichteten Graphen, den sogenannten RDF-Graph. Er bietet eine graphische Darstellung von RDFs und ist sehr gut für Menschen lesbar. Ein *RDF-Graph* besteht aus Knoten, die die Ressourcen repräsentieren und Kanten, die beschriftet sind. Knoten und Kanten sind mit eindeutigen Bezeichnern (Prädikaten) beschriftet. Abbildung 4 zeigt einen Graph, der aus zwei Knoten und einer Kante besteht.

Ein anderes Beispiel für einen einfachen Graph mit URIs stellt die Abbildung 5 dar, wobei Ellipsen eine Ressource, Rechtecken Literale und eine gerichtete Kante mit einer Beschriftung als Prädikate dienen.

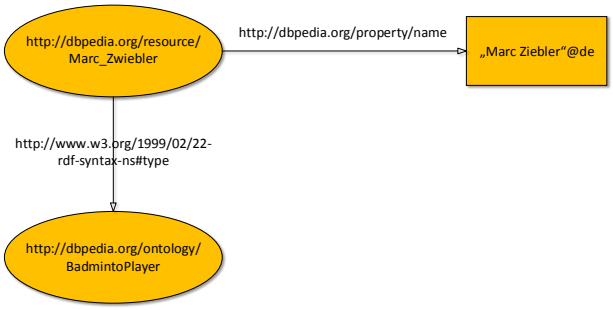


Abbildung 5: Ein RDF-Graph mit URIs

2.1.1.3 Abkürzungen:

In RDF werden alle Ressourcen, Prädikaten und Objekts mit URIs versehen. Wenn man eine Aussage in RDF darstellen möchte, muss man die URIs vollständig ausschreiben. Das RDF-Tripel „Marc Zwiebler hat den Trainer Xu Yan Wang" könnte also wie folgt aussehen:

```
<http://dbpedia.org/resource/Marc_Zwiebler>
<http://dbpedia.org/property/coach>
<http://dbpedia.org/resource/Xu_Yan_Wang>
```

Da diese Schreibweise sehr lang und unübersichtlich ist, bietet RDF die Möglichkeit, ähnlich wie in *XML* Namensräume zu definieren. Zur Vereinfachung verwendet man eine abkürzende Schreibweise, die *QNames* anwendet. Ein QName enthält ein Präfix, das einem namespace-URI zugewiesen worden ist, gefolgt von einem Doppelpunkt und einem lokalen Namen.

```
QName = Präfix : lokaler Name
```

Zum Beispiel, für die folgende URI:

```
URI          =  http://dbpedia.org/resource/Marc_Zwiebler
QName        =  dbr : Marc_Zwiebler
Präfix       :  dbr = http://dbpedia.org/resource/
lokaler Name :  Marc_Zwiebler
```

Somit könnte das oben genannte RDF-Tripel "Marc Zwiebler hat den Trainer Xu Yang Wang" so aussehen:

```
Präfix : dbr : http://dbpedia.org/resource/
Präfix : dbp : http://dbpedia.org/property/

<dbr:Marc_Zwiebler> <dbp:coach> <dbr:XU_Yan_Wang>
```

2.1.1.4 RDF-Syntax

Um den Austausch von Daten, die im RDF dargestellt sind, zu erleichtern, benötigt es eine Serialisierung. Das RDF-Datenmodell ist nicht an eine bestimmte Syntax gebunden und kann daher als verschiedene Arten dargestellt werden. Zu den bekanntesten Notationen zählen *N-Triple*[10], welche eine Untermenge der *N3-Notation*[11] bezeichnen, und *RDF/XML*. Um N-Tripel noch einfacher zu gestalten, wurde die RDF-Syntax *Turtle* entwickelt. RDF/XML ist am meistens Verbreitete

[10] http://www.w3.org/TR/n-triples/
[11] http://www.w3.org/DesignIssues/Notation3.html

Syntax für RDF-Dokumente und Grund dafür ist, dass für viel Programmiersprachen Bibliotheken für den Umgang mit XML zur Verfügung stehen [9, S.40].

- *Turtle-Syntax*: Zeilenbasierte unformatierte Textdarstellung von RDF-Graphen. Die Turtle-Syntax unterscheidet sich von N-Tripel nur darin, dass Abkürzungen verwendet werden und sich daher die Schreibweise wesentlich übersichtlicher darstellt lässt. Daher ist Turtle-Syntax für Menschen besser lesbar. Ein Turtle-Dokument ist wie folgt aufgebaut:

  Deklaration von Präfixen: Am Anfang stehen alle benötigte Präfixe in der Form:

  ```
  @prefix: dbr : <http://dbpedia.org/resource/>
  ```

  daraus kann man sehr bequem eine URI Angabe machen. Zum Beispiel die URI von Marc Zwiebler:

  ```
  dbr:Marc_Zwiebler = <http://dbpedia.org/resource/Marc_Zwiebler>
  ```

  Jede Zeile besteht aus einem RDF-Tripel <Subjekt,Prädikat,Objekt>. Das folgendes Beispiel zeigt ein Turtle-Dokument für den RDF-Graphen aus der Abbildung 3:

  ```
  @prefix: dbr: <http://dbpedia.org/resource/>
  @prefix: dbp: <http://dbpedia.org/property/>

  dbr:Marc_Zwiebler dbp:coach dbr:Xu_Yan_Wang.
  ```

  Literale werden in der Form dargestellt, die oben beschrieben wurde. Zum Beispiel könnte folgende Aussage „Marc Zwiebler wurde am 13.03.1984 geboren“ in Turtle-Syntax so aussehen:

  ```
  @prefix: dbr: <http://dbpedia.org/resource/>
  @prefix: dbp: <http://dbpedia.org/property/>

  dbr:Marc_Zwiebler dbp:dateOfBirth „1984-03-13“^^<http://www.w3.org/2001/XMLSchema#date>.
  ```

- *RDF/XML:* Ein RDF-Dokument in Form von RDF/XML ist ein wohlgeformtes XML-Dokument, das über zusätzliche Merkmale und Einschränkungen verfügt.[12] Diese Merkmale und Einschränkungen unterstützen die Erfassung, den Austausch und die Zusammenführung von Daten[12, S.29]. Daher wird XML-Serialisierung im Semantic Web verwendet. Ein RDFXML-Dokument besteht aus einer Liste von *nodeElements*. Jedes nodeElement enthält eine Liste von *propertyElements*. Es bedarf nur Ressourcen, Literale und Prädikate in RDF/XML darzustellen, um eine Syntax zu definieren. Der folgende RDF-Graph aus Abbildung 6 dient als Beispiel für die RDF/XML -Syntax.

Abbildung 6: Ein einfacher RDF-Graph zur Darstellung RDF/XML

Ressourcen : Eine Ressource wird in einem Description Element mit dem Attribut „about“ definiert. Eine Description packt alle Statements über ein Subjekt zusammen. Durch das *about*-Attribut wird eine Ressource mit der Angabe ihrer URI definiert. Zum Beispiel:

[12] http://www.w3.org/TR/rdf-syntax-grammer/

```
<rdf:Description rdf:about="http://dbpedia.org/resource/Marc_Zwiebler">
  <dbp:birthName>Marc Zwiebler</dbp:birthName>
</rdf:Description>
```

Prädikate: Ein Prädikat wird durch ein Prädikat-Element definiert. Ein Prädikat-Element erhält als Namen die URI des Prädikates der Aussage und das jeweilige Objekt der Aussage. Da das Objekt sowohl eine Ressource als auch ein Literal sein kann, ist es unterschiedlich darstellbar.

- *Objekt als Literal* : Das Literals wird als Inhalt der Prädikatelements verwendet. Zum Beispiel:

```
<rdf:Description rdf:about="http://dbpedia.org/resource/Marc_Zwiebler">
  <dbp:birthName xml:lang=„de“>Marc Zwiebler</dbp:birthName>
  <dbp:dateOfBirth rdf:datatype=„http://www.w3.org/2001/XMLSchema#date“>1984-03-13</dbp:
      dateOfBirth>
</rdf:Description>
```

  Es ist zu beachten, dass Datentypen und Sprachangaben als Attribute zum Prädikat-Element hinzugefügt sind.

- *Objekt als Ressource*: Hier wird die URI der Ressource als Wert des *rdf:resource*-Attributs des Prädikat-Elements angegeben. Zum Beispiel:

```
<rdf:Description rdf:about="http://dbpedia.org/resource/Marc_Zwiebler">
  <dbp:coach rdf:resource="http://dbpedia.org/resource/Xu_Yan_Wang"/>
</rdf:Description>
```

2.1.1.5 Blank Node

Ein Blank Node oder leerer Knoten ist eine Ressource ohne URI. Blank Node kann sowohl als eine Ressource als auch als ein Literal vorkommen. Er repräsentiert einen Platzhalter und besitzt keine URI. Blank Node wird dann nützlich, wenn einen lokalen Identifikator benötigt wird, weil ein Teilgraph sich nicht auf eine externe URI beziehen kann. Im Grunde genommen, verschafft der Blank Node eine virtuelle URI, um eine bestimmte Struktur zu realisieren. Es werden lokale IDs definiert, die als URIs für den Blank Nodes dienen. Im Graphen ist Blank Node ein eindeutiger Knoten, der weder eine URI hat noch ein Literal ist und verknüpft Teile eines RDF-Graphen. RDF Node wird bei Reifizierung, Container und komplexe Datentypen (z.B. Rezept eines Kuchens, das aus mehrere Komponenten besteht) verwendet.

Um es zu veranschaulichen, stellen wir das folgende Beispiel vor: Man möchte eine Liste deutscher Badmintonspieler „Die Liste Deutsche Badmintonspieler“ aufstellen. Das Ergebnis könnte in Turtle-Notation folgendermaßen aussehen:

```
@Präfix: category: <http://dbpedia.org/resource/Category>.
@Präfix: dct: <http://purl.org/dc/terms/subject>.
@Präfix: rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@Präfix: dbr: <http://dbpedia.org/resource/>.

category:German_Badmintons_Players dct:subject _:x.
_:x rdf:type rdf:Bag.
_:x rdf:_1 dbr:Marc_Zwiebler.
_:x rdf:_2 dbr:Hans_Riegel.
_:x rdf:_3 dbr:Ingo_Kindervater.
_:x rdf:_4 dbr:Juliane_Schenk.
```

Abbildung 7: Beispielgraph für einen Bag-Container mit Blank Node

Die Abbildung 7 verdeutlicht die Rolle eines Blank Nodes in einem RDF-Graph für das obengenanntes Beispiel. Der Blank Node hält in diesem Fall die Elemente der Bag zusammen. Man gibt dem Blank Node einen Namen (in unserem Beispiel „x“), dadurch ist es gekennzeichnet, dass die folgenden Elemente zu derselben Bag gehören. Der Name ist nicht Teil des RDF-Graphen. Wie man in Abbildung 7 sehen kann, erscheint der Name dort nicht.

2.1.1.6 RDF-Schema:

Mit RDF ist es möglich, Aussagen zu formulieren. Was weiterhin fehlt, ist die Semantik– die Bedeutung von Begriffen. Wenn mehr Wissen benötigt wird, kann RDF-Schema eingesetzt werden. Anhand RDF-Schema sind wir in der Lage, ein Vokabular zu definieren, das zur Beschreibung von Klassenbeziehungen und Eigenschaften für Ressourcen dient, eine Art einfache Ontologie-Sprache. Ein Schema wird Instanz genannt und ist selbst in RDF geschrieben und ist ein gültiges RDF-Dokument. RDF-Schema ist von W3C (2004) standardisiert. Das Kernvokabular von RDF-Schema ist unter IRI: *http://www.w3.org/2000/01/rdfschema#* mit Präfix *rdfs* festgelegt. Die wichtiges Elemente von RDF-Schema sind: *Klassen,Properties und Hierarchien/Beziehungen untereinander*.

- *Klassen*: RDF-Schema unterteilt alle Ressourcen in Klassen. Die Wurzelklasse für alle anderen Klassen ist *rdfs:Resource*. Klassen werden durch *rdfs:Class* beschrieben. Mit *rdf:typ* kann man genau formulieren, dass eine Klasse Element der Klasse *rdf:Class* ist. Zum Beispiel: *dbo:Person rdf:type rdfs:Class.*

  – *rdfs:Resource*: alle Ressourcen sind davon abgeleitet und damit Elemente der Menge Ressource. Die Ressource *rdfs:resource* ist Element der Menge *rdfs:Class*. Zum Beispiel:

  dbo:Person rdf:type rdfs:Resource .

  – *rdf:Property*: ist die Klasse aller Properties. Die Elemente dieser Klasse stellen die Beziehungen dar.

  – *rdfs:Literal*: für Datentypen wird *rdfs:Datatype* verwendet. Einfache Literale, wie Integer, String fallen in die Klasse *rdfs:Literal*. Zum Beispiel:

  "Marc Zwiebler" rdf:type rdfs:Literal .

- *Unterklassen und Klassenhierarchien*

  Mit *rdfs:subClassOf* können wir Beziehungen und Hierarchien zwischen Klassen definieren. Eine Klasse A kann in subClassOf-Relation zu einer Klasse B stehen. Eine solche Relation bildet eine transitive Beziehung zwischen Mengen. RDF-Schema verlangt nicht, dass Klassen eine strikte Hierarchie definieren. Eine Klasse kann verschiedene Oberklassen haben. Zum Beispiel:

  dbo:BadmintonPlayer rdfs:subClassOf dbo:Athlete .
  dbo:Athlete rdfs:subClassOf dbo:Person .

- *Properties*

  Alle Prädikate sind Elemente der Klasse Property. Solche Elemente sind an der Prädikat-Position in einem Statement. Zum Beispiel:

  dbp:coach rdf:type rdf:Property .

- *Unterproperties und Hierarchien auf Properties*

  Analog zu vorher dargestellte Beziehungen zwischen Klassen und Unterklassen ist es in RDF-Schema vorgesehen, die Beziehungen zwischen Properties ebenso zum Ausdruck zu bringen. Zum Beispiel:

  dbp:coach rdfs:subPropertyOf dul:coparticipatesWith .
  dul:coparticipatesWith rdfs:subPropertyOf dul:associatedWith .

- *Einschränkungen von Properties*: Es gibt zwei Arten von Einschränkungen für Properties.

  – *rdfs:range* hierdurch werden Einschränkungen des Wertebereichs für eine Property definiert. Zum Beispiel könnte der Wertebereich von *dbp:coach* auf den Typ *dbo:Person* eingeschränkt werden.

  ```
  dbp:coach rdfs:range dbo:Person.
  ```

  – *rdfs:domain* die Definitionsbereich wird eingeschränkt. Es wird bestimmt auf welche Klassen eine Property angewendet werden darf. Beispielweise könnte festgelegt werden, dass die Property *dbp:coach* nur auf Ressourcen der Klasse *dbo:Athlete* durchgeführt werden darf.

  ```
  dbp:coach rdfs:domain dbo:Athlete.
  ```

Zur Veranschaulichung der Nutzung des RDF-Schemas stellt die Abbildung 8 eine kleine Ontologie für die Aussage „Marc Zwiebler hat den Coach Xu Yang Wang“ dar. In TurtleNotation könnte das Beispiel aus der Abbildung 8 so formuliert sein:

```
@Präfix: rdf : <http://www.w3.org/1999/02/22-rdfsyntax-ns#>.
@Präfix: rdfs:  <http://www.w3.org/2000/01/rdf-schema#>.
@Präfix: dbr: <http://dbpedia.org/resource/>.
@Präfix: dbp: <http://dbpedia.org/property/>.
@Präfix: dbo: <http://dbpedia.org/ontology/>.
@Präfix: dul : <http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#>.

dbr:Marc_Zwiebler dbp:coach dbo:Xu_Yan_Wang.
dbr:Marc_Zwiebler rdf:type dbo:BadmintonPlayer.
dbo:BadmintonPlayer rdfs:subClassOf dbo:Person.
dbr:Xu_Yan_Wang rdf:type dbo:SportsManager.
dbo:SportsManager rdfs:subClassOf dbo:Person.
dbp:coach rdfs:subPropertyOf dul:coparticipatesWith.
dbp:coach rdfs:rang dbo:BadmintonPlayer.
dbp:coach rdfs:domain dbo:SportsManager.
```

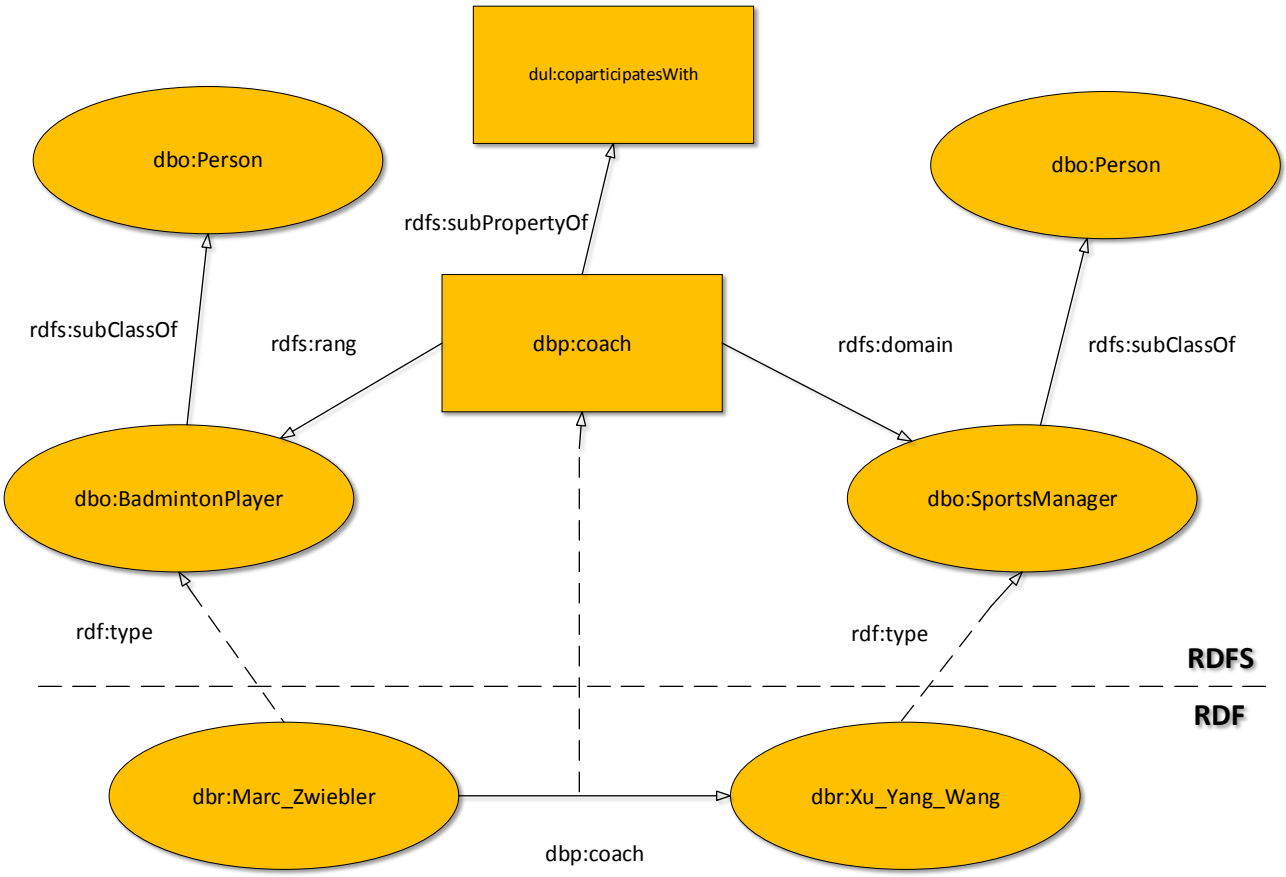


Abbildung 8: Abgrenzung zwischen RDF und RDFS (Eigene Darstellung, in Anlehnung an [13, S.57])

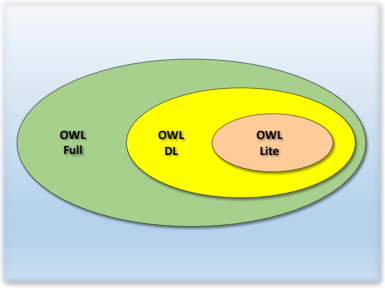


Abbildung 9: Sprachebenen der Web Ontology Language (Eigene Darstellung, in Anlehnung an [15, S.57])

2.1.2 OWL

OWL ist die Abkürzung von *Web Ontology Language*, ist wie RDF ein webbasierte Ontologiesprache. Sie wurde vom W3C (2004) als ein Standard anerkannt. Da OWL im Gegensatz zu RDF-Schema sehr ausdrucksstark ist, können in OWL komplexe prädikatenlogische Aussagen getroffen werden. OWL basiert auf der Syntax von RDF und der Prädikatenlogik erster Stufe. In OWL besteht eine Wissensbasis aus einer Menge von OWL-Aussagen (sog. A-Box) und Schlussfolgerungen (T-Box). Die T-Box (Terminologisches Wissen) enthält Klassen, Attribute und Eigenschaften. Beispiel: Die Klasse *BadmintonPlayer:{x| BadmintonPlayer(x)}* oder die Property *coach:{(X,Y) | coach(X,Y)}*. Die A-Box (Assertionales Wissen) enthält Instanzen. Beispiel: *BadmintonPlayer(Marc_Zwiebler) coach(Xu_Yan_Wan,Marc_Zwiebler)*[14, S.45].

Grundbausteine von OWL sind Klassen. Es gibt zwei besondere Klassen: *owl:Thing* und *owl:Nothing*, wobei owl:Nothing leer und owl:Thing die Oberklasse aller Klassen ist. Zum Beispiel:

```
dbo:Person  rdfs:subClassOf  owl:Thing.
owl:Nothing  rdfs:subClassOf  dbo:Person.
```

In OWL werden nicht nur Klassen definiert, sondern Properties und Individuen(einzelne Instanzen der Klassen). Die Properties und Klassenhierarchien sind in OWL komplexer und ausdrucksstärker definiert als in RDFS. Beispiel: Das Property „anchestor“ kann als ein transitives Property definiert werden und das Property „play“ als symmetrisches Property. Das Property „birthDate“ kann als funktionales Property definiert werden. Das folgende Beispiel zeigt die Definition der oben genannten Properties:

```
:ancestor    rdf:type  owl:TransitiveProperty.
:play        rdf:type  owl:SymmetricProperty.
:birthDate   rdf:type  owl:FunctionalProperty.
```

Da sich die Anforderungen an Ontologien stark unterscheiden, wurde OWL in drei Variante entworfen [15, S.58ff.]. Die Abbildung 9 stellt das Verhältnis aller Varianten zueinander dar.

- *OWL Lite* : diese Variante ist eingeschränkt und bietet Klassenhierarchie und Restriktionen an. Restriktionen für Properties sind nur mit bestimmten Kardinalitäten (nur mit den Werten 0 oder 1) erlaubt. Es handelt sich um eine eingeschränkte Variante, die sich hauptsächlich für einfache Konzept-Hierarchien mit wenigen einschränkenden Randbedingungen eignet. Hier können Klassen nur im Bezug auf Oberklassen definiert werden.

- *OWL DL (description logic)*: sie bietet den besten Mittelweg. OWL DL ist am weitesten verbreitet und von Reasoner unterstützt. Sie unterstützt alle syntaktischen Möglichkeiten der Prädikatenlogik. Sie ist entscheidbar und vollständig. Es gibt darin die Unterscheidung zwischen Klassen und Properties. Restriktionen für Properties sind mit beliebigen Kardinalitäten vorhanden.

- *OWL Full* : Diese Variante bietet alle syntaktischen Möglichkeiten von RDF an, dadurch ist RDF voll kompatibel. Im OWL Full kann man alle Sprachkonstrukte von RDF verwenden. Allerdings ist sie wegen der hohen Komplexität nicht mehr entscheidbar.

Das folgende Beispiel deklariert, dass jeder Badmintonspieler nur von einem Coach trainiert werden darf.

```
<owl:Class rdf:ID="BadmintonPlayer">
 <rdfs:subClassOf>
  <owl:Restriction>
   <owl:onProperty rdf:resource="#coach"/>
    <owl:minCardinality>1</owl:minCardinality>
    <owl:maxCardinality>1</owl:maxCardinality>
 </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>
```

Eine Detaillierte Beschreibung zu OWL und seine Syntax kannist online verfügbar[13].

2.2 SPARQL

Die Abkürzung *SPRQL:* steht für *Protocol And RDF Query Language*[14]. Sie definiert eine Abfragesprache und ein Abfrageprotokoll für RDF-Dokumente und wurde 2008 als offizielle Empfehlung des W3C freigegeben. SPARQL verwendet eine zu SQL ähnliche Syntax, die das Extrahieren und Modifizieren von Daten aus bestehenden RDF-Graphen ermöglicht. Außerdem ist es möglich, Sortierkriterien, die Filterung der Ergebnismenge anhand konkreten Regeln, aber auch die Strukturierung der Ergebnisse sowohl in tabellarischer Form als auch in Form eines RDF-Graphen einzubauen. Weiterhin gilt, dass die Graphpattern verwendet werden, um Ergebnisse zu erzielen. Zur Darstellung von Graphpattern dient die Turtle-Syntax.

2.2.1 Struktur einer Abfrage

Eine einfache SPARQL-Abfrage besteht aus mehreren Abschnitten: Deklaration von Präfixen, Abfrageergebnis und Graphpattern. Die Abbildung 10 stellt ein Beispiel für eine SPARQL-Abfrage dar und dient als Musterbeispiel zur Veranschaulichung alle Teile eines SPARQL-Abfrage in diesem Abschnitt:

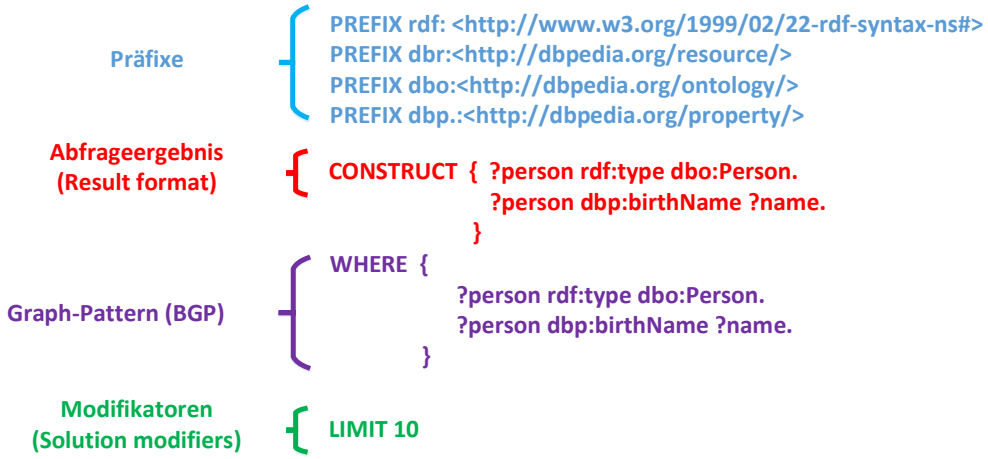


Abbildung 10: Beispiel einer SPARQL-Abfrage

[13] http://www.w3.org/TR/owl-ref/
[14] http://www.w3.org/TR/rdf-sparql-query/

Im Folgenden werden die Bestandteile jeder SPARQL-Abfrage beschrieben:

- *Präfixen:* Alle Ressourcen, Prädikate und Objekts sind mit URIs versehen. Um die Tripel, die im Graphpattern vorkommen, übersichtlicher darzustellen, werden Präfixe definiert. Ein Präfix wird mit dem Schlüsselwort „PREFIX“ ohne abschließenden Punkt definiert. Beispielweise würde das erste Tripel aus dem Graphpattern in der Abbildung 10 ohne Deklaration von Präfixen folgendermaßen aussehen:

  ```
  ?person <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/Person>
  ```

  Ohne Präfixen sehen die Tripeln sehr lang und unübersichtlich aus.

- *Data Set:* In SPARQL ist es möglich, Wissensbasen in mehrere Teildokumente zu unterteilen. Diese Teildokumente repräsentieren RDF-Graphen, die mit URIs versehen sind. Die URIs dienen als Name solcher RDF-Graphen. Der Name kann zum Beispiel der Pfad der RDF-Datei sein, in der der Graph gespeichert ist, oder jeder beliebige andere URI. SPARQL bietet die Möglichkeit, mehrere RDF-Graphen als Wissensbasis für eine Abfrage zu nutzen. Mit dem Schlüsselwort *FROM*, gefolgt von der URI des RDF-Graphen wird ein solcher Graph für die Abfrage definiert.

- *Result format:* Hier werden die Angaben bezüglich Formatierung der Ergebnisse gemacht. Das *SELECT* Schlüsselwort definiert eine Projektion. Das Schlüsselwort *CONSTRUCT* hingegen bildet einen RDF-Graph aus allen Tripeln, die im Result Format definiert sind und stellt dies als Ergebnis der Abfrage dar. *SELECT* und *CONSTRUCT* werden später detailliert in Abschnitt 2.2.5 erklärt. Die Abbildung 12 stellt das Ergebnis der Abfrage aus der Abbildung 11 als einen RDF-Graphen dar.

- *Graphpattern* (Basic Graph Pattern): Das Graphpattern beschreibt mittels Tripeln, welcher Teil des RDF-Graphen ausgewählt werden soll. Ein Graphpattern passt auf einen Teilgraphen, wenn die Ausdrücke aus diesem Teilgraphen durch die Variablen ersetzt werden können und die Lösung ist ein solcher Teilgraph (vgl. Abbildung 11). Jedes Tripel in einem Graphpattern ist in Turtle-Syntax (vgl. Abschnitt 2.1.1.4) angegeben. Als Beispiel zeigt Abbildung 11 eine SPARQL-Abfrage und die zugehörige Ergebnismenge zu der angegebenen Wissensbasis.

- *Solution modifiers*: Anhand von Solution Modifiers wie *LIMIT, OFFSET* oder *ORDERED BY* kann die Ergebnisliste beschränkt oder auch sortiert werden. Zum Beispiel beschränkt *LIMIT 10* die Größe der Ergebnisliste auf 10 Elemente.

2.2.2 Einfache Graphpattern

Es besteht ein Unterschied zwischen Graphpattern und RDF-Graph: *Subjekte, Prädikate und Objekte* können im Graphpattern durch Variablen ersetzt werden. Die Variablen spielen eine große Rolle bei der SPARQL-Abfrage beziehungsweise bei der Formulierung von Graphpattern. In diesem Abschnitt werden Variablen und im Anschluss daran Blank Node in SPARQL erklärt.

- *Variablen:* Eine Variable beginnt immer mit dem Zeichen *„?“* oder *„$“* und anschließend kommt eine Folge von alphanumerischen Zeichen, zum Beispiel „?person“. Eine Variable kann als Subjekt, Objekt oder Prädikat eines Triples vorkommen. Eine Variable kann mehrmals in einer SPARQL-Abfrage vorkommen. In jedem Tripel bezieht sie sich dann auf den gleichen Wert. Zum Beispiel kommt die Variable *?person* in folgenden Tripeln aus dem Graphpattern von der Abbildung 11 zwei mal als Subjekt vor:

  ```
  ?person rdf:type dbo:Person .
  ?person dbp:birthName ?name .
  ```

- *Blank Nodes:* Wie in Abschnitt 2.1.1.5 beschrieben, besitzt ein Blank Node keine URIs, sondern nur generierte IDs als Bezeichner und dient als Platzhalter. In SPARQL können sie als Subjekt oder Objekt vorkommen. Sie können wie Variablen auch bei der Durchführung einer Abfrage mit Werten des RDF-Graphen belegt werden. Durch vorher bestimmte ID kann ein Blank Node innerhalb eines Graphpattern sehr oft verwendet werden. In einem Graphpattern darf nicht gleiche ID mehrfach pro Abfrage eingesetzt werden. Weiterhin gilt, dass ein Blank Node sich innerhalb eines Graphpattern wie ein Variable eingesetzt werden. Ein Blank Node kann im Ergebnis einer Abfrage vorkommen, wenn der RDF-Graph auch Blank Node besitzt.

Nachdem nun alle Bestandteile einer SPARQL-Abfrage vorgestellt sind, wird hier eine einfache Graphpattern anhand des Beispiels aus der Abbildung 11 beschrieben. Das Graphpattern sieht wie folgt aus:

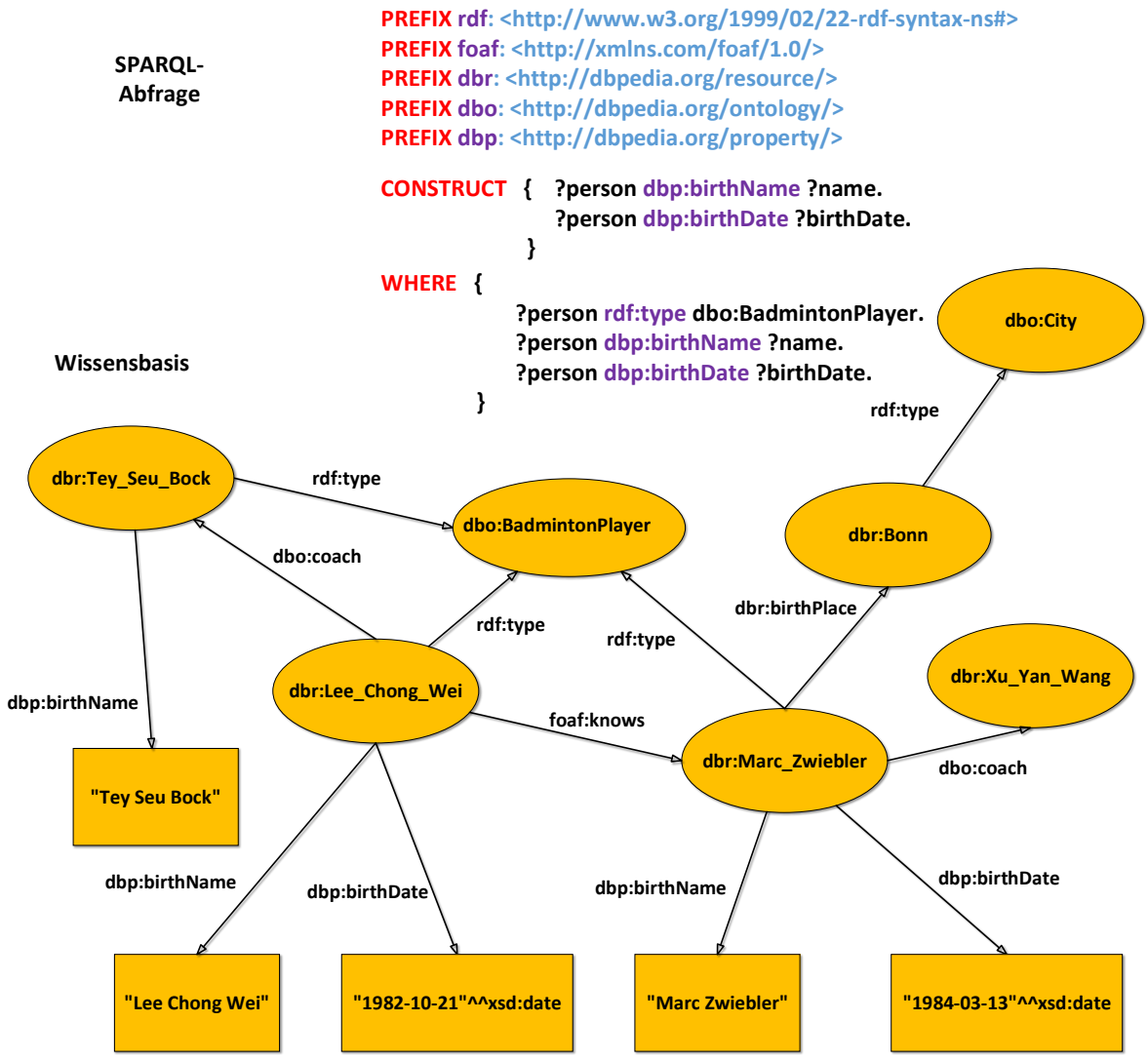


Abbildung 11: Beispiel einer SPARQL-Abfrage und eine Wissensbasis

SPARQL-Abfrage

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/1.0/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

CONSTRUCT { ?person dbp:birthName ?name.
            ?person dbp:birthDate ?birthDate.
          }
WHERE     {
            ?person rdf:type dbo:BadmintonPlayer.
            ?person dbp:birthName ?name.
            ?person dbp:birthDate ?birthDate.
          }
```

Ergebnis :

```
dbr:Marc_Zwiebler dbp:birthName "Marc Zwiebler"@de.
dbr:Marc_Zwiebler dbp:birthDate "1984-03-13"^^xsd:date.
dbr:Lee_Chong_Wei dbp:birthName "Lee Chong Wei"@en.
dbr:Lee_Chong_Wei dbp:birthDate "1982-10-12"^^xsd:date.
```

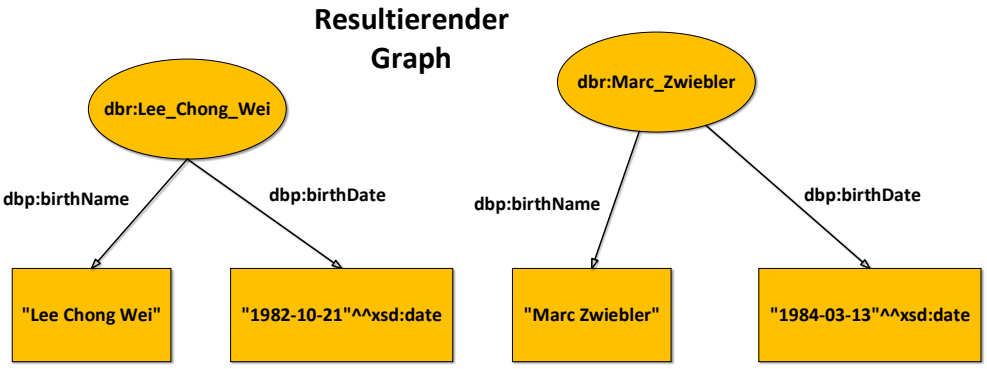


Abbildung 12: Das Resultat der Abfrage aus der Abbildung 11 als ein RDF-Graph

```
WHERE { ?person rdf:type dbo:BadmintonPlayer.
        ?person dbp:birthName ?name.
        ?person dbp:birthDate ?birthDate.
      }
```

2.2.3 Abfrage mit Datentypen

In SPARQL ist es möglich, Anfragen mit Datenwerten zu stellen. Subjekt, Prädikat und Objekt eines Tripels können durch Konstanten oder Variablen in einem Graphpattern definiert werden. Es gibt typisierte und untypisierte Literale, die in Abschnitt 2.1.1.1 beschrieben sind. Wenn ein Literal in untypisierter Form vorliegt, so kann mittels einer SPARQL-Abfrage ohne Angabe des Typs nach ihm gesucht werden. Die Literale mit Typisierung werden dann nicht mit in die Ergebnismenge aufgenommen. Zum Beispiel wird hier nach einem Badmintonspieler mit dem Namen *"Marc Zwiebler"* gesucht:

```
WHERE { ?person rdf:type dbo:BadmintonPlayer.
        ?person dbp:birthName "Marc Zwiebler".
      }
```

Die untypisierte Literale mit der Sprachangabe oder ohne werden unterschiedlich behandelt. Wenn die Sprachangabe vorhanden ist, sollte sie genau berücksichtigt werden, wie hier:

```
?person dbp:birthName "Marc Zwiebler"@de.
```

Besitzt ein Literal einen Datentyp, sollte der gesuchten Typ in der Abfrage berücksichtigt und angegeben werden. Folgendes Graphpattern gibt alle Badmintonspieler aus, die genau 10 Titel gewonnen haben:

```
WHERE { ?person rdf:type dbo:BadmintonPlayer.
        ?person dbp:titles "10"^^<http://www.w3.org/2001/XMLSchema#Integer>.
      }
```

2.2.4 Komplexe Graphpattern

Nachdem die grundlegende Idee und Funktionsweise der SPARQL-Sprache erläutert wurde, soll hier ein Überblick über ihre wichtigsten Bestandteile und Ausdrucksmittel gegeben werden. Es ist möglich aus mehreren einfachen Graphpattern komplexere Graphpattern zu erzeugen. Dadurch ist es möglich, bestimmte Bedingungen an einen Teil des gesamten Graphpattern zu stellen, um einen neuen Teilgraph zu definieren. Zum Beispiel können Teilgraphen aus optionalen oder mehrere alternativen Pattern definiert und mit einander kombiniert werden. Dies geschieht, indem man Teilgraphen mit geschweiften Klammern voneinander trennt. Die Variablen werden dabei an die entsprechenden Werte des Datentripels gebunden. Das folgende Beispiel zeigt ein Graphpattern, der aus drei Teilgraphen besteht. In diesem Beispiel hat die Gruppierung keinen Effekt auf das Ergebnis, weil keine zusätzliche Ausdrucksmittel verwendet wurden.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT ?person ?name ?birthDate
WHERE { {?person rdf:type dbo:BadmintonPlayer.
         ?person dbp:birthName ?name.}
         ?person dbp:birthDate ?birthDate.
       }
```

2.2.4.1 OPTIONAL

Mit dem Schlüsselwort *„OPTIONAL“* lassen sich Pattern definieren, welche mit anderen Pattern kombiniert und als ganzes einen komplexen Graphpattern darstellen werden können. Solche Pattern müssen nicht unbedingt in der Wissensbasis vorhanden sein. Das optionale Graphpattern wird benutzt, wenn die Daten nicht vollständig oder eventuell teilweise in der Wissensbasis vorhanden sind. Hierfür werden alle Tripeln in einer geschweiften Klammer mit dem vorangestellten Schlüsselwort *OPTIONAL* stellt. In Abbildung 11 ist deutlich zu erkennen, dass die Personen *dbr:Tey_Seu_Bock* und *dbr:Xu_Yang_Wang* nicht in der Ergebnisliste vorgekommen sind. Die Person *dbr:Tey_Seu_Bock* besitzt keine Property *dbp:birthDate* und die Person *dbr:Tey_Seu_Bock* besitzt weder eine Property *dbp:birthName*, noch eine Property *dbp:birthDate*. Um nun aber auch alle Badmintonspieler, die kein Geburtsdatum besitzen, im Graphen abzudecken, muss das entsprechende Pattern als *optional* deklariert werden. Zum Beispiel:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dbo: <http://dbpedia.org/ontology/>
CONSTRUCT{ ?person rdf:type dbo:BadmintonPlayer.
           ?person dbp:birthName ?name.
           ?person dbp:birthDate ?birthDate.
         }
WHERE    { ?person rdf:type dbo:BadmintonPlayer.
           ?person dbp:birthName ?name.
           OPTIONAL{?person dbp:birthDate ?birthDate.}}
```

Resultierender Graph:

```
dbr:Marc_Zwiebler rdf:type dbo:BadmintonPlayer.
dbr:Marc_Zwiebler dbp:birthName "Marc Zwiebler"@de.
dbr:Marc_Zwiebler dbp:birthDate "1984–03–13"^^xsd:date.
```

```
dbr:Lee_Chong_Wei rdf:type dbo:BadmintonPlayer.
dbr:Lee_Chong_Wei dbp:birthName "Lee Chong Wei"@en.
dbr:Lee_Chong_Wie dbp:birthDate "1982-10-12"^^xsd:date.
dbr:Tey_Seu_Bock rdf:type dbo:BadmintonPlayer.
dbr:Tey_Seu_Bock dbp:birthName "Tey Seu Bock"@en.
```

2.2.4.2 FILTER

FILTER wird dort eingesetzt, wo die Abfragen nicht mehr mit komplexen Graphpattern möglich sind. Zum Beispiel

- Alle Badmintonspielern, die jünger als 25 Jahre alt sind
- Alle Deutsche Badmintonspielern
- Alle Athleten, die zwischen 70 kg bis 80 kg wiegen

Mit dem Schlüsselwort *FILTER* kann man konkrete Regeln definieren und die Ergebnismenge einschränken. Damit kann man nach Werten in einem bestimmten Bereich suchen oder literalen nach bestimmtem Muster aussortieren oder nach bestimmten Text mit passender Sprachangabe suchen. Eine komplette Auflistung sämtlicher Filter ist online[15] zu finden.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

CONSTRUCT { ?person dbp:birthName ?name.
            ?person dbp:birthDate ?birthDate.
          }
WHERE   { ?person rdf:type dbo:BadmintonPlayer.
          ?person dbp:birthName ?name.
          ?person dbp:birthPlace ?country.
          ?country rdf:type dbo:Country.
          FILTER( ?country = dbr:Germany )
        }
```

Diese Abfrage ermittelt einen RDF-Graph aus allen deutschen Badmintonspielern mit ihrem Geburtsdatum und ihrem Namen. Ein FILTER-Pattern wird durch das Schlüsselwort *FILTER* gefolgt von einem Filterausdruck gebildet. Der Filterausdruck besteht aus Parametern und Vergleichsoperatoren, Booleschen und Arithmetische Operatoren und liefert für jedes Resultat als Wert entweder *TRUE* oder *FALSE*. Für numerische Datentypen wie *xsd:dateTime, xsd:string oder xsd:Boolean* sind Vergleichsoperatoren und für andere Typen und sonstige RDF-Elemente sind nur = und *!*= verfügbar. Im Folgenden Beispiel wird nach Werte in einem bestimmten Bereich gesucht:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

CONSTRUCT { ?person dbp:birthName ?name.
            ?person dbp:weight ?weight.
          }
WHERE { ?person rdf:type dbo:Athlete.
        ?person dbp:birthName ?name.
        ?person dbp:weight ?weight.
         FILTER( (?weight =< 80) && (?weight => 70) )
      }
```

[15] http://www.w3.org/TR/rdfsparqlquery/#SparqlOps

Diese Abfrage findet alle Athleten, die zwischen 70 kg bis 80 kg wiegen. Die Filterfunktion dieser Abfrage besteht aus zwei Ausdrücken: *(?weight <= 80)* und *(?weight >= 70)*, die jeweils aus einem Parameter und einem Vergleichsoperator bestehen. Die beide Ausdrücke werden dann mit einem Booleschen Operator && in logische Verbindung gesetzt.

2.2.4.3 UNION

Das Schlüsselwort UNION erlaubt die Angabe alternativer Teile eines Musters. Mit UNION werden zwei Teilpattern zu einem Graphpattern verknüpft. Der UNION kann als Logisches *Oder* verstanden werden. Das Ergebnis entspricht der Vereinigung der Ergebnisse für die beiden Bedingungen. Gleiche Variablennamen in beiden Teile von UNION beeinflussen sich nicht. Das folgende Beispiel zeigt eine Abfrage nach allen sowohl weiblichen als auch männlichen Athleten.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

CONSTRUCT { ?person rdf:type dbo:Athlete.
            ?person dbp:birthName ?name.
          }
WHERE   { ?person rdf:type dbo:Athlete.
          ?person dbp:birthName ?name.
          { ?person dbo:gender <http://dbpedia.org/resource/Female>.} UNION
          { ?person dbo:gender <http://dbpedia.org/resource/Male>.}
        }
```

2.2.5 Abfrageformate

In allen bis jetzt gestellte Abfragen in dieser Arbeit wurde als Result Form (vgl. Abschnitt 2.2.1) immer *CONSTRUCT* verwendet. SPARQL bietet weitere Abfrageformate wie *SELECT, ASK* und *DESCRIBE*. In diesem Abschnitt werden alle Abfrageformate vorgestellt.

2.2.5.1 SELECT

Das Schlüsselwort *SELECT* definiert eine Projektion. Hier werden die Variablen aus dem Graphpattern ausgewählt, die im Ergebnis der Abfrage benutzt werden. Bei SELECT-Abfragen ist das Ergebnis eine Menge von Bindungen, die sich besonders für eine sequentielle Verarbeitung eignen. Das folgende Beispiel zeigt eine Abfrage mit den zugehörigen Ergebnissen:

In Abbildung 13 wird eine SELECT-Abfrage mit zwei Variablen *?name* und *?birthDate* definiert. Die Abfrage sucht in der gegebenen Wissensbasis nach *dbp:birthName* und *dbp:birthDate* unter allen Badmintonspielern. Das Ergebnis dieser Abfrage ist eine Tabelle. Man muss in Betracht ziehen, dass das Pattern *?person dbp:birthDate ?birthDate.* nicht auf alle Ressourcen im Graphen passt. *dbr:Teu_Seu_Bock* besitzt kein Geburtsdatum und ist daher nicht im Ergebnis dieser Abfrage.

2.2.5.2 CONSTRUCT

Mittels CONSTRUCT ist die Weiterverarbeitung von Daten möglich. SPARQL bietet die Möglichkeit aus den Lösungen des Graphpattern wiederum einen RDF-Graph zu rekonstruieren. Die CONSTRUCT-Klausel dient als eine Schablone für den neuen RDF-Graphen und die Syntax entspricht des Graphen-Patterns. Infolge dessen sind Ergebnisse strukturiert und es gibt Beziehungen zwischen Elementen. Die Abfrage aus der Abbildung 11 findet einen neuen RDF-Graphen, der für jeden Badmintonspieler die Tripel *?person dbp:birthName ?name.* und *?person dbp:birthDaten ?birthDate.* enthält.

SPARQL unterstützt zwei weitere Ausgabeformate:

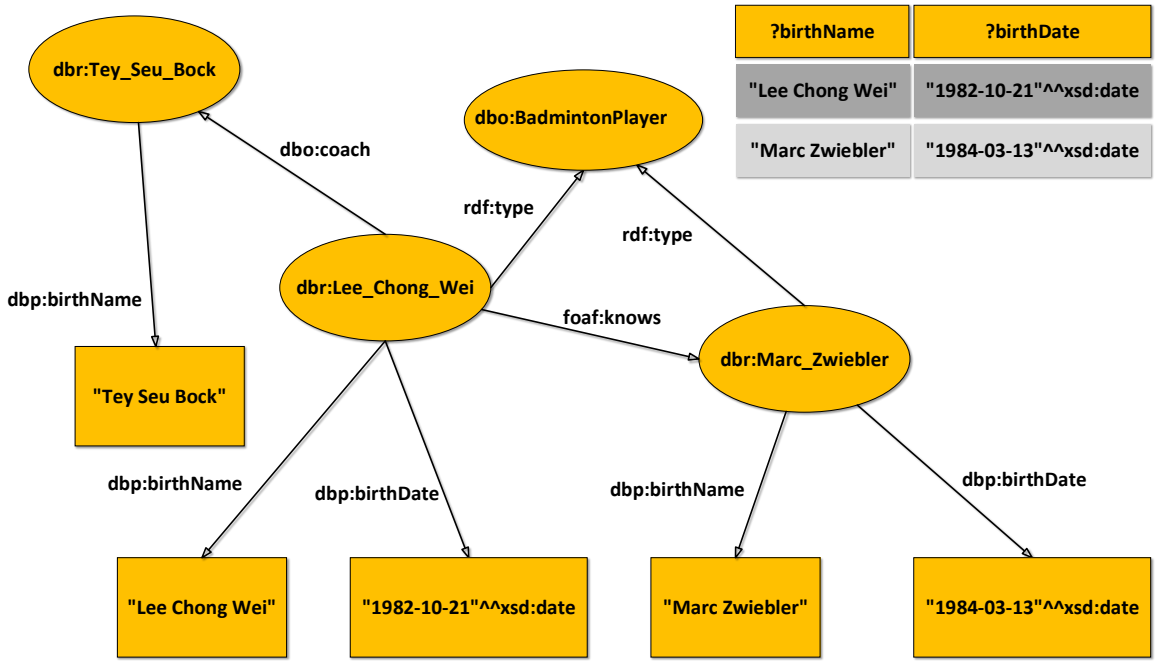
SPARQL-Abfrage

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/1.0/>
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

SELECT ?birthName ?birthDate

WHERE {
        ?person rdf:type dbo:BadmintonPlayer.
        ?person dbp:birthName ?birthName.
        ?person dbp:birthDate ?birthDate.
      }
```

Wissensbasis



Ergebnis

?birthName	?birthDate
"Lee Chong Wei"	"1982-10-21"^^xsd:date
"Marc Zwiebler"	"1984-03-13"^^xsd:date

Abbildung 13: Beispiel einer SELECT-Abfrage

2.2.5.3 ASK und DESCRIBE

ASK: prüft nur, ob es Ergebnisse gibt. Als Ergebnis liefert die Anfrage entweder *„TRUE“* oder *„FALSE“* zurück. *DESCRIBE*: liefert zu jeder gefundenen URI eine RDF-Beschreibung. Die alternativen Formaten werden in dieser Arbeit im Folgenden nicht betrachtet.

2.2.6 SPARQL Query Results XML Format

Das *SPARQL Query Results XML Format*[16] definiert, wie die Abfrageergebnisse eines Clients an einen SPARQL-Server auszusehen haben. Dabei bietet das *SPARQL Query Results XML Format* die Möglichkeit, Abfrage-Ergebnisse an die in der Abfrage festgelegten Variablen zu binden [16]. Je nach Art der Abfrage ist die Antwort unterschiedlich.

- *CONSTRUCT-, DESCRIBE-Abfrage*: Das Ergebnis von CONSTRUCT und DESCRIBE ist ein RDF-Graph und wird z.B.: als RDF/XML übertragen (Vgl. Abschnitt 2.1.1.4).

- *SELECT-Abfrage*: SPARQL definiert ein spezielles XML-Format für die Serialisierung von Ergebnissen einer SELECT-Abfrage.

Ein SPARQL Results-Dokument für SELECT-Abfrage ist wie folgt aufgebaut:
Ein *Results Dokument* beginnt mit einem *sparql-Element* und Angabe der URI *http://www.w3.org/2005/sparqlresults#*. Jedes *sparql-Element* besteht aus zwei Kinderelementen, *head-Element* und *results-Element*. *head-Element* ist das erste Kinderelement und enthält Elemente für alle abgefragten Variablen. Für jede Variable aus der SELECT-Abfrage wird ein leeres Element mit der Attributname *name* und dem Namen der Variablen als Wert definiert. Folgendes Beispiel zeigt ein *head-Element* für die Abfrage aus Abbildung 13:

```
<?xml version="1.0"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#>
  <head>
    <variable name="birthName"/>
    <variable name="birthDate"/>
  </head>
</sparql>
```

Das zweite Kinderelement ist entweder ein *results-Element* oder ein *boolean-Element*. Ein *boolean-Element* kommt in Falle eines *ASK*-Operators, welcher als Ergebnis nur Boolesche Werte zurückliefert. Ein *results-Element* enthält eine Liste von *result-Element*. Jedes *result-Element* stellt eine Zeile der Tabelle der Abfrage-Ergebnisses dar. Das *result-Element* besteht aus mehreren Kinderelementen, die *binding-Element* heißen. Die *binding-Elemente* sind in der Reihenfolge sortiert, in der die Variablen im *SELECT-Operator* sind. Jedes *binding-Element* ist ein Kinderelement des *result-Element* mit der Variablennamen für das Attribut *„name“*. Zum Beispiel sieht ein *results-Element* für zwei Variablen *?birthName* und *?birthDate* der Abfrage aus der Abbildung 13 wie folgt aus:

```
<results>
  <result>
    <binding name="birthName"/>...</binding>
    <binding name="birthDate"/>...</binding>
  </result>
</results>
```

Die Wert eines *binding-Elements* wird wie folgt definiert:

- *Ressource U mit URI*:
  `<binding><uri>U</uri></binding>`

- *Literal S*
  `<binding><literal>S</literal></binding>`

- *Literal S mit der Sprachangabe L*:

[16] http://www.w3.org/TR/2013/REC-rdf-sparql-XMLres-20130321/

<binding><literal xml:„L“>S</literal></binding>

- *Typisiertes Literal S mit Datentyp D*:
  <binding><literal datatype:„D“>S</literal></binding>

- *Blank Node mit label I*:
  <binding><bnode>I</bnode></binding>

Die Abbildung 14 zeigt die SPARQL-Antwort auf die SELECT-Abfrage aus Abbildung 13:

```
<?xml version="1.0" encoding="utf-8"?>
<sparql xmlns="http://www.w3.org/2005/sparql-results#">
 <head>
   <variable name="dbr:birthName"/>
   <variable name="dbr:birthDate"/>
 </head>
 <results>
  <result>
    <binding name="birthName">
      <literal xml:"de">Lee Chong Wei</literal>
    </binding>
    <binding name="birthDate">
      <literal datatype:"http://www.w3.org/2001/XMLSchema#date">1982-10-12</literal>
    </binding>
  </result>
  <result>
    <binding name="birthName">
      <literal xml:"de">Marc Zwiebler</literal>
    </binding>
    <binding name="birthDate">
      <literal datatype:"http://www.w3.org/2001/XMLSchema#date">1984-03-13</literal>
    </binding>
  </result>
 </results>
</sparql>
```

Abbildung 14: Beispiel einer SPARQL-Antwort

Abbildung 15 illustriert die Ergebnistabelle der oben gezeigten *SELECT-Antwort*:

Ergebnistabelle

?birthName	?birthDate
"Lee Chong Wei"	"1982-10-21"^^xsd:date
"Marc Zwiebler"	"1984-03-13"^^xsd:date

Abbildung 15: Ergebnistabelle einer SELECT-Abfrage

2.3 Linked (Open) Data

Linked (Open) Data stellt Daten frei zugänglich zur Verfügung. Dabei sind die Daten mittels RDF (*Resource Description Framework*)(vgl. Abschnitt 2.1.1) strukturiert und frei verfügbar. Die Daten werden als *RDF-Graph* repräsentiert. Dieser

RDF-Graph kann mittels der Abfragesprache *SPARQL* 2.2 abgefragt werden. Die Knoten und die Kanten des *RDF-Graph* sind mit *URI(Uniform Resource Identifiers)* versehen. *HTTP* dient als Kommunikationsprotokoll für *Linked Open Data*. Mittels der oben genannten Technologien stellt *Linked (Open) Data* Wissen aus verschiedensten Domänen in einer Form, die von Maschinen interpretiert werden kann, zur Verfügung. Dadurch ist es möglich, mit automatischen Methoden das Auffinden von Mustern.

Grundkonzept

Die Prinzipien von *Linked Open Data*, die vom *Tim Berners-Lee* zusammengefasst wurde, sind folgendermaßen:

- Nutze eindeutige Identifikatoren (*URIs*) als Name für Dinge.
- Nutze *HTTP-URIs* um diese Dinge im Web auffindbar zu machen.
- Nutze den *RDF-Standard* zum Strukturieren der *URIs*.
- Verbinde *URIs* mit anderen URIs um weitere Informationen auffindbar zu machen.

Linked Open Data Cloud (LOD-Cloud)

Die *Linked Open Data Cloud* verbindet alle *Datensets*(Datenquellen), die im *Linked Open Data Format* veröffentlicht wurden. Abbildung 16 visualisiert die Zusammenhänge zwischen den einzelnen *Datasets*. Die Größe der Kreise entspricht ungefähr der Anzahl der Triples in den jeweiligen *Datensets* und die Dicke der Kante repräsentiert, wie und wie viele RDF-Triples zwischen *Datenset* untereinander verbunden sind und im Zentrum aller *Datensets* seht die *DBpedia*.

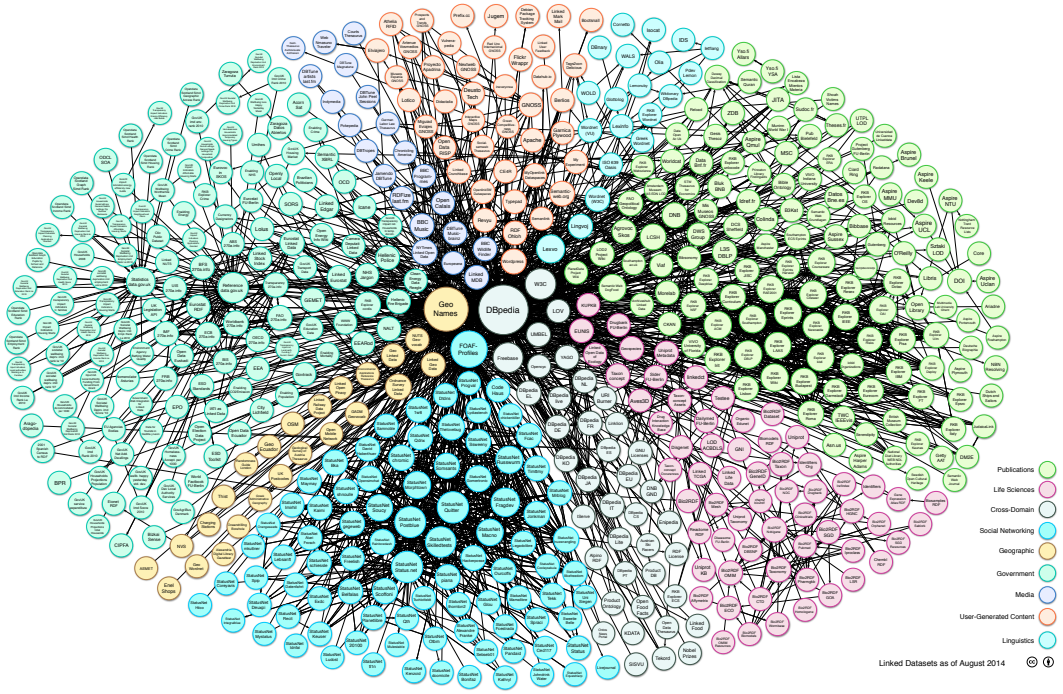


Abbildung 16: Linked Open Data Cloud [17]

Die *Linked Open Data Cloud* von August 2014 ist in Abbildung 16 visualisiert. Bereits im August 2014 beinhaltete die *LOD-Cloud* 1014 *Datensets* und seit Mai 2012 gibt es über Billionen Tripel in der *Linked Open Data Cloud* [18]. Google nutzt *Linked Open Data*, um seine Suchresultate zu verbessern.

[18] http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/

Um Links in Linked Open Data auszunutzen, existiert außerdem Operatoren, der Links zwischen Datensets folgt (insbesondere *owl:sameAs*), um in anderen Datensets weitere Informationen zu einem Objekt zu finden. Dadurch ist es möglich Hintergrundwissen aus verschiedenen Datensets in der *Linked Open Data Cloud* zu verbinden.

Eine der am häufigsten referenzierten Datensets in der *Linked Open Data Cloud* ist *DBpedia*. *DBpedia* [19] ist eine frei verfügbare *Linked (Open) Data* Datenset(*Dataset*) und stellt Wissen aus Wikipedia, der größten Web-Enzyklopädie dar. Das Wissen steht mit Hilfe von semantischen Technologien frei zur Verfügung. Die englische Version von *DBpedia* beschreibt 4.58 Millionen Instanzen und davon sind 4.22 Millionen als eine *Ontologie* klassifiziert. Mittlerweile gibt es *DBpedia* in 125 Sprachen[20]. *DBpedia* ist mit anderen Datensets über 50 Millionen RDF-Links verbunden. Zum Beispiel die Ressource *http://dbpedia.org/resource/Teheran* repräsentiert die Stadt *Teheran* im *DBpedia*.

2.3.1 Jena Framework

Das *Jena* Semantic Web Framework [21] ist eine Programmierumgebung zur Verarbeitung von RDF-Daten. Jena wird aktiv vom HP Labs Semantic Web Programme[22] seit 2000 weiterentwickelt und besitzt eine große Community. *JENA2* wurde im Jahr 2003 veröffentlicht und unterstützt RDF, RDF-Schema, OWL und SPARQL [17, S.74 ff.]. In Jena ist es möglich, Modelle aus RDF-Dokumenten zu erzeugen und sie in RDF-Dokumente zurück zu schreiben. In diesem Abschnitt werden die grundlegende Elemente, die zur Verarbeitung von SPARQL-Abfragen nötig sind, erklärt. Jena bietet unter anderem die Interfaces *Model*, *Resource*, *Property*, *Literal* und *Statements*, die die entsprechenden Einheiten von RDF in Jena repräsentieren, sowie Schnittstellen wie: Java RDF API, ARQ.

Java RDF API: In JENA repräsentiert die Klasse *„Model“* RDF-Graphen. RDF API ermöglicht Zugriffe auf die Daten im Modell. Dadurch können Modelle erstellt oder manipuliert werden. Weiterhin können Tripels zum Model hinzugefügt oder aus dem Model gelöscht werden. Ebenso können einfache Abfragen an die Modelle gestellt werden. Das *Model* ist ein Interface, das von Klassen implementiert wird, die die Daten im Arbeitsspeicher halten oder im Dateisystem speichern. Es ist möglich, ein Model sowohl mittels RDF API als auch aus einem RDF-Dokument zu erzeugen. Ein Model besteht aus einer Liste von *Statements*. Neben *Model* gibt es weitere Klassen und Datentypen in der RDF API:

- *Property:* Repräsentiert ein RDF(S)-Prädikat.
- *Resource:* Repräsentiert eine RDF(S)-Ressource, die einen URI hat.
- *RDFNode:* Repräsentiert alle RDF-Ressourcen, d.h. auch Literale, die keinen URI haben.
- *Statement:* Repräsentiert ein RDF-Tripel im Model.

Abbildung 17 stellt die Beziehungen zwischen diesen Schnittstellen im folgenden Klassendiagramme dar.

[19] http://dbpedia.org
[20] http://wiki.dbpedia.org/about
[21] http://jena.sourceforge.net/
[22] http://www.hpl.hp.com/semweb/

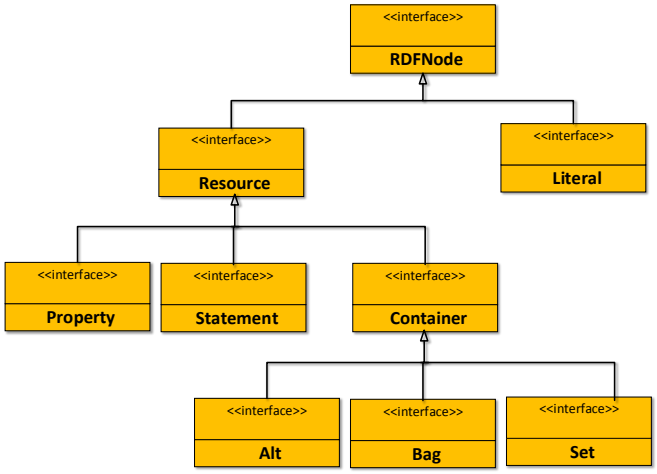


Abbildung 17: Klassendiagramm der Klassen für RDF API des Jena in Anlehnung an [23]

Das Interface *RDFNode* wird von den Klassen *Ressource* und *Literal* implementiert. Die beiden Klassen können als Objekt in einem Tripel auftreten. Das Interface *Literal* repräsentiert ein Literal im Modell. Ein Literal kann in Jena als String, Boolean, Integer, Character oder als Objekt vorkommen. Das Interface *Resource* repräsentiert Ressourcen von RDF-Dokumenten.

Folgender Quellcode demonstriert, wie das Statement „Lin Dan ist ein Badmintonspieler“ durch RDF API implementiert und in ein Modell hinzugefügt wird:

```
// Erzeugen ein leeres Modell
Model model = ModelFactory.createDefaultModel();
// Erzeugen die Ressource Lin Dan
Resource linDan = model.createResource("http://dbpedia.org/resource/Lin_Dan");
// Erzeugen Property rdf:type
Property type = model.createProperty("http://www.w3.org/1999/02/22-rdf-syntax-ns#type");
// Erzeugen die Ressource BadmintonPlayer
Resource badmintonPlayer = model.createResource("http://dbpedia.org/ontology/BadmintonPlayer");
// Einfügen oben genannten Statements ins Modell
 model. add(linDan, type, badmintonPlayer);
```

Der nächste Quellcode zeigt, wie ein Literal mit Datentyp in Jena erzeugt wird. Als Beispiel dient das Statement „Lin Dan wurde am 1983-10-14 geboren“.

```
// Erzeugen ein Literal mit Datentype
Literal datum = model.createTypedLiteral("1983-10-14","http://www.w3.org/2001/XMLSchema#date");
// Erzeugen Property \emph{birthDate}
Property birthDate = model.createProperty("http://dbpedia.org/property/birthDate");
// Einfügen ins Modell
 model. add(linDan, birthDate, datum);
```

In Jena gibt es zwei Klassen zur Repräsentation von RDF-Dokumenten: Die Klasse *Model* und die Klasse *OntoModel*. Die Klasse *Model* repräsentiert ein RDF(S)-Model und die Klasse *OntoModel* erweitert das Basisklasse Modell um Konzeption und Vokabular von OWL. Dadurch ist eine Verarbeitung von Ontologien in Jena möglich. Ausführliche Information und

eine detaillierte Beschreibung kann auf der Homepage[24] gelesen werden.

ARQ API: ist das Abfragesystem zu Jena. ARQ beinhaltet Datenstrukturen zur Repräsentation von Abfragen und Ergebnissen, einen SPARQL-Parser, und die Schnittstellenbeschreibung für eine Abfragebearbeitungskomponente und Möglichkeiten zum Umgang mit dem Ergebnissen einer Abfrage[17, S.47 ff.]. ARQ API bietet zwei Möglichkeiten SPARQL-Abfragen zu bearbeiten: Auf syntaktischer Ebene und auf algebraische Ebene. Zum Beispiel dienen die Klassen *Element* und *Query* zur syntaktischen Verarbeitung den Abfragen. Die abstrakte Klasse *Element* bietet eine Datenstruktur, die Abfragen in Jena repräsentiert. Die einzelne Klassen aus dieser Datenstruktur stellen Elemente von Abfragen dar. Zum Beispiel: *ElementGroup*, *ElementOption*, *ElementFilter* und *ElementUnion*.
Die Klasse *Query* bietet viele Methoden, um Daten aus einem RDF-Dokument oder mittels URI*s* ins Modell zu übertragen.[25]
Folgender Quellcode zeigt, wie eine Query mittels ARQ API erzeugt wird:

```
// Eine Query als Text
String queryText = "...";
// Erzeugen einer Query
Query query = QueryFactory.create(queryText);
```

Die Klasse *QueryExecution* bietet die Möglichkeit, eine Abfrage auszuführen und die Ergebnisse ins Modell zu speichern. Folgender Quellcode demonstriert es:

```
// Erzeugen einer QueryExcecution
QueryExecution qexec = QueryExecutionFactory.sparqlService(urlService,queryText);
// Ausführen einer Query und Übermitteln Ergebnisse ins Modell
Model model = qexec.execConstruct();
```

2.4 Maschinelles Lernen

In den folgenden Abschnitten werden Konzepte und Grundlagen des maschinellen Lernens vorgestellt. Basierend auf maschinellem Lernen werden Grundlagen sowie diverse Lernansätze besprochen. Es wird der Lernansatz von *SeCo-Algorithmen*, die in dieser Arbeit eingesetzt wird, vorgestellt. Weiterhin wird einen kurzen Überblick über die tatsächliche Realisierung von *SeCo-Algorithmen* im *SeCo-Framework* gegeben. Anschließend wird *Weka*, ein beim maschinellen Lernen häufig eingesetztes Instrument und das dafür verwendete *ARFF-Format* dargestellt.

2.4.1 Konzepte des maschinellen Lernen

Es gibt zahlreiche Definitionen für *„Maschinelles Lernen“*, wie zum Beispiel die folgenden Aussagen:

- *„Learning denotes changes in the system that are adaptive in the sense that they enable the system to do the same task or tasks drawn from the same population more efficiently and more effiectively the next time.“ [18]*.
  Diese Definition von *maschinellem Lernen* bezieht sich auf eine Veränderung eines Systems, um eine Aufgabe bei Wiederholung derselben Aufgabe besser zu lösen.

- *„A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its perfomance at tasks in T, as measured by P, imporves with experience.“ [19]*
  In dieser Definition sind sowohl Aspekte des Lernens, Aufgabenlösung und als auch Messbarkeit enthalten.

Maschinelles Lernen befasst sich mit Computerprogrammen, die aus Erfahrungen Muster erkennen. Der Lerner stellt während des Prozesses des Lernens eine Hypothese, die das erlernte Muster möglichst genau nachbildet. Eine Klasse ist eine Abbildung, die jedem Eingabewert einen Ausgabewert zuordnet. Zum Beispiel kann in einer Obstfarm beim Sortieren der Produkten anhand von Merkmalen wie *Farbe* und *Größe* die Produktklasse ermittelt werden.

Bei der Suche nach der Bedeutung des Lernens sollten verschiedene Sichten berücksichtigt werden. Lernen bedeutet für Menschen, sich durch Erfahrungen, Studium oder Lehre Wissen anzueignen. Im Gegenteil ist Lernen für Maschinen an

[24] https://jena.apache.org/documentation/ontology/
[25] https://jena.apache.org/documentation/query/app_api.html

Mustererkennen gebunden und nicht an das Erlernen von Wissen. Eine Maschine lernt also dann, wenn sie seine Leistung bei einer Aufgabe mit zunehmender Erfahrung verbessert [19, Seite 2 ff.] Die Aufgabe beim maschinellem Lernen besteht darin, aus den gesammelten klassifizierten Daten eine Funktion zu generieren, die aus Merkmale den gegeben Daten den Wert der Klasse berechnet.
Es gibt zwei grundsätzliche Ansätze, um Wissen aus Daten zu gewinnen beziehungsweise zu Lernen:

- *Überwachtes Lernen* Beim *überwachtem Lernen* wird versucht, aus einer Menge von Beispieldaten, die eine Klasseneinteilung enthalten, Zusammenhänge zu erkennen. Durch Zusammenhänge zwischen den Attributwerten und der Klassifizierung können neue noch unklassifizierte Instanzen einer Klasse zugeordnet werden. Zum Beispiel sind eine Menge von früheren taktischen Aufstellungen eines Fußballmannschaft bekannt, und nun möchte man vorhersagen, wie sich diese Fußballmannschaft beim nächsten Spiel taktisch aufstellt.

- *Unüberwachtes Lernen* Eine Menge von Beispieldaten liegen vor, wobei die Klassenzugehörigkeit der Trainingsbeispiele unbekannt ist. Es wird versucht, anhand der Attributwerte ähnliche Instanzen in Gruppen zusammenzufassen. Ein solches Vorgehen veranschaulicht das Verfahren des unüberwachten Lernens [20].(z.B. Clustering)

2.4.2 Grundbegriffe

In dieser Arbeit werden die Grundbegriffe aus maschinellem Lernen erklärt [21].

Attribut

Ein Attribut besitzt einen Namen, einen Wertebereich und das zugehörige Klassenattribut. Die Attribute beschreiben unterschiedliche Eigenschaften der Instanzen und sind sehr wichtig, weil sie das, was für das Lernen verwendet wird, darstellen. In dieser Arbeit werden *nominale* und *numerische* Attribute verwendet. Zum Beispiel ist das Attribut *Gewicht* ein *numerisches* Attribut, welches Fließkommawerte annehmen kann. Nominale Attribute hingegen bestehen aus Zeichensymbolen. Zum Beispiel das Attribut *Geschlecht*, das als Wert entweder *„Male“* oder *„Female“* annehmen kann. Für diese Arbeit sind nur die folgenden Typen relevant:

- *Nominale Attribute* Ein *nominales Attribut* stellt Zeichensymbole dar. Zum Beispiel sind die Attribute *birthName* mit den Wert *„Marc Zwiebler“* und *gender* mit den Wert *„Male“*.

- *Numerische Attribute* Ein *numerisches Attribut* besitzt natürliche Ordnungsrelation und hat als Wert eine *(reelle)* Zahl. Zum Beispiel das Attribut *gewicht* mit den Wert *80*.

Als Beispiel folgende Attribute:

Tabelle 1: Beispiel für Attribute

Name	Typ	*Wertemenge*
gender	Nominal	'Male','Female'
weight	Numeric	0...∞
discipline	Nominal	{'BadmintonPlayer','Sprint_(running)','TennisPlayer'}

Instanz

Eine Instanz beschreibt einen bestimmte Zustand. Dabei stellt jede Instanz ein Beispiel für das zu lernende Konzept dar. Instanzen werden als Vektor von Attributewerten dargestellt. Das Folgende Beispiel zeigt eine *Instanz*, die eine Person aus dem RDF-Graph aus Abbildung 10 beschreibt:

```
birthName ='Marc Zwiebler', gender ='Male', discipline ='BadmintonPlayer', MaleBadmintonPlayer = 'TRUE'
```

Hier gibt es drei *Attribute*: Der Name, *birthName*, das Geschlecht, *gender* und die Sportart, *discipline*. Das Klassenattribut *MaleBadmintonPlayer* zeigt, ob diese Person zu dieser Klasse gehört oder nicht. Die oben gezeigte *Instanz* beschreibt :

- wie diese Person heißt (birthName ='Marc Zwiebler')
- männlich ist (gender ='Male')
- ein BadmintonPlayer ist (discipline ='BadmintonPlayer')
- und gehört zu der Klasse männlichen BadmintonPlayer (MaleBamintonPlayer ='TRUE')

Klassenattribut

Das Klassenattribut ist das letzte Element einer Instanz und repräsentiert den Wert, den das gelerntes Modell vorhersagen soll. Im Beispiel aus Tabelle 2 soll der Lerner eine Vorhersage darüber machen, wann eine Person als männlicher BadmintonPlayer gilt. Das Attribut *MaleBadmintonPlayer* ist ein Klassenattribut.

Klasse

Eine Klasse ist eine Gruppe von Beispielen, die gemeinsame Merkmale besitzen. Ziel eines Regellerners ist es, mit Hilfe gelernter Regeln die Klassen neuer Instanzen vorher zu sagen. In Tabelle 2 gibt es die Klassen *„TRUE“* und *„FALSE“*. Eine Person ist ein männlichen Badmintonspieler, genau dann wenn *MaleBadmintonPlayer* den Wert *„TRUE“* hat und die Wert *„TRUE“* repräsentiert die *Klasse „TRUE“*.

Trainingsmenge

Eine Trainingsmenge ist eine Menge von Instanzen. Die Trainingsmenge enthält das Wissen, welches von dem Algorithmus extrahiert und gelernt werden soll. Alle Instanzen in der Trainingsmenge enthalten Informationen über ihre Klassenzugehörigkeit. Zuerst wird eine Regelmenge mittels einer Trainingsmenge gelernt, anschließend werden andere *Instanzen* aus der Testmenge ohne Klassenzugehörigkeit, mit gewonnen Wissen richtig eingeordnet. Zum Beispiel besitzen die *Instanzen* aus Tabelle 2 eine Zuordnung und können als *Trainingsmenge* benutzt werden. Sie besitzen einen Wert für das Klassenattribut *„MaleBadmintonPlayer“*. Jede *Instanz* wird mit den vier *Attributen „birthName“*, *„gender“*, *„discipline“* und *„MaleBadmintonPlayer“* beschrieben. Ein Beispiel für eine Trainingsmenge ist in Tabelle 2 zu sehen.

Tabelle 2: Beispiel für Trainingsmenge

#	*birthName*	gender	*discipline*	*MaleBadmintonPlayer*
1	'Marc Zwiebler'	'Male'	'BadmintonPlayer'	'TRUE'
2	'Teu Seu Bock'	'Female'	'Coach'	'FALSE'
3	'Usain Bolt'	'Male'	'Sprint_(running)'	'FALSE'
4	'Juline Schenk'	'Female'	'BadmintonPlayer'	'FALSE'
5	'Hossein Rezazadeh'	'Male'	'Weightlifter'	'FALSE'
6	'Boris Becker'	'Male'	'TennisPlayer'	'FALSE'
7	'Marc Zwiebler'	'Male'	'BadmintonPlayer'	'TRUE'
8	'Lee Yong Dae'	'Male'	'BadmintonPlayer'	'TRUE'
9	'Andreas Agasi'	'Male'	'TennisPlayer'	'FALSE'
10	'Timo Boll'	'Male'	'Tabel_Tennis_Player'	'FALSE'

2.4.3 Separate & Conquer-Algorithmen

Beim *überwachtem Lernen* werden Algorithmen eingesetzt, die unterschiedliche Herangehensweise haben und sich in verschiedene Klassen einteilen. Hier werden nur die Algorithmen, die zur Klasse der Separate-and-Conquer Algorithmen (Abk. SeCo-Algorithmen) gehören, vorgestellt. *Separate and Conquer*-Algorithmen werden auch *Covering*-Algorithmen genannt. Der Begriff „Separate and Conquer“ ist von Pagallo und Haussler [22]. Sie haben ihren Ursprung in der AQ-Familie von Algorithmen[23]. Grundsätzlich wird die Suche nach Regeln mittels SeCo-Algorithmen wie folgt eingegangen: Diese

Algorithmen suchen auf einer *Trainingsmenge* eine *Regel* wie folgt: Zuerst wird eine Regel gesucht und alle Instanzen, die von der Regel abgedeckt (*Conquer*) werden, werden aus den Trainingsdaten entfernt *(Separate)*. Dieser Schritt wird rekursiv solange durchgeführt, bis entweder keine positiven Beispiele in der Trainingsmenge vorhanden sind oder die Suche durch Kriterien, die vorher festgelegt sind, beendet[21].

2.4.3.1 Definitionen

Zur Beschreibung der SeCo-Algorithmen benötigen wir die folgenden Definitionen:

Regel

Eine Regel ist von der Form *if C then A*, wobei C (*Regelkörper*) eine Menge von Bedingungen und A (Regelkopf) eine Aussage ist. A enthält die Klassenattribute. Ein Beispiel für die Klasse *MaleBadmintonPlayer* aus Tabelle 2 in Form einer Regel wird hier dargestellt:

(Regelkopf)MaleBadmintonPlayer= **'TRUE'** ← *gender* ='**Male**' *discipline* ='**BadmintonPlayer**'(Regelkörper)

Nach dieser Regel ist ein Beispiel *„TRUE“*, wenn der *Regelkörper* in der Regel erfüllt ist. Der *Regelkörper* besteht aus den Bedingungen *gender=„Male“* und *discipline=„BadmintonPlayer“*. Regelkopf in dieser Regel ist *MaleBadminton-Player=„TRUE“*.

Bedingung

Eine Instanz muss bestimmte Eigenschaften haben, damit sie einer bestimmten *Klasse* zugeordnet werden kann. Eine Bedingung definiert ein bestimmter Wert oder Wertebereich eines bestimmten *Attributs*. Eine *Bedingung* besteht aus einem *Attribut*, einer Operation und dem Wert oder Wertebereich des Attributs. Die Operationen in dieser Arbeit sind = für das Vergleich bei nominalen Attributen und <,≥ bei numerischen Attributen. Zum Beispiel:

```
gender     = 'Male'
age         > 18[Jahre]
discipline = 'BadmintonPlayer'
```

Default-Regel

Ein Default-Regel beinhaltet alle *Instanzen* aus der Trainingsmenge, die nicht durch *Regeln* abgedeckt sind. Sie beinhaltet keine *Bedingungen*. Die Klasse für eine *Default-Regel* ist immer diejenige, die in den Trainingsmengen am häufigsten vorkommt.

Heuristik

Ein Verfahren, dass beim Regellernen verwendet wird, um die Güte einer Regel abzuschätzen.

Theorie

Eine Theorie bezeichnet die Menge der Regeln. Die Regeln werden von einem Regellerner mittels Heuristiken gelernt.

Generalisierung

Bedingungen aus dem Bedingungsteil einer Regel werden fallen gelassen oder gelockert, so dass die Regel auf eine größere Anzahl von Situationen anwendbar ist. Die Größe des Suchraums wird durch die Anzahl der *Attribute* und *Attributwerte* festgelegt.

Spezialisierung

Zusätzliche Bedingungen werden dem Bedingungsteil einer Regel hinzugefügt oder vorhandene Bedingungen verschärft, so dass die Regel nur noch eine kleinere Anzahl von Datensätzen bezeichnet.

2.4.3.2 Merkmale von Seco-Algorithmen

Die verschiedenen SeCo-Algorithmen lassen sich nach drei Merkmalen unterscheiden: der Hypothesensprache, dem Suchverfahren und der verwendeten Heuristik zur Vermeidung von Überbestimmtheit.

Hypothesensprache

In der Hypothesensprache wird das jeweils zu lernende Konzept beschrieben. Die Hypothesensprachen sind *Regelmengen* mit unterschiedlichen Spezifikationen. Die *Regelmengen* können in *disjunktiver Normalform (DNF)*,*konjunktiver Normalform (KNF)* oder als *Entscheidungslisten* beschrieben werden[21].

- *Disjunktiver Normalform (DNF)*: Eine Form der Konzeptbeschreibung. Für jede Klasse wird Disjunktion von Konjunktionen von Literalen erzeugt. Ein Literal beschreibt eine Bedingung 2.4.3.1. Zum Beispiel:

$$(\texttt{gender} = \texttt{'Male'} \bigwedge \texttt{discipline} = \texttt{'BadmintonPlayer'}) \bigvee (\texttt{height} > 180)$$

- *Konjunktiver Normalform (KNF)*: Die konjunktive Normalform ist eine Konjunktion von disjunktiv verknüpften Literalen. Zum Beispiel:

$$(\texttt{gender} = \texttt{'Male'} \bigvee \texttt{discipline} = \texttt{'BadmintonPlayer'}) \bigwedge (\texttt{height} > 180)$$

- *Entscheidungslisten*: Eine Menge von Regeln, die entweder in *DNF* oder *KNF* ist. Die Reihenfolge von Regeln ist wichtig für die Klassifizierung. Möchte man nun prüfen, zu welcher Klasse eine *Instanz* gehört, werden die Regeln der Liste der Reihenfolge nach auf diese *Instanz* angewendet, bis alle Bedingungen erfüllt sind. Am Ende dieser Liste steht die Default-Regel, für den Fall, dass keine andere Regel zutrifft.

Suchverfahren

Anhand einem Suchverfahren wird definiert, wie der *Hypothesenraum* beim Lernen einer *Regelmenge* durchsucht werden soll. *Suchalgorithmus*,*Suchstrategie* und *Suchheuristik* beschreiben einen Suchverfahren. Mit dem *Suchalgorithmus* können neue Hypothesen im Hypothesenraum gefunden und verfeinert werden. Eine *Suchstrategie* eines Suchalgorithmus legt fest, wie der Hypothesenraum nach Lösungen durchgesucht werden. In *SeCo-Algorithmen*, die hier betrachtet werden, gibt es zwei Suchstrategien: *Generalisierung* und *Spezialisierung*. Beim *Generalisierung* werden *Bedingungen* aus dem *Regelkörper* entfernt, um möglichst mehrere Beispiele abzudecken. Im Gegensatz dazu werden bei der *Spezialisierung* weitere *Bedingungen* zum *Regelkörper* eingefügt, um bestimme Beispiele abzudecken. Der Raum für Hypothesen ist im allgemeinen sehr groß, so dass geeignete *Heuristiken* gefunden werden müssen, um den Suchraum zu beschränken. Weiterhin kann man anhand von Heuristiken genau den Wert der Regel einschätzen.

Heuristik zur Vermeidung von Überbestimmtheit

Überbestimmtheit tritt auf, wenn eine Regelmenge sich zu sehr auf die Trainingsmenge angepasst hat. Dies führt dazu, dass es in manchen Fällen für jedes Beispiel in der Trainingsmenge genau eine Regel gibt, so dass die Regel genau nur diese Beispiele abdeckt. Um Überbestimmtheit zu vermeiden, werden *Pre-Pruning* oder *Post-Pruning* verwendet. Viele SeCo-Algorithmen verwenden diese Techniken, um entweder vor oder nach dem Aufstellen einer Regelmenge Überbestimmtheit zu verhindern.

```
procedure SeparateAndConquer(examples)
{
  theory := ∅
  while positive (examples)≠∅
  {
    rule := FindBestRule (examples)
    covered := cover (rule, examples)
    if RuleStoppingCriterion (theory, rule, examples)
        exit while
    examples := examples \ covered
    theory := theory ⋃ rule
  }
  theory := PostProcess (theory)
  return (theory)
}

procedure FindBestRule (examples)
{
   initRule := InitializeRule (examples)
   initVal := EvaluateRule (initRule)
   bestRule := <initVal, initRule>
   rules := {bestRule}
   while rules≠∅
   {
      candidates := SelectCandidates (rules, examples)
      rules := rules \ candidates
      for candidate∈candidates
      {
         refinements := RefineRule (candidate, examples)
         for refinement∈refinements
         {
            evaluation := EvaluateRule (refinement, examples)
            while (!StoppinCriterion (refinement, evaluation, examples) )
            {
               newRule := <evaluation, refinement>
               rules := InsertSort (newRule, rules)
               if ( newRule > bestRule )
                  bestRule := newRule
            }
         }
      }
      rules := FilterRules (rules, examples)
   }
   return (bestRule)
}
```

Quelltext 1: Allgemeiner SeCo-Algorithmus [24]

2.4.3.3 Funktionsweise von SeCo-Algorithmus

In Abbildung 2.4.3.2 wird einen generischen SeCo-Algorithmus vorgestellt, der alle Algorithmen dieser Klasse repräsentiert [21]. Der Algorithmus besteht aus zwei Prozeduren: **FindBestRule** und **SeparateAndConquer**. Er startet mit der leeren Regelmenge(Theory). Wenn es positive Beispiele in der Trainingsmenge gibt, wird die Prozedur **FindBestRule** aufgerufen, um eine Regel zu lernen, die positive Beispiele abdeckt. Anschließend werden alle positive Beispiele, die von dieser Regel abgedeckt sind, aus der Trainingsmenge entfernt und die Regel zur Regelmenge hinzugefügt. Diese Prozedur wird solange wiederholt und dabei neue Regeln gelernt, bis kein weiteres positives Beispiel in der Trainingsmenge existiert oder das Kriterium **RuleStoppingCriterion** erfüllt ist. Danach wird **PostProcess** ausgeführt. Die Prozedur **FindBestRule** untersucht den Hypothesenraum nach einer geeigneten Regel. Diese Prozedur gibt die beste gefunden Regel zurück. Zuerst wird eine initiale Regel, die als beste Regel markiert ist, in die Regelmenge aufgenommen. Aus der Regelmenge werden Regeln für die Verfeinerung ausgewählt und zur Menge der Kandidaten hinzugefügt. Für jeden Kandidat wird mit **RefineRules** eine Verfeinerung vorgenommen[25]. Es gibt zwei Suchstrategien, wie eine Verfeinerung an Regeln durchgeführt werden kann:

- Top-Down: Es wird eine allgemeine Regel, die meist die Form: *true*→Klasse hast, durch Einfügen von Bedingungen so verändert, dass sie möglichst viele positive Beispiele abdeckt.

- Bottom-Up: Eine Regel, die genau ein positives Beispiel abdeckt, wird durch Entfernen von Bedingungen so generalisiert, dass sie möglichst viele positive Beispiele und möglichst wenige negative Beispiele abdeckt.

2.4.3.4 SeCo-Framework

Das *SeCo-Framework* ist eine Implementierung den SeCo-Algorithmen in Java. Es wurde von der *Knowledge Engineering Group* an der TU Darmstadt entwickelt. Dadurch, dass das *SeCo-Framework* modular aufgebaut ist, können alle Algorithmen, die als Suchstrategie den *SeCo-Algorithmus* verwenden, hinzugefügt werden. Im *SeCo-Framework*[26] sind verschiedene Eigenschaften von Lernalgorithmen durch eine XML-Konfigurationsdatei einstellbar. Das *SeCo-Framework* verwendet eine Entscheidungsliste, um neue Beispiele zu klassifizieren. Das *Defaultlernverfahren* sortiert zuerst alle Werte eines Klassenattributs nach Anzahl der Instanzen mit diesem Klassenattribut. Danach wird für jeden möglichen Wert des Klassenattributs außer dem Wert mit der höchste Anzahl an Beispielen in der Trainingsmenge Regeln gelernt. Das *SeCo-Framework* besteht aus drei modular aufgebauten Packages *seco.models*, *seco.learners* und *seco.heuristics*. Die Trainingsmenge werden dem *SeCo-Framework* in der Form einer *ARFF-Datei* bereitgestellt. Regeln liegen im *SeCo-Framework* in *disjunktiver Normalform* vor. Es gibt zwei Arten von *Attributen*, *nominale* und *numerische*. Die *nominalen* Attribute repräsentieren im *SeCo-Framework* eine endliche Menge von möglichen Zeichenketten ohne Ordnungsrelation und die *numerische* Attribute repräsentieren Der implementierte Suchalgorithmus im *Seco-Framework* ist eine *Top-Down-Suche*. Über eine *XML-Datei (Config-Datei)* kann einen Lern-Algorithmus aus unterschiedlichen Teilen zusammengesetzt werden. Bei Thiel[25] ist beschrieben, wie diese *XML-Datei (Config-Datei)* aufgebaut ist. Mit einer *Factory* und einer *XML-Datei (Config-Datei)* kann man aus beliebigen *SeCo-Komponenten* einen Regellerner zusammenbauen. Im *Seco-Framework* gibt es drei wichtige Komponente, die in folgenden vorgestellt werden:

- seco.models enthält die Klassen für das Modell der Hypothesensprache. Tabelle 3[21] stellt alle Klassen des Packages *seco.models* dar:

Tabelle 3: Klassen von seco.models

Klasse	Beschreibung
Condition	Elternklasse der Bedingungen
NominalCondition	Nominale Bedingungen
NumericCondition	Numerische Bedingungen
Rule	Regel
RuleComparator	Testet 2 Regeln auf Gleichheit
RuleSet	Regelset
CandidateRule	Regel mit Zusatzinformation

- seco.learners enthält verschiedene Regellernalgorithmen, mit deren Hilfe *Regeln* und *Regelsätze* gefunden werde können. Das Package *seco-learners* ist die Implementierung des Algorithmus (s. Abschnitt 2.4.3.2). Die Klasse *Covering* ist die Implementierung der äußeren Schleife *SeparateAndConquer* und *TopDownBeamSearch* ist die Implementierung von *FindBestRule*.

- seco.heuristics enthält die Klassen für die Heuristiken.

2.4.4 Weka

Weka (*Waikato Enviroment for Knowledge Analysis*)[26] stellt eine Sammlung von verschiedenen *Machine Learning* Instrumenten und wurde an der *University of Waikato* entwickelt. Weka ist eine in Java entwickelte Software, die unter den Bedingungen der GNU (*General Public License*) verwendet werden kann. Weka enthält eine Sammlung von Visualisierungswerkzeugen und Algorithmen zur Datenanalyse, Regressionsanalyse, Visualisierung, Feature-Auswahl und zur Clusteranalyse. Weiterhin gibt es umfangreiche Sammlungen von Algorithmen, die z.B. zur Klassifizierung eingesetzt werden können. In dieser Arbeit werden die Trainingsdaten zuerst in das von Weka eigenentwickelte Dateiformat ARFF

[26] http://www.cs.waikato.ac.nz/ml/weka/

(*Attribute Relation File Format*) umgewandelt, und dann zum Regellernen an das SeCo-Framework weitergereicht. Im folgenden wird die Struktur einer ARFF-Datei kurz beschrieben[27].

ARFF-Datei

Eine *ARFF-Datei* beschreibt einer Menge von Instanzen, die durch eine Menge von Attributen dargestellt sind. *ARFF-Dateien* sind in zwei Abschnitte aufgeteilt, *Header*- und *Data*-Informationen. Kommentare beginnen mit einem Prozentzeichen und können überall innerhalb der ARFF-Datei vorkommen:

```
% ARFF-Datei car.arff
@RELATION car.arff
```

- *Header-Abschnitt:* Am Anfang des Header-Abschnitts steht der Name der Relation:

  ```
  %RELATION <relation-name>
  Bsp: @RELATION car.arff
  ```

  Namen, die Leerzeichen enthalten, müssen in einfache Anführungszeichen gesetzt werden.

- Deklaration der Attribute: Eine Attribute wird in folgender Form definiert:

  ```
  @attribute <attribute-name> <datatype>
  Bsp:
  @attribute gender       {female,male}
  @attribute discipline   {singlePlayer, doublePlayer, mixedPlayer}
  @attribute age           real
  @attribute birthPlace   {Germany,England,USA,China,Iran,Korea}
  @attribute class        {TRUE,FALSE}
  ```

- *Klassenattribut* Das Klassenattribut wird nach allen Attributen im Header in dieser Form definiert:

  ```
  @attribute class {TRUE,FALSE}
  ```

- *<datatype>* Es gibt vier mögliche Datentypen:

  - *numeric*: Numerische Attribute können real oder integer sein. Hierfür wird als <datatype> nur *real* eingetragen. Zum Beispiel:

    ```
    @attribute age       real
    ```

  - *<nominal-specification*: Nominale Werte repräsentieren mögliche Ausprägungen eines Attributs. Nominale Werte werden in einer geschweiften Klammer aufgezählt. Zum Beispiel:

    ```
    @attribute gender       {female,male}
    ```

  - *string*: String-Attribute repräsentieren Zeichenwerte.Zum Beispiel:

    ```
    @attribute birthName       string
    ```

  - *date [<date-format>]*: Daten-Attribute werden durch das Format „yyyy-MM-dd HH:mm:ss“ repräsentiert.

- *Data-Abschnitt:* beginnt mit *@data*, danach folgt pro Zeile eine Instanz, wobei die Werte derer Attribute aufgelistet und durch ein Hochkomma getrennt sind. Die Werte müssen in der gleichen Reihenfolge angegeben werden wie im Attribut-Abschnitt festgelegt. Fragezeichen repräsentieren fehlende Werte. Ein Beispiel für einen Data-Abschnitt:

  ```
  @data
  female,singlePlayer,28,Germany,FALSE
  female,singlePlayer,23,USA,TRUE
  male,mixedPlayer,22,China,FALSE
  male,singlePlayer,36,Iran,TRUE
  male,doublePlayer,30,Korea,FALSE
  ```

3 Klassifizierung auf Open Data mit SeCo

In diesem Kapitel wird die konzeptuelle Vorgehensweise beim Lernen auf Open Data mit SeCo detailliert beschrieben. Es beschreibt die Grundidee unseres Tools `db2seco`, das aus *SPARQL-Abfragen* und Prädikaten mit den SeCo-Algorithmen lernt. Die Implementierung wird im nächsten Kapitel getrennt beschrieben.

3.1 Ansatz

Zuerst werden Daten aus einer Linked Data-Datenbank (z.B. DBpedia) anhand einer SPARQL-Abfrage geholt. Es ist empfehlenswert dabei den *Solution Modifier* LIMIT zu benutzen, um das Ergebnis der SPARQL-Abfrage einzuschränken. Der Benutzer kann markieren, welche Daten zu der Klasse gehören, die gelernt werden soll. Zwecks Lernen werden die Daten in das von SeCo verwendeten Datenformat (ARFF-Datei) 2.4.4 umgeschrieben. Anschließend wird generierte ARFF-Datei an das SeCo-Tool weitergegeben. Anhand dieser ARFF-Datei, die als Trainingsmenge für den SeCo-Algorithmus dient, generiert der SeCo-Tool eine *Regelmenge*. Der Inhalt der *Regelmenge* wird in die ursprüngliche SPARQL-Abfrage eingefügt, so dass die neue SPARQL-Abfrage die gesuchte Klasse beschreibt. Mit der gelernten Regelmenge kann ebenfalls gelernt werden, indem sie als Prädikat in die ARFF-Datei aufgenommen wird. Abbildung 18 illustriert diese Vorgehensweise:

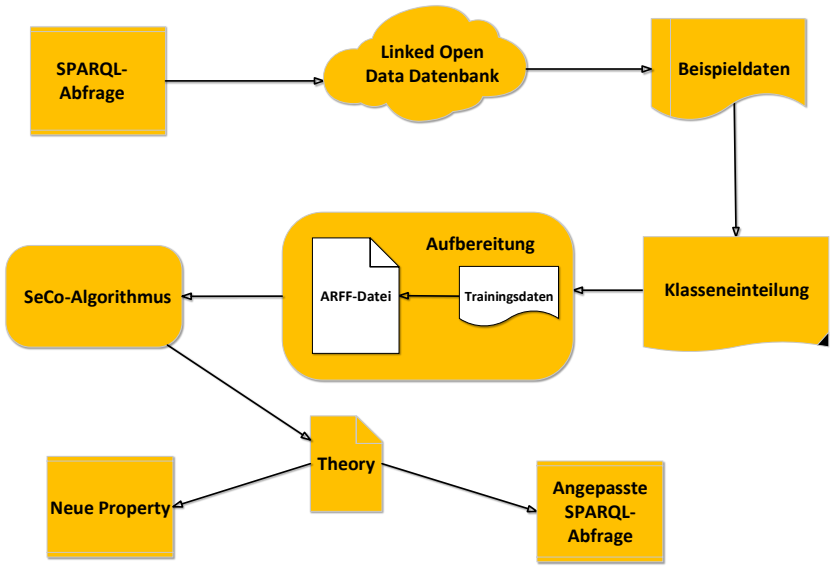


Abbildung 18: Architektur von db2seco

Bei den gesamten Verarbeitungsprozess ergeben sich Probleme, weil die Technologien unterschiedliche Annahmen über das Format der Daten haben. Die SeCo-Implementierung benötigt wohlgeformte Eingabedaten. Im Semantic Web gilt dagegen das AAA-Prinzip *"Andybody can say anything about anything"* (Allemang & Hendler). Es besagt, dass jeder im Web jeder alles über alles sagen kann. Das Grundkonzept von Linked Open Data sieht also nicht vor, dass die Daten einem Schema entsprechen müssen. Zum Beispiel in DBPedia werden Daten in Form eines RDF-Graphen gespeichert. Einige Datensätze entsprechen einem Schema, nämlich *dbpedia.org/ontology*, andere aber nicht. Daher kommt es sehr oft vor, dass Datensätze in DBPedia fürs maschinelle Lernen nur bedingt geeignet sind; sehr viele Einträge sind nicht vorhanden oder im falschen Format.

In den folgenden Abschnitten wird beschrieben, wie man die Technologien kombinieren muss und wie die Daten aus einem *Dataset* angepasst werden, um vom SeCo-Framework benutzt werden zu können.

3.2 Datenabfrage

Die SPARQL-Abfrage wird an das Tool `db2seco` gegeben. Die *SPARQL*-Abfrage wird zusammen mit einer *Graph IRI* direkt an *Jena* (siehe Abschnitt 2.3.1) übergeben. Jena schickt die *SPARQL*-Abfrage an die Linked Data-Datenbank weiter. Im

Fall eines Fehlers bei der Abfrage wird dieser dem Benutzer angezeigt. db2sceo speichert das Ergebnis der Abfrage, um die Ergebnisse dem Benutzer anzuzeigen.

3.3 Datenaufbereitung und Klassifizierung

Aus dem Abfrage-Ergebnis werden Instanzen, die vom Benutzer markiert werden sollen, generiert. Da wir mit *CONSTRUCT*-Abfragen arbeiten, liegen die Daten in Form eines RDF-Graphen vor. Für jedes Subjekt, das im RDF-Graphen vorkommt, wird jetzt eine Instanz erstellt. Immer wenn es im RDF-Graphen für ein Subjekt und ein Prädikat keinen Eintrag gibt, wird das Symbol „*?*“ als Wert verwendet, weil dieses Symbol im *ARFF-Format* für unbekannte Werte vorgesehen ist. Außerdem bekommen jeder Datensatz ein zusätzliches Datenfeld. Das Datenfeld repräsentiert den Attributwert für die zu lernenden Klasse und ist editierbar. In der GUI-Version des Programms werden in einer Tabelle alle Instanzen eingetragen.

Allerdings kann es sein, dass wir für gleiches Subjekt und Prädikat mehrere Einträge als Objekt im RDF-Graphen finden. Das folgende Beispiel illustriert das:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

CONSTRUCT {
                ?person dbp:name ?birthName.
                ?person dbo:weight ?weight.
                ?person dbo:birthDate ?birthDate.
                ?person dbo:birthPlace ?birthPlace.
           }
WHERE    {      ?person rdf:type dbo:Person.
                ?person dbp:name ?birthName.
                ?person dbo:weight ?weight.
                ?person dbo:birthDate ?birthDate.
                ?person dbo:birthPlace ?birthPlace.
                FILTER (?birthName = "Jessica Messenger"@en)
          }
LIMIT 10
```

Das Ergebnis der oben gezeigten Abfrage wird hier in *N3* (s. Abschnitt 2.1.1.4) dargestellt:

```
dbr:Jessica_Messenger dbo:birthDate "1989+01:00"^^xsd:gYear.
dbr:Jessica_Messenger dbo:birthDate "1989+02:00"^^xsd:gYear.
dbr:Jessica_Messenger dbo:weight    "44452.1"^^xsd:double.
dbr:Jessica_Messenger dbp:name      "Jessica Messenger"@en.
dbr:Jessica_Messenger dbo:birthPlace  dbr:Derbyshire.
dbr:Jessica_Messenger dbo:birthPlace  dbr:England.
dbr:Jessica_Messenger dbo:birthPlace  dbr:UK.
```

Für das Prädikat *dbo:birthPlace* werden mehrere Werte zurückgeliefert.

Es gibt mehrere Möglichkeiten, damit umzugehen:

1. Man könnte alle vorgekommenen Werte des Prädikats in einer Menge eintragen und die Menge würde den Wert des Prädikats darstellen. Ein Beispiel wären *Blank Nodes* 2.1.1.5, mit denen eine Liste von mehreren Objekten dargestellt wird.

2. Man erzeugt für alle vorgekommenen Werte jeweils eine eigene Instanz.

3. Es wird einen zufälligen Wert des Prädikats übernommen und alle anderen vorgekommenen Werte ignoriert.

Die erste Möglichkeiten eignet sich nicht, weil das *ARFF-Format* es nicht zulässt, dass ein Attributewert in einer Instanz mehrere Werte enthält. Die zweite Möglichkeiten ist ebenfalls nicht geeignet. Wenn man für alle Prädikatwerte eigene Instanzen definiert, werden diese neuen Instanzen die Lernergebnisse übermäßig beeinflussen: Wenn ein Subjekt mehrmals vorkommt, wird es wichtiger, denn es hat mehr Einfluss auf die Metriken beim Regellernen, weil dabei die richtigen und falschen Instanzen gezählt werden. Deswegen wurde in dieser Arbeit die dritte Möglichkeit gewählt.

3.4 ARFF-Datei

Damit *SeCo-Framework* aus den vom Nutzer gelabelten Beispieldaten lernen kann, werden sie in einer *ARFF-Datei* gespeichert. Dazu muss zuerst der Header der *ARFF-Datei* geschrieben werden. Für jedes Prädikat aus den Beispieldaten wird eine Zeile geschrieben. Wenn die Werte eines Prädikats Zahlen sind, dann wird *„real“* als *<datatype>* geschrieben. Für alle andere Arten von Werten wird der Typ *Nominal* verwendet.[27] Für *Nominal* werden alle Werte des Prädikats aufgezählt und in Hochkommata gestellt. Nachdem alle Attribute geschrieben sind, folgt das *Klassenattribut* mit den Werten *„TRUE“* und *„FALSE“*. Danach folgen die Instanzen als Trainingsbeispiele für das *SeCo-Framework*. Zu jedem Subjekt aus den Trainingsdaten werden die Prädikate in die *ARFF-Datei* geschrieben und zuletzt das *Klassenattribut* dieser Instanz angehängt.

Ein wichtiges Problem beim Schreiben ist, dass die Prädikate in einer *ARFF-Datei* typisiert sind, damit man für die unterschiedliche Datentypen unterschiedliche Arten von Regeln definieren kann. Im semantischem Web gilt das Prinzip *„AAA“* 3.1 Dadurch ergeben sich die folgende zwei Probleme: 1. Die Daten sind nicht von dem Typ, der für ein Prädikat vorgesehen ist 2. Der Datentyp ist für unterschiedliche Instanzen unterschiedlich.

3.4.1 Daten vom falschen Typ

Es kann vorkommen, das der Wert eines Prädikats in einem *Dataset* nicht den Typ hat, der für dieses Prädikat vorgesehen ist. Folgendes Beispiel illustriert das:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

CONSTRUCT {
               ?person dbp:name ?birthName.
               ?person dbo:birthDate ?birthDate.
               ?person dbo:weight ?weight.
               ?person dbo:birthPlace ?birthPlace.
          }
WHERE    {     ?person rdf:type dbo:Person.
              ?person dbp:name ?birthName.
              ?person dbo:birthDate ?birthDate.
              ?person dbo:weight ?weight.
              ?person dbo:birthPlace ?birthPlace.
              FILTER ( ?birthName = "Jessica Messenger"@en)
         }
LIMIT 10
```

Diese Abfrage liefert als Ergebnis für das Subjekt *„Jessica Messenger“* das Prädikat *dbo:birthDate* den Wert *„1989“* mit dem Datentyp *xsd:Integer*.

```
dbr:Jessica_Messenger dbo:birthDate    "1989+01:00"^^xsd:gYear.
dbr:Jessica_Messenger dbo:birthDate    "1989+02:00"^^xsd:gYear.
dbr:Jessica_Messenger dbo:weight       "44452.1"^^xsd:double.
dbr:Jessica_Messenger dbp:name         "Jessica Messenger"@en.
dbr:Jessica_Messenger dbo:birthPlace   dbr:Derbyshire.
dbr:Jessica_Messenger dbo:birthPlace   dbr:England.
dbr:Jessica_Messenger dbo:birthPlace   dbr:UK.
```

Das Prädikat *dbo:birthDate* hat aber als *rdfs:range* den Datentype *xsd:date*. Im *ARFF-Format* muss jeder Datensatz dem Typ entsprechen, der im Header definiert ist. Deshalb wird in solchen Fällen in der ARFF-Datei das Symbol *„?“* für diese Instanz als Wert des Prädikats eingetragen.

3.4.2 Inkonsistente Datentypen

In den Trainingsdaten können unterschiedliche Datentypen als Wert eines Prädikats vorkommen. Die folgende Abfrage 19 illustriert das Problem:

[27] Das *SeCo-Tool* unterstützt in der verwendeten Version nur *Nominal* und *Numeric*.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

CONSTRUCT {
                ?person dbp:name ?birthName.
                ?person dbo:weight ?weight.
                ?person dbo:birthDate ?birthDate.
                ?person dbo:coach ?coachName.
            }
WHERE    {       ?person rdf:type dbo:Person.
                ?person dbp:name ?birthName.
                ?person dbo:weight ?weight.
                ?person dbo:birthDate ?birthDate.
                OPTIONAL{ ?person dbo:coach ?coachName.}
                ?person dbo:birhPlace <http://dbpedia.org/resource/Derbyshire>.
          }
LIMIT 10
```

Abbildung 19: SPARQL-Abfrage über alle Personen, die aus *Derbyshire* stammen

Abbildung 19 zeigt eine SPARQL-Abfrage über Personen, die in *Derbyshire* geboren sind. Das Ergebnis der SPARQL-Abfrage aus Abbildung 19 wird in Abbildung 20 gezeigt:

```
dbr:Jo_Guest           dbo:birthDate     "1972-02-22+02:00"^^xsd:date.
dbr:Jo_Guest           dbo:weight        "53900.0"^^xsd:date.
dbr:Jo_Guest           dbp:name          "Jo Guest"@en.
dbr:Ross_Davenport     dbo:birthDate     "1972-02-22+02:00"^^xsd:date.
dbr:Ross_Davenport     dbo:weight        "76204.8"^^xsd:date.
dbr:Ross_Davenport     dbp:name          "Ross_Davenport"@en.
dbr:Jessica_Messenger  dbo:birthDate     "1989+02:00"^^xsd:gYear.
dbr:Jessica_Messenger  dbo:weight        "44452.1"^^xsd:double.
dbr:Jessica_Messenger  dbp:name          "Jessica Messenger"@en.
```

Abbildung 20: Ergebnis einer SPARQL-Abfrage mit einem inkonsistenten Datentyp

Das Prädikat *dbo:birthDate* hat als *range* den Datentyp *xsd:date*. Allerdings werden hier für das Subjekt *Jessica Messenger* unterschiedliche Datentypen als Wert des Prädikats zurückgeliefert. In einer der Instanzen ist das der Typ `gYear` *<http://www.w3.org/2001/XMLSchema#gYear>* und in einer anderen `date` *<http://www.w3.org/2001/XMLSchema#date>*.

Als Lösung werden die Datentypen lokal gespeichert und der am häufigsten vorgekommene Datentyp festgelegt. Im oben gezeigten Beispiel wird den Datentyp *Date* für das Prädikat *dbo:birthDate* festgelegt. Eine andere Lösung wäre, für das jeweilige Prädikat den vordefinierten Wertebereich *range <http://www.w3.org/2000/01/rdf-schema#range>* abzufragen und den zugehörigen Datentyp zu übernehmen. Es kommt aber vor (z.B. *dbp:dateOfBirth*), dass für ein Prädikat keine *range* vordefiniert ist. Daher wird diese Lösung nicht angewendet.

3.5 Lernen

Die produzierte *ARFF-Datei* wird an das *SeCo-Tool* zum Lernen einer *Regelmenge* weitergereicht. Dabei wird eine in `db2seco` spezifizierte *Konfigurationsdatei* ebenfalls an das *SeCo-Tool* weitergegeben. Aus der *ARFF-Datei* werden die Trainingsmenge und das Ergebnis als *Regelmenge* generiert. Das *SeCo-Tool* rechnet die Klassifikation für die Trainingsmenge aus. *Precision* und *Recall* werden zusammen mit der *Regelmenge* in der *GUI* angezeigt. Um *Precision* und *Recall* zu berechnen, müssen für die *Instanzen* Unterscheidungen getroffen werden. Dazu werden die Trainingsmenge an das *SeCo-Tool* weitergegeben, um richtige und falsche Klassifizierung festzustellen. In den graphischen Ausgabe des *SeCo-Tools* werden die *Instanzen* in einer Tabelle farbig dargestellt.

3.6 Anpassung der SPARQL-Abfrage

Aus der gelernten *Regelmenge* wird die neue *SPARQL-Abfrage* erzeugt. Dazu wird die alte *SPARQL-Abfrage* verändert, um die durch das *SeCo-Tool* gelernte *Regelmenge* zu integrieren. Für jede einzelne Regel aus der *Theory* wird die entsprechende

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

CONSTRUCT {
                ?person dbp:name ?birthName.
                ?person dbo:birthPlace ?country.
                ?person dbo:weight ?weight.
                ?person dbo:birthDate ?birthDate.
                ?person dbo:coach ?coachName.
            }
WHERE    {       ?person rdf:type dbo:BadmintonPlayer.
                ?person dbp:name ?birthName.
                ?person dbo:weight ?weight.
                ?person dbo:birthDate ?birthDate.
                OPTIONAL{ ?person dbo:coach ?coachName.}
                ?person dbo:birthPlace ?country.
                ?country rdf:type dbo:Country.
          }
LIMIT 10
```

Abbildung 21: SPARQL-Abfrage über alle Badmintonspieler mit unter anderem der Stadtangehörigkeit

Variable ausgewählt, um sie in ein *FILTER-Element* einzubauen. Das *FILTER-Element* wird in das *Graphpattern* der alten *SPARQL-Abfrage* eingefügt. Das neue *SPARQL-Abfrage* wird in der *GUI* angezeigt.

Abbildung 21 illustriert eine *SPARQL-Abfrage* über alle Badmintonspieler mit unter anderem dem Geburtsort (*dbo:birthPlace*) und Abbildung 22 zeigt eine *ARFF-Datei*. In dieser *ARFF-Datei* haben *Instanzen*, die als Stadtangehörigkeit (*dbo:birthPlace)* den Wert *<http://dbpedia.org/resource/Indonesia>* haben als Klassenattribut den Wert *„TRUE“*. In Abbildung 23 wird die vom *SeCo-Framework* generierte Regelmenge der Trainingsmenge aus Abbildung 22 gezeigt. In Abbildung 24 wird die neue *SPARQL-Abfrage* gezeigt, die von `db2seco` generiert wurde. Das *FILTER-Pattern FILTER (?country = <http://dbpedia.org/resource/Indonesia>)* wurde zur alten SPARQL-Abfrage hinzugefügt. Um diese *FILTER-Pattern* zu erzeugen, wird für die vom *SeCo-Tool* gelernten Regel *http://dbpedia.org/ontology/birthPlace = http://dbpedia.org/resource/Indonesia* das Triple gesucht, in dem das Prädikat *dbo:birthPlace* vorkommt. Aus diesem Triple wird das Objekt ausgesucht, in diesem Fall die Variable *?country*. Aus dieser Variable und den Attributwert *<http://dbpedia.org/resource/Indonesia>* wird ein *FILTER-Pattern* erzeugt.

Im Allgemeinen besteht eine *Regelmenge* aus mehreren Regeln, die jeweils als Konjunktion ausgedrückt sind. Für jede Bedingung aus einer Regel wird eine Filterbedingung erstellt. Da jede Regel aus mehreren Bedingungen bestehen kann, die gleichzeitig erfüllt sein müssen, werden die Filterbedingungen mit dem logischem „Und“ zusammengesetzt. Wenn es mehr als eine Regel in der Regelmenge gibt, werden die Filterbedingungen für jede Regel zu einer Disjunktion verbunden.

Interessanterweise liefert die neu gelernte Abfrage als Ergebnis neue Datensätze, die bei der alten Abfrage nicht vorgekommen sind, wenn eine Abfrage den solution modifier LIMIT enthält. Diese Datensätze treffen auf die alte Abfrage zu, sind aber nicht im Ergebnis der alten Abfrage vorgekommen, weil das Abfragelimit erreicht war. Erwartungsgemäß kommen in der neuen Abfrage diejenigen alten Datensätze nicht vor, die dem Pattern nicht entsprechen.

Variablen können in unterschiedlichen Teilpattern einer *SPARQL-Abfrage* vorkommen. Ein Filter-Pattern kann solche Variablen enthalten. Wenn ein Filter-Pattern zu einer SPARQL-Abfrage hinzugefügt wird, ist es nicht ersichtlich, welche Variablen aus welchem Teil-Graphen gefiltert sind. Da die SPARQL-Abfrage einen Graphen als Ergebnis liefert und das Filterpattern sich auf den gesamten Graphen bezieht, spielt dieser Fall keine große Rolle.

3.7 Einfügen eines Prädikats

In der GUI kann der Benutzer aus der gelernten *Regelmenge* ein neues Prädikat mit einem selbstgewählten Namen erzeugen. Dazu werden neue Tripel zu dem RDF-Graphen hinzugefügt, die einen *boolean*-Wert haben. Der Benutzer kann aber auch zuvor seine Auswahl ändern und wieder Seco starten, falls ihm die gelernte Regelmenge nicht richtig erscheint. Mit der jeweils neuen Regelmengen kann der Benutzer dann weitere Prädikate lernen.

```
@ATTRIBUTE 'http://dbpedia.org/ontology/birthDate' {'1992-01-30',
     '1980-02-08','1989-03-30','1982-02-22','1990-02-24','1989-09-02'}

@ATTRIBUTE 'http://dbpedia.org/ontology/birthPlace'
     {'http://dbpedia.org/resource/Thailand',
      'http://dbpedia.org/resource/India',
      'http://dbpedia.org/resource/Indonesia}

@ATTRIBUTE 'http://dbpedia.org/ontology/weight' 'real'

@ATTRIBUTE 'http://dbpedia.org/ontology/coach'
   {'http://dbpedia.org/resource/Chen_Qiqiu',
    'http://dbpedia.org/resource/Pullela_Gopichand',
    'http://dbpedia.org/resource/Zhang_Ning'}

@ATTRIBUTE 'http://dbpedia.org/property/name'
  {'Wang Lin\@{en}','Yao Lei\@{en}','Sudket Prapakamol\@{en}',
   'Prapakamol,'Sudket\@{en}','Xia Huan\@{en}','Ponsana,
   'Boonsak\@{en}','Wang, Lin\@{en}','Pantawane, Arundhati\@{en}',
   'Boonsak Ponsana\@{en}','Arundhati Pantawane\@{en}'}

@ATTRIBUTE Class {'TRUE','FALSE'}

@data

'1989-03-30','http://dbpedia.org/resource/Indonesia,'57000.0',
    'http://dbpedia.org/resource/Zhang_Ning','Wang Lin\@{en}',TRUE
'1982-02-22','http://dbpedia.org/resource/Thailand','70000.0',?,
    'Boonsak Ponsana\@{en}',FALSE
'1990-02-24','http://dbpedia.org/resource/Indonesia,'65000.0',?,
    'Yao Lei\@{en}',TRUE
'1989-09-02','http://dbpedia.org/resource/India','50000.0',
    'http://dbpedia.org/resource/Pullela_Gopichand',
    'Arundhati Pantawane\@{en}',FALSE
'1992-01-30','http://dbpedia.org/resource/Indonesia,'65000.0',
    'http://dbpedia.org/resource/Chen_Qiqiu','Xia Huan\@{en}',TRUE
'1980-02-08','http://dbpedia.org/resource/Thailand',
    '65000.0',?,'Prapakamol, Sudket\@{en}',FALSE
```

Abbildung 22: ARFF-Datei der SPARQL-Abfrage aus Abb 22

```
                number of rules.............: 1
                number of conditions........: 1
                referred attributes.........: 1
                average rule lenght.........: 1.0
                RuleSet.....................:
Class = TRUE :- http://dbpedia.org/ontology/birthPlace = http://dbpedia.org/resource/Indonesia
                defaultRule.................: Class = FALSE
```

Abbildung 23: Die vom *SeCo-Framework* generierte Regelmenge der Trainingsmenge aus Abb 22

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>

CONSTRUCT {
                ?person dbp:name ?birthName.
                ?person dbo:birthPlace ?country.
                ?person dbo:weight ?weight.
                ?person dbo:birthDate ?birthDate.
                ?person dbo:coach ?coachName.
            }
WHERE    {     ?person rdf:type dbo:BadmintonPlayer.
               ?person dbp:name ?birthName.
               ?person dbo:weight ?weight.
               ?person dbo:birthDate ?birthDate.
               OPTIONAL{ ?person dbo:coach ?coachName.}
               ?person dbo:birthPlace ?country.
               ?country rdf:type dbo:Country.
               FILTER ( ?country = <http://dbpedia.org/resource/Indonesia>.)
          }
LIMIT 10
```

Abbildung 24: Die neue von *db2seco* generierte SPARQL-Abfrage

4 Implementierung

Im vorherigen Kapitel wurden Techniken skizziert, mit denen Daten aus Linked Open Data Cloud geliefert und mittels den Methoden den SeCo-Algorithmen neue Regel extrahiert wird. Beruhend auf die Techniken wurde das Tool `db2seco` in Java 7 programmiert. Es benutzt die Jena version 2.9.4 und das *SeCo-Framework* der TU-Darmstadt (siehe Abschnitt 2.4.3). *db2seco* lässt sich auf zwei Arten benutzen. Eine Kommandozeilenversion, die auch zum Testen benutzt wird, und die Version mit GUI zur interaktiven Benutzung.

4.1 Überblick

Die übliche Benutzung der GUI-Version wird in Abbildung 25 illustriert. Der Benutzer sieht zuerst das Startfenster (s. Abbildung 26) Nachdem er eine SPARQL-Abfrage eingegeben hat, wird sie an *Jena* weitergeleitet, das ein RDF-Modell zurückliefert (Quelltext 2). Dabei wird für jedes Prädikat ein Datentyp festgelegt, wie in Kapitel 3.4 beschrieben.

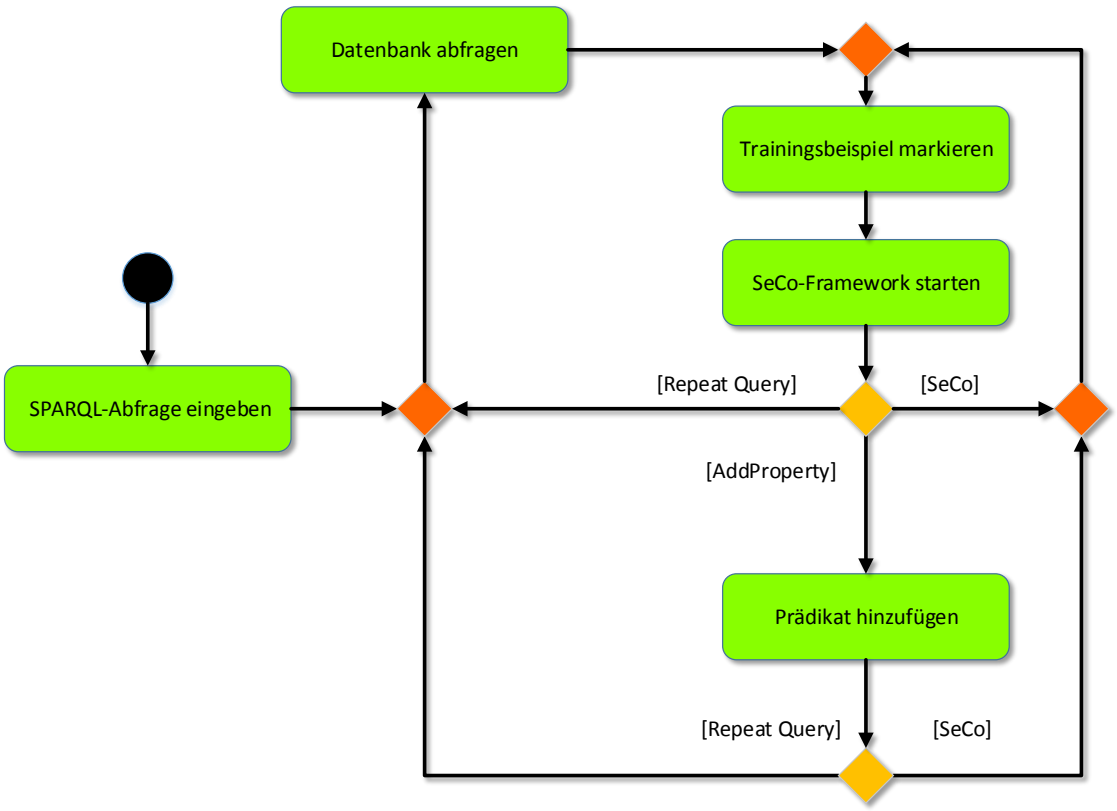


Abbildung 25: Aktivitätsdiagramm des Tools *db2seco*

```java
public static Model queryDataSet(ApplicationModel applicationModel) {
    Model tmpModel = null;
    QueryExecution qexec = null;
    try {
        Query q = QueryFactory.create(applicationModel.getQueryTxt());
        qexec = QueryExecutionFactory.sparqlService(
                applicationModel.getUrlService(), q);
        tmpModel = qexec.execConstruct();
    } finally {
        if (qexec != null) {
            qexec.close();
        }
    }
    ApplicationUtils.findPropertyDatatypes(tmpModel, applicationModel);
    return tmpModel;
}
```

Quelltext 2: SPARQL-Abfrage mit Jena

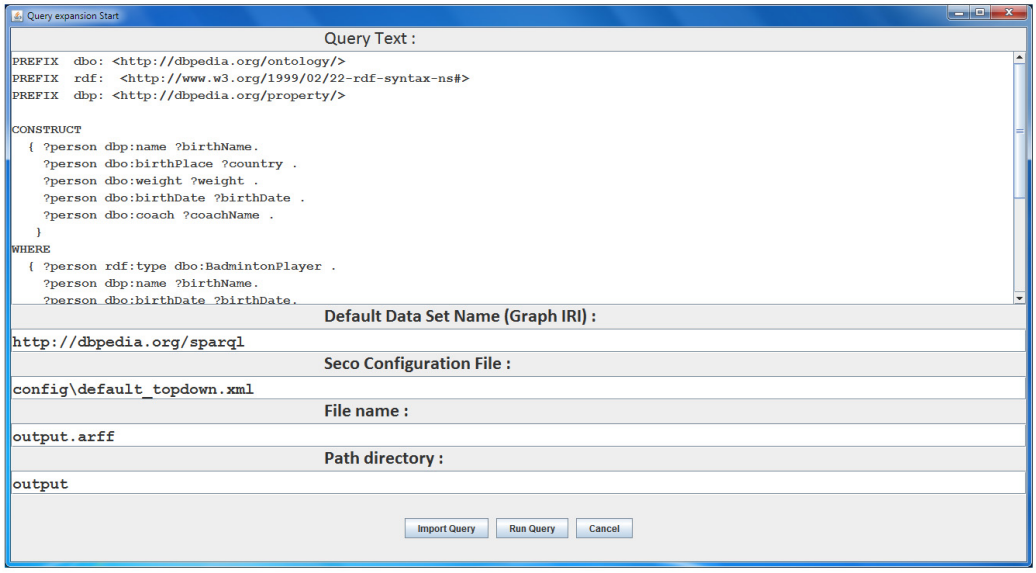


Abbildung 26: Startfenster der GUI

Die RDF-Daten werden nun zu einer Tabelle aufbereitet. Dazu können die von Jena angebotenen Methoden benutzt werden, wie in Quelltext 3 zu sehen. Die Methode `getAllProperties` finde alle Prädikate im RDF-Modell. Daraus wird die Kopfzeile der Tabelle erzeugt. Um den Rest der Tabelle zu erzeugen, werden alle Instanzen des Modells abgefragt und fehlendes Werte als “?” eingetragen, entsprechend dem Format für fehlende Einträge in ARFF.

```
public JTable updateTrainingTable(Model model,
    LinkedHashMap<Resource, Boolean> instanceSelected) {
    Vector<String> tableHeader = new Vector<String>();
    Vector<Object> tableContent = new Vector<Object>();
    ArrayList<Property> properties = getAllProperties(model);
    // Make header of Table
    tableHeader.add(ApplicationUtils.checkBoxRowName);
    for (int i = 0; i < properties.size(); i++) {
        tableHeader.add(properties.get(i).toString());
    }
    // Get all instances in Model and create Row for Table of Checkboxs
    ResIterator instances = model.listSubjects();
    while (instances.hasNext()) {
        Resource current_Resource = instances.next();
        String cell;
        Vector<Object> row = new Vector<Object>();
        row.add(new Boolean(instanceSelected.get(current_Resource)));
        for (int i = 0; i < properties.size(); i++) {
        if (model.contains(current_Resource, properties.get(i))) {
            Property property = properties.get(i);
            Statement statement = current_Resource.getProperty(property);
            cell = getObjectToValue(statement.getObject(), false);
        } else {
            cell = "?";
        }
        row.add(cell);
        }
        tableContent.add(row);
    }
    JTable table = new JTable(tableContent, tableHeader) {
        @Override
        public boolean isCellEditable(int row, int col) {
            if (super.getColumnName(col) == "positive example")
                return true;
                return false;
            }
        };
        table.getColumn("positive example").setCellRenderer(
                table.getDefaultRenderer(Boolean.class));
        table.getColumn("positive example").setCellEditor(
                table.getDefaultEditor(Boolean.class));
    return table;
}
```

Quelltext 3: Erzeugen einer Trainingsmenge

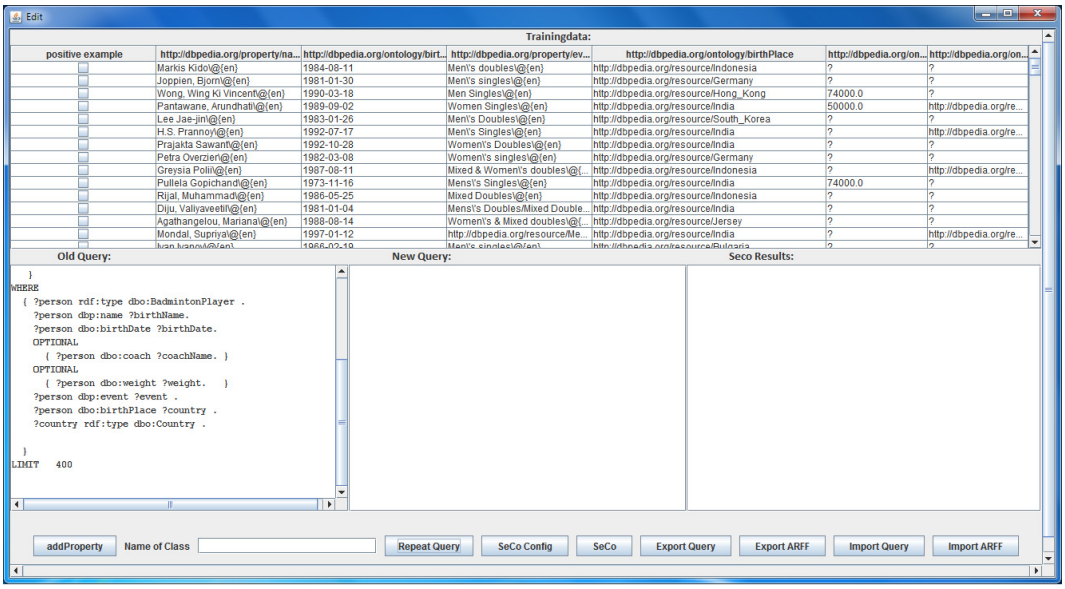


Abbildung 27: Editfenster der GUI

Der Benutzer markiert nun die positiven Beispiele (Abbildung 28) und startet dann das Lernen. Daraufhin wird eine ARFF-Datei geschrieben und zur Verarbeitung an das *SeCo*-Tool übergeben, das eine Regelmenge lernt. Mit Hilfe der Regelmenge werden die Instanzen im Hauptfenster der GUI eingefärbt (Abbildung 29). Richtig gelernte Einträge erscheinen schwarz (**true positive**) oder blau (*true negative*). Falsch gelernte Einträge erscheinen lila (*false negative*) oder rot (*false positive*). Außerdem wird eine neue SPARQL-Abfrage erzeugt und angezeigt.

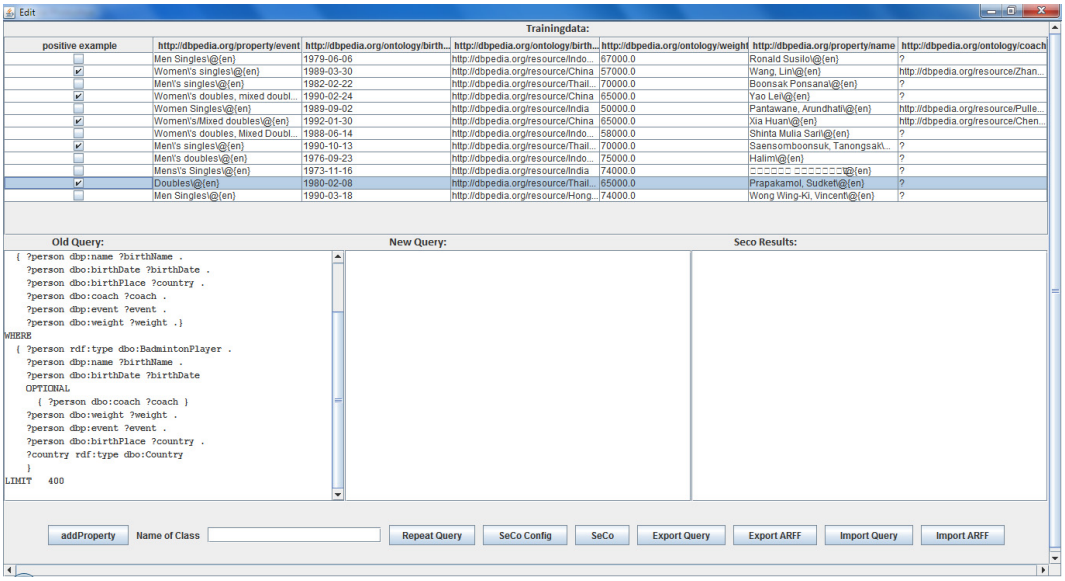


Abbildung 28: Editfenster der GUI mit Markierungen

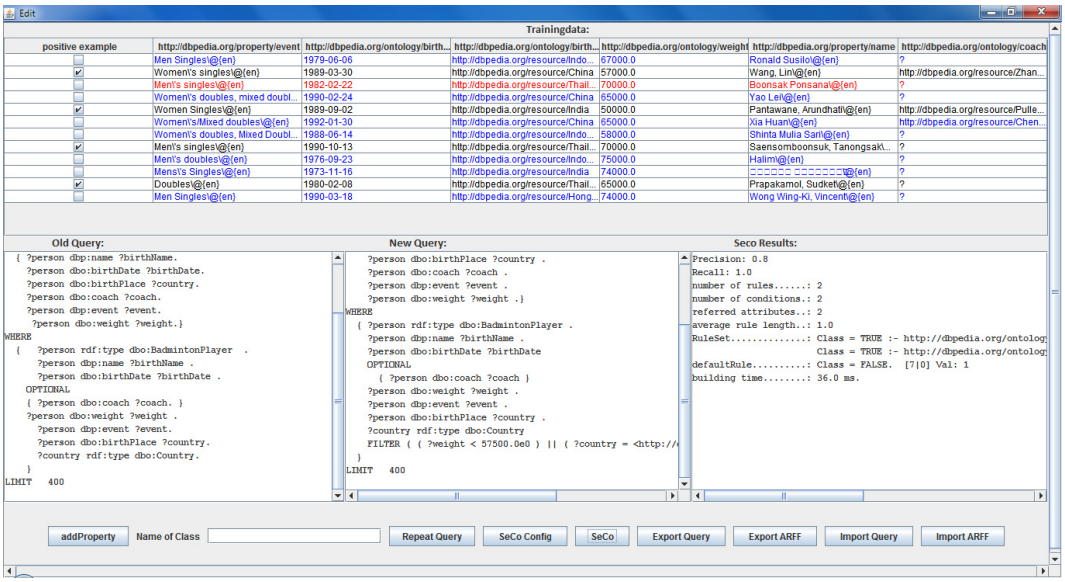


Abbildung 29: Editfenster der GUI nach SeCo-Lauf

Der Benutzer kann nun seine Auswahl ändern und neu lernen. Außerdem kann er die von *db2seco* neu gelernte Abfrage starten und mit ihr weiter arbeiten. Schließlich kann er mit “Add Property” ein Prädikat zum RDF-Modell hinzufügen. Das Prädikat wird entsprechend der aktuellen Regelmenge erzeugt: als *positive* klassifizierte Beispiele erhalten den Wert TRUE, die anderen den Wert FALSE.

4.2 Anpassung der Abfrage

Um die SPARQL-Query anzupassen, muss die vom *seco-Tool* gelernte Regelmenge in ein FILTER-Pattern umgesetzt werden, wie in Kapitel 3.6 beschrieben. Dazu wird aus der alten SPARQL-Abfrage ein Jena-`Query`-Objekt erzeugt und diese Abfrage dann erweitert. Die vom *SeCo*-Tool gelernten Regeln lassen sich aber nicht direkt dafür verwenden, den sie beziehen sich auf die Attribute in der von *db2seco* erzeugten ARFF-Datei. Diese Attribute haben die gleichen Namen wie Prädikate im RDF-Modell, und diese stammen aus der SPARQL-Abfrage.

Deshalb enthält *db2seco* eine Klasse `QueryAdapter` zum Verändern der SPARQL-Abfrage, die in den Quelltexten 4 und 5 gezeigt werden. Es wird dabei ein FILTER-Pattern in KNF-Form generiert, indem alle generierte FILTER-Expressions in einem FILTER-Pattern mit `E_LogicalOr` verbunden werden. Dieses PATTERN wird an die bestehende Abfrage angefügt. Da jede Regel aus mehreren Bedingungen besteht kann, werden die einzelnen Bedingungen (Conditions) als *Subexpression* generiert und in Form einer Konjunktion mittels `E_LogicalAnd` verbunden. Für jede *Subexpression* benötigt *QueryAdapter* das Attribut, die Operation und den Wert des Attributs. Um das Attribut zu bestimmen, wird im Graphpattern nach einem Triple gesucht, dessen Prädikat den gleichen Wert hat wie das Attribut der Bedingung, um die Objektvariable aus diesem Tripel zu holen. Daraus wird ein *ObjectNode* generiert, der als das Attribut der *Subexpression* dient. Die Operation der *Subexpression* wird aus der Art der Bedingung bestimmt. Für den Wert des Attributs schließlich wird aus dem Attributwert der Bedingung ein `nodeValue` erzeugt.

Die Methode `findContainingTriple` durchsucht rekursiv das Graphpattern (den WHERE-Abschnitt) in der der SPARQL-Abfrage nach `ElementPahtBlock`-Elementen. Für jedes Element in einem solchen Block durchsucht er alle Tripel. Wenn bei einem Triple der Prädikat mit dem Attributnamen aus der Bedingung übereinstimmt, wird das entsprechende Tripel zurück gegeben.

Das *SeCo-Tool* unterstützt die Datentypen *numeric* und *nominal* für numerische und symbolische Werte. Rule generation mit regulären Ausdrücken für Strings wird nicht unterstützt. Deshalb werden String-Werte zur nominalen Werten konvertiert. Da für *nominal* nur Gleichungen als Bedingungen gelernt werden können, ist in *db2seco* eine automatische Konversion von Kalenderdaten zu *numeric* implementiert. Dafür wird mit der Java-internen `Date`-Klasse das Datum in einen double-Wert konvertiert, und der Typ des entsprechenden Feldes auf *numeric* gesetzt. Dadurch kann *db2seco* Ungleichungen über Datumswerte lernen.

```
public String appendRuleSetToQuery(RuleSet<Rule> rules, Query query ) {
    ElementGroup mainGroup = (ElementGroup) query.getQueryPattern();
    Expr expression = null;
    for (Rule rule : rules) {
        if (expression == null)
            expression = handleRule(mainGroup, rule);
        else {
            Expr nextRule = handleRule(mainGroup, rule);
            expression = new E_LogicalOr(expression, nextRule);
        }
    }
    ElementFilter newFilter = new ElementFilter(expression);
    mainGroup.addElementFilter(newFilter);
    return query.toString();
}

public Expr handleRule(ElementGroup pattern, Rule rule) {
    Expr expression = null;
    for (Condition condition : rule.getBody()) {
        if (expression == null)
            expression = createFilterSubexpression(pattern, condition);
        else {
            Expr nextExpression = createFilterSubexpression(pattern, condition);
            expression = new E_LogicalAnd(expression, nextExpression);
        }
    }
    return expression;
}

public Expr createFilterSubexpression(ElementGroup pattern, Condition condition) {
    String attribute = condition.getAttr().name();
    TriplePath originalDef = findContainingTriple(pattern,attribute);
    Node objectOfTriple = originalDef.getObject();
    ExprVar objectVariableNode = new ExprVar(objectOfTriple);
    NodeValue conditionNode;
    Expr expression;
    if (condition instanceof NominalCondition) {
        conditionNode = createFilterConditionNodeNominal(condition);
        expression = new E_Equals(objectVariableNode, conditionNode);
    } else {
        expression = compareExpressionForFilter(condition, objectVariableNode);
    }
    return expression;
}
```

Quelltext 4: Erzeugen eines FILTER-Patterns

```
1  private TriplePath findContainingTriple(ElementGroup oldPattern,
2              String attribute_name) {
3      List<Element> elements = oldPattern.getElements();
4      TriplePath found = null;
5      for (Element element: elements) {
6          if (element instanceof ElementPathBlock)
7              found = findContainingTriple((ElementPathBlock) element,
                   attribute_name);
8          else if (element instanceof ElementUnion) {
9              found = findContainingTriple((ElementUnion) element, attribute_name);
10         } else if (element instanceof ElementOptional) {
11             found = findContainingTriple((ElementOptional) element,
                   attribute_name);
12         } else if (element instanceof ElementMinus) {
13             found = findContainingTriple((ElementMinus) element, attribute_name);
14         }
15         if (found != null)
16             return found;
17     }
18     return null;
19 }
20
21 private TriplePath findContainingTriple(ElementPathBlock element,
22             String attribute_name) {
23     Iterator<TriplePath> e = ((ElementPathBlock) element).patternElts();
24     while (e.hasNext()) {
25         TriplePath next = e.next();
26         Node predicate = next.getPredicate();
27         String predicateName = predicate.toString();
28         if (predicateName.equals(attribute_name)) {
29             return next;
30         }
31         predicateName.hashCode();
32     }
33     return null;
34 }
```

Quelltext 5: Suche im Graphpattern einer SPARQL-Abfrage nach Attributname

4.3 Tests

Um die Problemstellung und entworfene Lösung zu testen, hat *db2seco* die Möglichkeit, ohne GUI zu starten.. Damit ist es möglich, die Teilfunktionen des Tools einzeln aufzurufen. Man kann eine ARFF-Datei mittels einer SPARQL-Abfrage generieren und/oder den SeCo-Regellerner mit einer ARFF-Datei starten.

Das Tool *db2seco* unterstützt das Abfrageformat *CONSTRUCT* und das Abfrageergebnis *RDF-Graph*. Alle Test wurden in Form *CONSTRUCT*-Abfragen geschrieben. Sowohl die einfachen als auch komplexe Graphpattern werden von *db2seco* unterstützt. Die Tests benutzen die folgenden Sprachkonstrukte von SPARQL:

testLearnRule Lernen nur einer SPARQL-Abfrage aus einer Datei.

testWithoutQuery Lernen mittels einer nicht markierten ARFF-Datei.

testlearnPlayersFromWales Lernen einer einfachen Regel mit nominalem Wert mittels einer vorher markierten ARFF-Datei.

testDefault Lernen aus einem einfachen Graph-Pattern mit Modifikatoren.

testPerson Lernen aus einem Graph-Pattern mit *OPTIONAL*.

testKorea Lernen aus einem komplexen Graph-Pattern mit *UNION, OPTIONAL* und *FILTER*.

testWithLanguage Lernen aus Literalfeldern mit Sprachangabe.

testUnitedKingdom Lernen mit speziellen Datentypen.

testJessica Lernen mit inkonsistenten Datentypen (vgl. Abschnitt 3.4.2).

5 Evaluation

Das Tool `db2seco` lernt mit verschieden SeCo-Algorithmen Klassen. Dabei muss der Nutzer einen Lehralgorithmus auswählen. Deshalb werden in diesem Kapitel die Ergebnisse der Anwendung von verschiedenen Klassifikatoren auf Linked Open Data vorgestellt und diskutiert. Diese Untersuchen beschäftigt sich damit, wie gut die Lernalgorithmen auf Daten von Linked Open Data funktionieren. Dazu werden die Ergebnisse von fünf *SPARQL-Abfragen* untersucht. Es soll untersucht werden, ob es in manchen Fällen zu Überanpassung kommt. Weiter wird betrachtet wie hoch die Komplexität der vom SeCo-Framework gelernten Abfragen ist.

Für die Evaluation wurden verschiedene *SeCo-Algorithmen* betrachtet. Konkret wurden die Konfigurationen `Accuracy`, `Precision`, `Recall`, `F-Measure`, `Ripper`, `Weighted` verwendet. Sie unterscheiden sich in ihrer Funktionsweise und wurden jeweils in der Praxis für verschiedene Anwendungen verwendet. Alle Klassifikatoren wurden bei der Evaluation in ihrer *Weka* Standardkonfiguration eingesetzt.

5.1 Evaluationsmethoden

Jedes Lernverfahren erzeugt Regeln, um Klassen zu beschreiben. Mit einer Metrik schätzt man die Güte einer Regel. Die meisten Metriken schätzen die Güte einer Regel anhand der positiven und negativen Beispiele, die sie abdeckt. In dieser Arbeit werden die *Precision*, *Recall*, *F-Measure* und *Accuracy* einer gefundene Regelmenge ausgerechnet, die hier genauer erklärt werden. Damit Metriken aussagekräftig sind, müssen sie auf unterschiedlichen Testdaten berechnet werden. Um Daten effizient zu verwenden, wird 10-fache *cross validation* angewandt, die anschließend erklärt wird.

5.1.1 Metriken

Positives Beispiel: Ein Beispiel gilt dann als ein positives Beispiel, wenn die Klasse des Beispiel mit der gesuchten Klasse übereinstimmt. Zum Beispiel das Beispiel aus Tabelle 2:

```
birthName = 'Marc Zwiebler', gender = 'Male', discipline = 'BadmintonPlayer'
```

Negatives Beispiel: Ein Beispiel gilt dann als ein negatives Beispiel, wenn die Klasse des Beispiel mit der gesuchten Klasse **nicht** übereinstimmt. Zum Beispiel das Beispiel aus Tabelle 2:

```
birthName = 'Teu Seu Bock', gender = 'Male', discipline = 'Coach'
```

- P: Anzahl aller positiven Beispiele
- N: Anzahl aller negativen Beispiele
- r: Die Kandidatenregel
- p: Die durch *r* abgedeckten positiven Beispiele
- n: Die durch *r* abgedeckten negativen Beispiele

Tabelle 4 [21] zeigt alle Variablen, die in dieser Arbeit und im *SeCo-Framework* verwendet wurden:

Tabelle 4: Variablendefinitionen für die Heuristiken

Variable	Beschreibung
P	Anzahl aller positiven Beispiele
N	Anzahl aller positiven Beispiele
r	Die Kandidatenregel
tn	Anzahl der Beispiel, die von *r* korrekterweise negativ klassifiziert sind (*true negatives*)
fn	Anzahl der Beispiel, die von *r* fälschlicherweise negativ klassifiziert sind (*false negatives*)
tp	Anzahl der Beispiel, die von *r* korrekterweise positiv klassifiziert sind (*true positives*)
fp	Anzahl der Beispiel, die von *r* fälschlicherweise positiv klassifiziert sind (*false positives*)

Precision:

ist der Anteil der Richtig-Positiven *(tp)* an der Anzahl aller abgedeckten Instanzen *(tp+fp)*

$$Precision = \frac{tp}{tp+fp}$$

Das Problem bei dieser Metrik ist, dass sie auch dann einen hohen Wert annimmt, wenn wenige `tp` Beispiele gefunden werden, falls es nicht zu viele `fn` gibt.

Recall:

beschreibt den Anteil der positiven Beispiele, die richtig klassifiziert wurden.

$$Recall = \frac{tp}{tp+fn}$$

Diese Metrik ist am höchstens, wenn alle `tp`s gefunden werden, unabhängig davon wie viele `fp`s von einer Regel abgedeckt werden.

F-Measure:

ist als harmonischer Mittelwert von Precision und Recall definiert.

$$F-Measure = 2 \cdot \frac{Precision \cdot Recall}{Precision+Recall}$$

Accuracy:

beschreibt den Anteil der Beispiele, die richtig klassifiziert wurden.

$$Accuracy = \frac{tp+tn}{tp+fp+tn+fn}$$

Diese Metrik ist auch dann hoch, wenn die Regeln die Trainingsmenge genau beschreiben.

5.1.2 Cross validation

Bei cross validation werden die Testdaten in gleich große Teile aufgeteilt. Dann wird jeder Teil einmal als Testmenge verwendet und alle anderen Daten jeweils als Trainingsmenge. Mit diesem Verfahren werden also dieselben Testdaten mehrmals verwendet, so dass insgesamt wenige Testdaten benötigt werden. Die Güte eines Lehrverfahrens berechnet sich dann aus dem Mittelwert der Ergebnisse für die einzelnen Testläufe.

5.2 Testdaten

Die Testdatensätze für die Versuche wurden durch fünf SPARQL-Abfragen gewonnen. Die Abfragen wurden mit *db2seco* an DBpedia gestellt, annotiert, und zum Lernen in ARFF-Dateien gespeichert. Von den Testdatensätzen wurden einige zufällig markiert und andere nach einem einfachen Kriterium.

Die Testdatensätze mit Namen *PlacesWithPopulation* wurde zufällig markiert und unterscheiden sich darin, wie viele Instanzen ausgewählt sind. Sie stammen aus der selben Query. Bei *QueryPersonUnder80KG* wurden die Einträge von Personen mit einem Wert für *weight* unter 80 annotiert. Bei *BadmintonPlayerIndonesia* wurde aus einer Abfrage von Bad-

mintonspielern diejenigen mit *BirthPlace* Indonesien markiert. Bei *AthleteUnder30* wurden Sportler mit einem *BirthDate* nach 1985 markiert. Von den letzten beiden Datensätzen gibt es jeweils eine Variation (*Random*), in der Datensätze zufällig markiert sind.

Tabelle 5: Die einzelnen verwendeten *Testdatensätze* mit ihren Merkmalen

Name	*#Instanzen*	*#Attribute(nominal)*	*#Attribute(numerisch)*	*Markierte Beispiele(%)*
PersonenUnder80KG	200	3	1	17.5
AthleteUnder30	200	3	2	32.5
BadmintonPlayerIndonesia	174	4	2	13.21
BadmintonPlayerIndonesia(Random)	174	4	2	13.21
AthleteUnder30(Random)	200	3	2	32.5
PlacesWithPopulation (10%)	1000	3	4	10
PlacesWithPopulation (20%)	1000	3	4	20
PlacesWithPopulation (40%)	1000	3	4	40

Bei den Testdaten fehlen häufig Attributwerte, wie in Abschnitt 3 beschrieben. Das *SeCo-Tool* ignoriert diese Werte beim Lernen der Regelmenge. Die einzelnen Regeln beziehen sich niemals darauf, ob ein Feld einen Wert hat.

5.3 Ergebnisse

Abbildungen 6 bis 13 zeigen die Ergebnisse der 10-fachen cross validation. Für jeden Testdatensatz sind zuerst die Metriken zur Bewertung des Qualität der Regelmenge aufgelistet und zur Dauer der cross validation. Rechts davon stehen Angaben zur Anzahl der Regeln und Bedingungen beim Lernen mit dem ganzen Datensatz. Der Eintrag NaN bezeichnet ein Feld ohne gültigen Wert (mit Nenner 0 im Bruch). Die Zeile mit Sternen (*) in Tabelle 11 macht kenntlich, dass durch diese Konfiguration bei diesem Testdatensaz niemals eine Regel gelernt wurde.

Tabelle 6: *PersonenUnder80KG*

Klassifikator	*Precision*	*Recall*	*F-Measure*	*Accuracy*	*Laufzeit (sec)*	*# Regeln*	*# Bedingungen*
Accuracy	1.0	1.0	1.0	1.0	0.75	1	1
Ripper	1.0	1.0	1.0	1.0	0.19	1	1
Precision	1.0	1.0	1.0	1.0	0.19	1	1
F-Measure	1.0	1.0	1.0	1.0	0.23	1	1
WRA	1.0	1.0	1.0	1.0	0.19	1	1

Tabelle 7: *AthleteUnder30*

Klassifikator	*Precision*	*Recall*	*F-Measure*	*Accuracy*	*Laufzeit (sec)*	*# Regeln*	*# Bedingungen*
Accuracy	0.98	0.98	0.98	0.99	0.35	1	1
Ripper	0.98	0.98	0.98	0.99	0.15	1	1
Precision	0.98	0.98	0.98	0.99	0.17	1	1
F-Measure	0.98	0.98	0.98	0.99	0.13	1	1
WRA	0.98	0.98	0.98	0.99	0.16	1	1

Tabelle 8: *BadmintonPlayerIndonesia*

Klassifikator	*Precision*	*Recall*	*F-Measure*	*Accuracy*	*Laufzeit (sec)*	*# Regeln*	*# Bedingungen*
Accuracy	1.0	1.0	1.0	1.0	0.19	1	1
Ripper	1.0	1.0	1.0	1.0	0.09	1	1
Precision	1.0	1.0	1.0	1.0	0.01	1	1
F-Measure	1.0	1.0	1.0	1.0	0.1	1	1
WRA	1.0	1.0	1.0	1.0	0.07	1	1

Tabelle 9: *BadmintonPlayerIndonesia (Random)*

Klassifikator	*Precision*	*Recall*	*F-Measure*	*Accuracy*	*Laufzeit (sec)*	*# Regeln*	*# Bedingungen*
Accuracy	0.0	0.0	NaN	0.86	1.9	22	23
Ripper	0.0	0.0	NaN	0.83	0.33	0	0
Precision	0.0	0.0	NaN	0.87	0.83	23	23
F-Measure	0.0	0.0	NaN	0.84	0.5	1	2
WRA	NaN	0.0	NaN	0.87	0.80	0	0

Tabelle 10: *AtheleteUnder30 (Random)*

Klassifikator	*Precision*	*Recall*	*F-Measure*	*Accuracy*	*Laufzeit (sec)*	*# Regeln*	*# Bedingungen*
Accuracy	0.16	0.05	0.07	0.61	3.38	51	55
Ripper	0.17	0.06	0.09	0.60	0.65	0	0
Precision	0.0	0.0	NaN	0.66	3.45	63	63
F-Measure	0.09	0.02	0.03	0.63	0.6	0	0
WRA	0.10	0.02	0.03	0.64	0.55	0	0

Tabelle 11: *PlacesWithPopulation (10%)*

Klassifikator	*Precision*	*Recall*	*F-Measure*	*Accuracy*	*Laufzeit (sec)*	*# Regeln*	*# Bedingungen*
Accuracy	0.0	0.0	NaN	0.89	18.26	100	102
Ripper	0.06	0.01	0.02	0.89	3.5	0	0
Precision	0.0	0.0	NaN	0.90	21	100	102
F-Measure	0.0	0.0	NaN	0.90	3.56	0	0
WRA	*	*	*	*	22.20	0	0

Tabelle 12: *PlacesWithPopulation (20%)*

Klassifikator	*Precision*	*Recall*	*F-Measure*	*Accuracy*	*Laufzeit (sec)*	*# Regeln*	*# Bedingungen*
Accuracy	0.3	0.03	0.05	0.80	56.21	188	195
Ripper	0.25	0.02	0.03	0.80	7.70	4	13
Precision	0.26	0.005	0.01	0.80	74.03	199	202
F-Measure	NaN	0.0	NaN	0.8	72.38	0	0
WRA	NaN	0.0	NaN	0.8	111.47	0	0

Tabelle 13: *PlacesWithPopulation (40%)*

Klassifikator	*Precision*	*Recall*	*F-Measure*	*Accuracy*	*Laufzeit (sec)*	*# Regeln*	*# Bedingungen*
Accuracy	0.45	0.13	0.20	0.59	170.90	326	354
Ripper	0.40	0.08	0.13	0.59	10.26	3	13
Precision	0.52	0.03	0.06	0.60	257.679	390	398
F-Measure	NaN	0.0	NaN	0.6	9767.67	0	0
WRA	0.41	0.12	0.19	0.58	237.52	1	8

Von den manuell markierten Testdatensätzen (Abbildungen 6, 7 und 8) wird nur *AthleteUnder30* nicht perfekt gelernt. Das Attribut *birthDate* ist numerisch, und numerische Attribute werden vom *SeCo-Tool* nicht genau gelernt, um Überanpassung zu vermeiden. Deshalb können beim Lernen kleine Fehler wie dieser auftreten.

Bei den kleinen, zufällig markierten Testdatensätzenn lässt sich *AthleteUnder30Random* gut lernen, *BadmintonPlayerIndonesia* dagegen gar nicht. Offenbar entscheiden sich alle Heuristiken dafür, die Instanzen einzeln mit sehr präzisen Regeln zu lernen. Auf dem vollständigen Testdatensatz finden die Konfigurationen *Accuracy* and *Precision* in beiden Fällen

eine Regelmenge, diese entsprechen aber nur einer Aufzählung der positiven Beispiele. Die Laufzeit bei den erfolgreichen Testabläufen (bei denen etwas gelernt wurde) ist für große Regelmengen meistens höher als für kleine Regelmengen.

Bei den Testdatensätzen aus Tabellen 11 bis 13 mit 1000 Einträgen nimmt die Laufzeit der cross evaluation mit der Anzahl positiver Beispiele stark zu. Im Extremfall dauert die Berechnung für *F-Measure* mehr als 2 Stunden und 42 Minuten. Die Heuristik *WRA* finden für diese Fälle keine nicht-triviale Regelmenge. Wenn man die Größe der Regelmenge unter Berücksichtigung von Precision und Recall aller Heuristiken betrachtet, dann erweist sich *Ripper* als die am besten geeignete Heuristik für zufällige Testdatensätze.

Zusammenfassend lässt sich sagen, dass das Lernen auf den manuell markierten Testdatensätzen sehr gut funktioniert, aber bei zufällig markierten Testdatensätze wie erwartet sehr umfangreiche Regelmengen entstehen. Die Länge der FILTER-Pattern bei den gelernten SPARQL-Abfragen entspricht genau der Anzahl gelernter Bedingungen in der Regelmenge. Das gleiche gilt für die Länge der gelernten SPARQL-Abfragen insgesamt, da sie beim Anpassen durch `db2seco` nur um das FILTER-Pattern erweitert werden. Für die Heuristiken Accuracy und Precision entstehen sehr lange SPARQL-Abfragen, für die anderen aber kurze.

Einige andere Arbeiten beschäftigen sich mit der Integration von Semantic Web-Datensätzen mit automatischen Lernalgorithmen und Tools. Ziel kann dabei das Lernen von Anfragen sein oder das Finden von Prädikaten.

Bei AutoSPARQL [28] ist es möglich, SPARQL-Abfragen mit Hilfe von Beispielen zu lernen. Der Benutzer gibt zuerst eine eigene SPARQL-Abfrage ein. Dann benutzt er eine graphische Oberfläche, die ihm immer wieder Vorschläge macht, die er als positive oder negative Beispiele klassifiziert. Die ursprüngliche Abfrage und die Queries werden dann auf Query Trees abgebildet. Dann können alle Trees zu einem Verbunden werden, der wiederum zu einer SPARQL-Abfrage umgesetzt werden. Wie *db2seco* lernt AutoSPARQL aus einer SPARQL-Anfrage mit Hilfe von Benutzereingaben eine neue Anfrage. Bei *db2seco* benutzt man dabei viele Beispiele auf einmal, und es wird ein sehr einfacher *Filter* gelernt, wenn es genug Beispiele gegeben hat.

Das Tool *FeGeLOD* [29] arbeitet auf bestehenden RDF-Graphen und erzeugt automatisch neue Prädikate und wendet data mining auf das Ergebnis an. Dabei werden verschiedene Algorithmen benutzt, um die Prädikate zu erzeugen. Es zeigt sich dabei, dass keiner der Algorithmen am Besten ist. In unserem Ansatz wählt der Benutzer selbst aus, welche Prädikate er lernen möchte, und erweitert damit die Datensätze.

Das Tool Rapidminer [30] integriert Semantic Web Abfragen in das data mining-tool rapid miner. Der Benutzer kann als zu lernende Daten einen RDF-Graphen angeben, dessen Daten dann verlinkt werden können. Linked open data kann dadurch zusammen mit anderen Daten verwendet werden, um Analysen vorzunehmen. Bei *db2seco* können adaptierte SPARQL-Queries gelernt werden, die ein Prädikat beschreiben, und diese Queries dann neu gesendet werden. Außerdem kann ein gelerntes Prädikat zur ARFF-Datei hinzugefügt werden.

7 Zusammenfassung und Ausblick

Linked open data bietet große Datenbestände an, die das Semantic Web erweitern. Diese Daten sind von unterschiedlicher Qualität. Manche von ihnen können direkt in weiter verarbeitet werden, andere müssen dagegen selbst verändert werden, bevor sie sich für automatisierte Weiterverarbeitung eignen.

Für die Arbeit mit linked open data werden deshalb Werkzeuge benötigt, um diese Daten abzufragen und anzupassen. Als grundlegende Technologien sind RDF und SPARQL standardisiert, diese eigenen sich aber nicht unbedingt für die Verwendung durch Menschen, die keine Experten sind. Deshalb werden Programme benötigt, die eine einfachere Schnittstelle zu linked open data-Datensätzen zur Verfügung stellen.

In dieser Arbeit wurde ein Ansatz vorgestellt, bei dem maschinelles Lernen auf linked open data angewandt wird, um linked data abzufragen und zu ergänzen. Dazu wurde das Jena-Framework zur Abfrage von RDF-Graphen mit einem Separate-and-Conquer-Regellerner verknüpft. Das Tool *db2seco* sendet SPARQL-Anfragen an Linked Data-Datenbanken und stellt die Ergebnisse dem Benutzer zur Verfügung. Mit diese Daten kann der Benutzer seine Auswahl verfeinern und so neue SPARQL-Anfragen lernen, und er kann neue Triples zum RDF-Graphen hinzufügen.

Das vom *seco*-Tool benutzte Datenformat ARFF stellt andere Anforderungen an das Format der Daten als linked open data. Es unterstützt nur einen einzelnen Wert für ein Attribut und benötigt für jedes Attribut eine Datentypdeklaration. *db2seco* macht deshalb an einigen Stellen Anpassungen an die Ergebnisse einer SPARQL-Abfrage, so dass diese Bedingungen erfüllt sind.

Die Evaluation hat gezeigt, dass das Lernen eines neuen Attributs auf linked open data gut funktioniert, wenn die Daten sinnvoll markiert sind. Deshalb ist es wichtig, das Programme wie *db2seco* bei der Datenaufbereitung helfen.

Es gibt einige Möglichkeiten, den Ansatz von *db2seco* zu erweitern. So könnte man *db2seco* so verändern, dass es alle Attribute zu den von der SPARQL-Anfrage angesprochenen Ressourcen ausgibt, und den Benutzer dann auswählen lassen, welche Werte er betrachten will. Andere Änderungen im Umgang mit den Daten sind möglich, zum Beispiel könnte man abgefragte Daten um Prädikate erweitern, die nicht vom Typ *boolean* sind. Dazu ist die hier verwendete Version des SeCo-Tools allerdings nicht geeignet. Außerdem wäre es möglich, den Unterschied zwischen linked open data und anderen Daten zu untersuchen, um zu sehen, welche Features *db2seco* noch bekommen sollte, um die Arbeit mit linked open data besser zu unterstützen.

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltexte

Literaturverzeichnis

[1] http://www.w3.org/RDF.

[2] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[3] http://www.w3.org/TR/owl-guide/.

[4] Christian Bizer, Tom Heath, and Tim Berners-Lee. Semantic web and linked data, Juli 2009. http://www.w3.org/2009/Talks/0120-campus-party-tbl/#(14).

[5] Harald Sack. Vorlesung semantic web. 2009. http://www.hpi.uni-potsdam.de.

[6] W. Becker and K. Benz. *Effizienz des Controlling*. Bamberger betriebswirtschaftliche Beiträge. Otto-Friedrich-Univ., 1996.

[7] Andreas Geissel Mario Jeckle Toby Baier, Michael Ebert. Xml schema, Mai 2001. http://www.edition-w3.de/TR/2001/REC-xmlschema-0-20010502/.

[8] O. Lassila und R. Swick. Resource description framework (rdf) model and syntax specification, February 1999. http://www.w3.org/TR/REC-rdf-syntax/.

[9] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph, and York Sure. *Semantic Web: Grundlagen*. Springer-Verlag, 2007.

[10] Michael C Daconta, Leo J Obrst, and Kevin T Smith. The semantic web: a guide to the future of xml, web services, and knowledge management, 2003.

[11] Heiko Paulheim. The semantic web lecture slides. http://www.ke.tu-darmstadt.de/lehre/archiv/ws-12-13/semantic-web/slides/02_RDF_Teil1.pdf/at_download/file.

[12] Shelley Powers. Practical rdf. 2003.

[13] John Breslin, Alexandre Passant, and Stefan Decker. *The social semantic web*. Springer Science & Business Media, 2009.

[14] Gunther Sudra. *Wissensbasierte Situationsinterpretation für eine kontextbezogene Chirurgieassistenz mittels Erweiterter Realität*. KIT Scientific Publishing, 2010.

[15] Gabriele Wichmann. *Entwurf Semantic Web: Entwicklung, Werkzeuge, Sprachen*. VDM Publishing, 2007.

[16] Dave Beckett and Jeen Broekstra. Sparql query results xml format. w3c candidate recommendation. *World Wide Web Consortium (W3C)*, 2013.

[17] Jeremy J Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations, 2004. http://www.hpl.hp.com/techreports/2003/HPL-2003-146.pdf.

[18] Herbert A Simon. Why should machines learn? In *Machine learning*, pages 25–37. Springer, 1983.

[19] Tom M Mitchell. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45, 1997.

[20] Rui Xu, Donald Wunsch, et al. Survey of clustering algorithms. *Neural Networks, IEEE Transactions on*, 16(3):645–678, 2005.

[21] Johannes Fürnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13(1):3–54, 1999.

[22] Giulia Bagallo and David Haussler. Boolean feature discovery in empirical learning. *Machine learning*, 5(1):71–99, 1990.

[23] Ryszard S Michalski. On the quasi-minimal solution of the general covering problem. proceeding of the v international symposium on information processing. pages 125–128, 1969.

[24] Johannes Fürnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13(1):3–54, 1999.

[25] Matthias Thiel. *Separate and conquer framework und disjunktive regeln*. PhD thesis, Master's thesis, TU Darmstadt, 2005. In German (English title: Separate and Conquer Framework and Disjunctive Rules). 5, 2005.

[26] Frederik Janssen and Johannes Fürnkranz. The seco-framework for rule learning. In *Proceedings of the German Workshop on Lernen, Wissen, Adaptivität-LWA*, 2012.

[27] Geoffrey Holmes Bernhard Pfahringer Peter Reutemann Ian H. Mark Hall, Eibe Frank. The weka data mining software, 2009. `http://www.kdd.org/explorations/issues/11-1-2009-07/p2V11n1.pdf`.

[28] Jens Lehmann and Lorenz Bühmann. Autosparql: Let users query your knowledge base. In *The Semantic Web: Research and Applications*, pages 63–79. Springer, 2011.

[29] Heiko Paulheim and Johannes Fümkranz. Unsupervised generation of data mining features from linked open data. In *Proceedings of the 2nd international conference on web intelligence, mining and semantics*, page 31. ACM, 2012.

[30] Mansoor Ahmed Khan, Gunnar Aastrand Grimnes, and Andreas Dengel. Two pre-processing operators for improved learning from semanticweb data. In *First RapidMiner Community Meeting And Conference (RCOMM 2010)*, volume 20, 2010.