NLP Data Cleansing Based on Linguistic Ontology Constraints

Dimitris Kontokostas[1], Martin Brümmer[1], Sebastian Hellmann[1], Jens Lehmann[1], and Lazaros Ioannidis[2]

[1] Universität Leipzig, Institut für Informatik, AKSW, `http://aksw.org`
`{lastname}@informatik.uni-leipzig.de`
[2] Aristotle University of Thessaloniki, Medical Physics Laboratory
`lioannid@math.auth.gr`

Abstract. Linked Data comprises of an unprecedented volume of structured data on the Web and is adopted from an increasing number of domains. However, the varying quality of published data forms a barrier for further adoption, especially for Linked Data consumers. In this paper, we extend a previously developed methodology of Linked Data quality assessment, which is inspired by test-driven software development. Specifically, we enrich it with ontological support and different levels of result reporting and describe how the method is applied in the Natural Language Processing (NLP) area. NLP is – compared to other domains, such as biology – a late Linked Data adopter. However, it has seen a steep rise of activity in the creation of data and ontologies. NLP data quality assessment has become an important need for NLP datasets. In our study, we analysed 11 datasets using the *lemon* and *NIF* vocabularies in 277 test cases and point out common quality issues.

Keywords: #eswc2014Kontokostas, Linked Data, NLP, data quality

1 Introduction

Linked Data (LD) comprises of an unprecedented volume of structured data on the Web and is adopted from an increasing number of domains. However, the varying quality of the published data forms a barrier in further adoption, especially for Linked Data consumers.

Natural Language Processing (NLP) is – compared to other domains, such as Biology – a late LD adopter with a steep rise of activity in the creation of vocabularies, ontologies and data publishing. A plethora of workshops and conferences such as LDL `http://ldl2014.org/`, WoLE `http://wole2013.eurecom.fr`, LREC `http://lrec2014.lrec-conf.org`, MLODE `http://sabre2012.infai.org/mlode`, NLP&DBpedia `http://nlp-dbpedia2013.blogs.aksw.org/program/`) motivate researchers to adopt Linked Data and RDF/OWL and convert traditional data formats such as XML and relational databases. Although guidelines and best practices for this conversion exist, developers from NLP are often unfamiliar with them, resulting in low quality and inoperable data. In this paper, we address the subsequently arising need for data quality assessment of those NLP datasets.

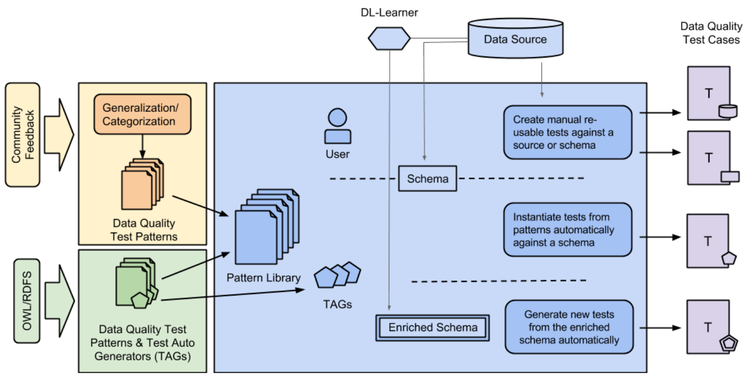

DL-Learner
Data Source
Data Quality Test Cases
Community Feedback
Generalization/ Categorization
Data Quality Test Patterns
User
Create manual re-usable tests against a source or schema
Schema
Instantiate tests from patterns automatically against a schema
Pattern Library
TAGs
OWL/RDFS
Data Quality Test Patterns & Test Auto Generators (TAGs)
Enriched Schema
Generate new tests from the enriched schema automatically
T

Fig. 1. Flowchart showing the test-driven data quality methodology. The left part displays the input sources of our pattern library. In the middle part the different ways of pattern instantiation are shown which lead to the Data Quality Test Cases on the right.

We extended a recently introduced test-driven data quality methodology [12] inspired by tests in software engineering. In its introduction, the methodology in [12] focused on two approaches: (1) automatically-generated test cases which were derived from the OWL/RDFS schema of the ontologies and (2) test cases adhering to patterns from a pattern library. These two approaches were evaluated on popular ontologies and data sets such as FOAF or the DBpedia Ontology and it was shown that the methodology is well suited for horizontal, multi-domain data quality assessment, i.e. massive detection of errors for five large-scale LOD data sets as well as on 291 vocabularies, independent of their domain or their purpose. In this paper, we will briefly introduce the methodology in Section 2 including a comparison of our methodology to `OWL` reasoning. The *Test Driven Data Engineering Ontology* is described in Section 3. Using the ontology, we can annotate test cases and provide support for different levels of result reporting allowing to give feedback to developers when running these tests and ultimately improving data quality.

Additionally, we show progress in implementing domain-specific validation by quickly improving existing validation provided by ontology maintainers. We specifically analysed datasets for two emerging domain ontologies, the *lemon model* [13] and the *NIF 2.0 Core Ontology* [10] in Section 4 and evaluated 11 datasets in Section 5.

2 Overview of Test-Driven Data Assessment Methodology

In this section we introduce basic notions of our methodology. A thorough description of test-driven quality assessment methodology can be found in [12]

Data Quality Test Pattern (DQTP). A data quality test pattern is a SPARQL query template with variable placeholders. Possible types of the pattern variables are IRIs, literals, operators, datatype values (e.g. integers) and regular expressions. Using `%%v%%` as syntax for placeholders, an example DQTP is:

```
SELECT ?s WHERE { ?s %%P1%% ?v1. ?s  %%P2%%   ?v2 .
                  FILTER ( ?v1 %%OP%% ?v2 ) }
```

This DQTP can be used for testing whether a value comparison of two properties $P1$ and $P2$ holds with respect to an operator OP. DQTPs represent abstract patterns, which can be further refined into concrete data quality test cases using test pattern bindings.

Test Pattern Binding. Test pattern bindings are valid DQTP variable replacements.

Data Quality Test Case. Applying a pattern binding to a DQTP results in an executable SPARQL query. Each result of the query is considered to be a violation of a test case. A test case may have four different results: success (empty result), violation (results are returned), timeout (test is marked for further inspection) and error (something prevented the query execution). An example test pattern binding and resulting data quality test case is[3]:

```
P1  =>  dbo:birthDate |   SELECT ?s WHERE {
P2  =>  dbo:deathDate |    ?s  dbo:birthDate  ?v1.
OP  =>  >             |    ?s  dbo:deathDate  ?v2.
                      |    FILTER ( ?v1 > ?v2 ) }
```

Test Auto Generator (TAG). A Test Auto Generator reuses the RDFS and OWL modelling of a knowledge base to verify data quality. In particular, a TAG, based on a DQTP, takes a schema as input and returns test cases. TAGs consist of a detection and an execution part. The detection part is a query against a schema and for every result of a detection query, a test case is instantiated from the respective pattern, for instance:

```
# TAG                              | # TQDP
SELECT DISTINCT ?P1 ?P2            | SELECT DISTINCT
WHERE {                            | ?s WHERE {
?P1 owl:propertyDisjointWith ?P2.| ?s %%P1%% ?v.
}                                  | ?s %%P2%% ?v.}
```

Additionally, we devise the notion of RDF test case coverage based on a combination of six individual coverage metrics [12].

The test-driven data quality methodology is illustrated in Figure 1. As shown in the figure, there are two major sources for the creation of tests. One source is stakeholder feedback from everyone involved in the usage of a dataset and the other source is the already existing `RDFS/OWL` schema of a dataset. Based on this, there are several ways to create tests:

1. *Manually create test cases:* Test cases specific to a certain dataset or schema can be written manually. This can be guided choosing suitable DQTPs of

[3] We use `http://prefix.cc` to resolve all name spaces and prefixes. A full list can be found at `http://prefix.cc/popular/all`

our pattern library. Tests that refer to the schema of a common vocabulary can become part of a central library to facilitate later reuse.
2. *Reusing tests based on common vocabularies:* Naturally, a major goal in the Semantic Web is to reuse existing vocabularies instead of creating new ones. We detect the used vocabularies in a dataset, which allows to re-use tests from a test pattern library.
3. *Using `RDFS/OWL` constraints directly:* As previously explained, tests can be automatically created via TAGs in this case.
4. *Enriching the `RDFS/OWL` constraints:* Since many datasets provide only limited schema information, we perform automatic schema enrichment as recently researched in [4]. Those schema enrichment methods can take an `RDF` dataset or a SPARQL endpoint as input and automatically suggest schema axioms with a certain confidence value by analysing the dataset. In our methodology, this is used to create further tests via TAGs.

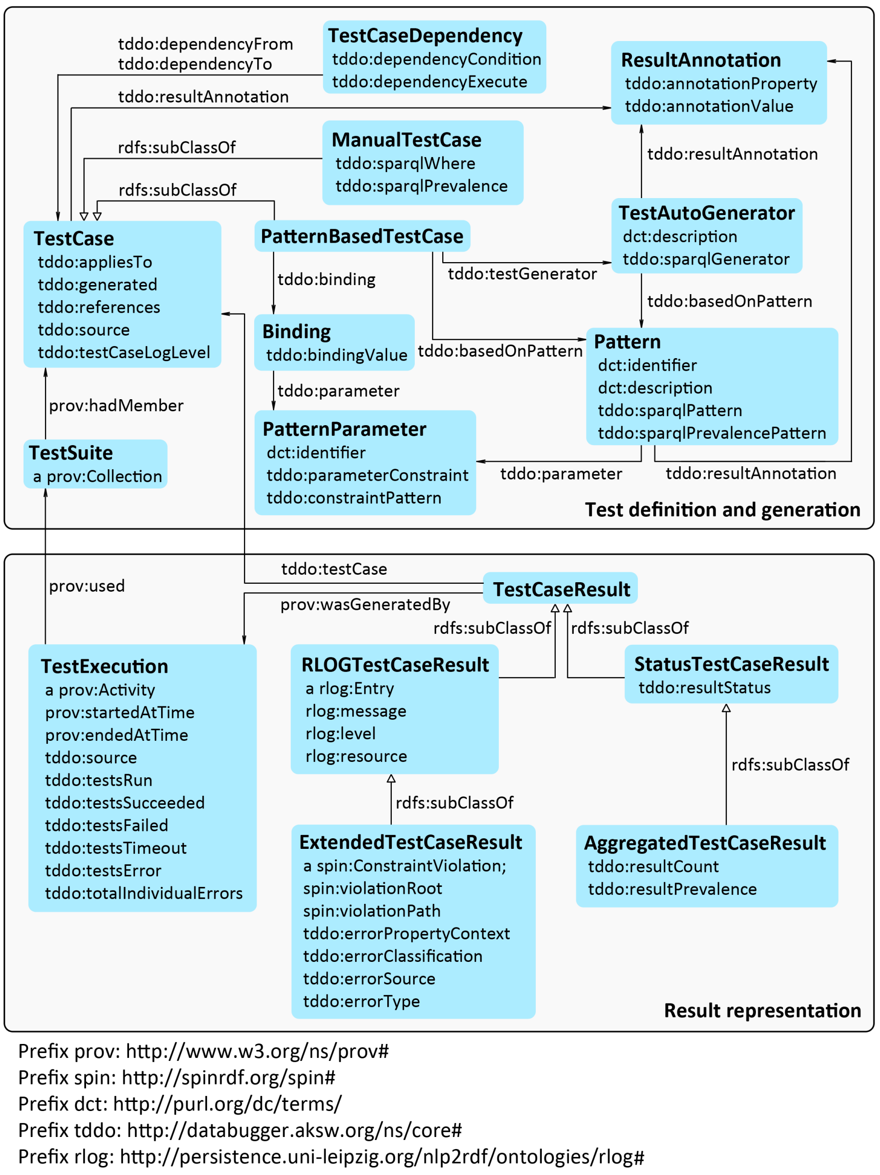
The *RDFUnit* Suite [4] [5] implements the test driven data assessment methodology The methodology is implemented in a Java component and released as open source under the Apache licence.

Relation Between SPARQL Test Cases and OWL Reasoning. SPARQL test cases can detect a subset of common validation errors detectable by a sound and complete `OWL` reasoner. However, this is limited by a) the reasoning support offered by the used SPARQL endpoint and b) the limitations of the OWL-to-SPARQL translation. On the other hand, SPARQL test cases can find validation errors that are not expressible in `OWL`, but within the expressivity of SPARQL (see [1] for more details and a proof that SPARQL 1.0 has the same expressive power as relational algebra under bag semantics). This includes aggregates, property paths, filter expressions etc. Please note that for scalability reasons full `OWL` reasoning is often not feasible on large datasets. Furthermore, many datasets are already deployed and easy to access via SPARQL endpoints. Additionally, the *Data Quality Test Pattern* (DQTP) library may arguably provide a more user friendly approach for building validation rules compared to modelling `OWL` axioms. However, the predefined DQTP library has some limitations as well, in particular a) it requires familiarity with the library in order to choose the correct DQTP and 2) custom validations cannot always correspond to an existing DQTP and manual SPARQL test cases are required.

3 Test Driven Data Engineering Ontology

The Test Driven Data Assessment methodology is implemented using RDF as input and output and complies with our accompanied ontology.[6] The ontology additionally serves as a self-validation layer for the application input (test-cases, DQTPs and TAGs) and output (validation results). The ontology consists of 20

[4] `https://github.com/AKSW/RDFUnit`
[5] `http://RDFUnit.aksw.org`
[6] `http://RDFUnit.aksw.org/ns/core#`



Prefix prov: http://www.w3.org/ns/prov#
Prefix spin: http://spinrdf.org/spin#
Prefix dct: http://purl.org/dc/terms/
Prefix tddo: http://databugger.aksw.org/ns/core#
Prefix rlog: http://persistence.uni-leipzig.org/nlp2rdf/ontologies/rlog#

Fig. 2. Class dependencies for the test driven data engineering ontology.

classes and 36 properties and reuses the PROV [2], RLOG[7] and spin[8] ontologies. As depicted in Figure 2, the ontology is centered around two concepts, the test case definition and generation and the result representation.

[7] http://persistence.uni-leipzig.org/nlp2rdf/ontologies/rlog#

[8] http://spinrdf.org

Test case definition and generation. We encapsulate a list of test cases in a *TestSuite*, a subclass of `prov:Collection` that enumerates the contained test cases with `prov:hadMember`. The class *TestCase* describes an abstract test case. For each test case, we provide provenance with the following properties:

– `:appliesTo` to denote whether the test case applies to a schema, a dataset or an application.
– `:source`, the URI of the schema, dataset or application.
– `:generated` on how the test case was created (automatic or manually).
– `:references` a list of URIs a test case uses for validation.
– `:testCaseLogLevel` an `rlog:Level` this test case is associated with. In accordance to software development, the available log levels are: TRACE, DEBUG, INFO, WARN, ERROR and FATAL.

Additionally, each *TestCase* is associated with two SPARQL queries, a query for the constraint violations and a query for the prevalence of the violations. The prevalence query is optional because it cannot be computed in all cases.

```
# Violation Query              | # Prevalence Query
SELECT DISTINCT ?s WHERE {     | select count(distinct ?s) WHERE {
  ?s  dbo:birthDate  ?v1.      |   ?s dbo:birthDate ?v1 .
  ?s  dbo:deathDate  ?v2.      |   ?s dbo:deathDate ?v2 . }
  FILTER ( ?v1 > ?v2 ) }       |
```

Concrete instantiations of a *TestCase* are the *ManualTestCase* and the *PatternBasedTestCase* classes. In the former, the tester defines the SPARQL queries manually while the in the latter she provides *Bindings* for a *Pattern*. Additionally, the ontology allows the definition of dependencies between test cases. For example *if test case A fails, do not execute test case B*. This is achieved with the *TestCaseDependency* class where `:dependencyFrom` and `:dependencyTo` define the dependent test cases, `:dependencyCondition` is the status result that triggers an execute or don't execute (`:dependencyExecute`) for the dependant test case.

A *Pattern* is identified and described with the `dct:identifier` and `dct:description` properties. The `:sparqlPattern` and `:sparqlPrevalencePattern` properties hold the respective SPARQL queries with placeholder for replacement. For each placeholder a *PatternParameter* is defined and connected to the pattern with the `:parameter` property.

PatternParameters are described with a `dct:identifier` and two restriction properties: the `:parameterConstraint` to restrict the type of a parameter to Operator, Resource, Property or Class and the optional `:constraintPattern` for a regular expression constraint on the parameter values.

Bindings link to a *PatternParameter* and a value through the `:parameter` and `:bindingValue` properties respectively. *PatternBasedTestCases* are associated with *Bindings* through the `:binding` property.

```
[] a tddo:PatternBasedTestCase ;
   tddo:binding [ a       tddo:Binding ;
      tddo:bindingValue lemon:Node ;
      tddo:parameter tddp:OWLDISJC-T1 ] ;
```

A *PatternBasedTestCase* can be automatically instantiated through a *TestAutoGenerator*. Generators hold a `dct:description`, a sparql query (`:generatorSparql`) and a link to a pattern (`:basedOnPattern`).

Result representation. For the result representation we reuse the PROV Ontology. The *TestExecution* class is a subclass of `prov:Activity` that executes a *TestSuite* (`prov:used`) against a `:source` and generates a number of *TestCaseResults*. Additional properties of the *TestExecution* class are `prov:startedAtTime` and `prov:endedAtTime` as well as aggregated execution statistics like: `:testsRun`, `:testsSucceeded`, `:testsFailed`, `:testsTimeout`, `:testsError` and `:totalIndividualErrors`.

The ontology supports four levels or result reporting, two for report on the test case level and two for individual error reporting. All result types are subclasses of the *TestCaseResult* class and for provenance we link to a *TestCase* with `:testCase` and a *TestExecution* with `prov:wasGeneratedBy` properties. The *StatusTestCaseResult* class contains a single `:resultStatus` that can be one of *Success*, *Fail*, *Timeout* and *Error*. The *AggregatedTestCaseResult* class adds up to the *StatusTestCaseResult* class by providing an aggregated view on the individual errors of a test case with the properties `:resultCount` and `:resultPrevalence`.

For the individual error reporting the *RLOGTestCaseResult* generates logging messages through the RLog ontology. For every violation, we report the erroneous resource (`rlog:resource`), a message (`rlog:message`) and a logging level (`rlog:level`). The logging level is retrieved from the *TestCase*.

The *ExtendedTestCaseResult* class extends *RLOGTestCaseResult* by providing additional properties for error debugging by reusing the spin ontology. In detail, an *ExtendedTestCaseResult* is a subclass of `spin:ConstraintViolation` and may have the following properties:

– `spin:violationRoot`: the erroneous resource.
– `spin:violationPath`: the property of the resource that the error occurs.
– `:errorPropertyContext`: lists additional properties that may provide a better context for fixing the error. For example, in the `dbo:birthDate` before a `dbo:deathDate` case, `dbo:birthDate` can be the `spin:violationPath` and `dbo:deathDate` the `:errorPropertyContext`.
– `:errorClassification`: is a sub-property of `dct:subject` that points to a SKOS error classification category.
– `:errorSource`: is a sub-property of `dct:subject` that points to a SKOS error source category. Example values can be data parsing, data publishing, mapping, pre processing, post processing, etc.
– `:errorType`: is a sub-property of `dct:subject` and that points to a SKOS error type category on the triple level. Example values can be: missing property, redundant property, inaccurate property.

The extended error annotation is generated through the *ResultAnnotation* class that is attached to a *TestCase* through the `:resultAnnotation` property. A ResultAnnotation must contain an `:annotationProperty` linking to one of the allowed *ExtendedTestCaseResult* properties and an appropriate value for `:annotationValue`. For the schema-based automatic test case generation some

	Total	Domain	Range	Datatype	Card.	Disj.	Func.	I. Func.
Lemon	172	40	34	1	29	64	3	1
NIF	86	42	24	4		6	10	

Table 1. Number of automatically generated test cases per ontology. We provide the total number of test cases as well as separated per `rdfs domain` and `range`, literal datatype, `OWL` cardinality (min, max, exact), property & class disjointness, functional and inverse functional constraints.

of the annotation may be known only on the *Pattern* level and other on the *TestAutoGenerator* level. Thus, *ResultAnnotations* are allowed in both classes and the error annotation are added up on the test case generation.

Finally, we provide `:testSuite`, an ontology annotation property, that links an ontology to an appropriate *TestSuite* for data validation purposes.

```
<http://example.com/ontology#>
    a owl:Ontology ;
    tddo:testCase <http://example.com/testCase> .
```

4 Test Case Implementation for Linguistic Ontologies

In this section, we will discuss the employment of RDFUnit for *lemon* and *NIF*, especially with regard to these questions: (1) What is the coverage of the automatically generated tests, what are their limitations. (2) Where is it feasible to use the predefined patterns from the pattern library [12]? Are there test cases that are too complex and need manual creation by an expert? (3) Which test cases can not be expressed at all as they are not expressible via SPARQL?

By running the existing RDFUnit Test Auto Generators (TAG) on the *lemon* and *NIF* ontologies we automatically generated 172 test cases for *lemon* and 86 test cases for *NIF* (cf. Table 1). Both ontologies are of similar size: *NIF* contains 19 classes and 46 properties while *lemon* 23 classes and 55 properties. The number of increased test cases in *lemon* results from the higher amount of defined cardinality and disjointness restrictions. The RDFUnit Suite, at the time of writing, does not provide full OWL coverage and thus, complex `owl:Restriction`s cannot be handled yet. In the frame of the examined ontologies, RDFUnit did not produce test cases for unions of (`owl:unionOf`) restrictions such as multiple cardinalities for `lemon:LexicalSense` and `lemon:LemonElement` or restrictions with `owl:allValuesFrom`, `owl:someValuesFrom` and `owl:hasSelf` for *NIF*.

Both *NIF* and *lemon* have defined semantic constraints that can not be captured in OWL and are too complex for the above-mentioned pattern library. In particular, *NIF* and *lemon* use natural language text in the `rdfs:comment` properties as well as their documentations and specification documents.

For *lemon*, the maintainers implemented a Python validator[9], which enables us to directly compare our efforts to a software validator. For *NIF* there was an early prototype of RDFUnit that used only manual SPARQL test cases.

4.1 Lemon

According to Table 1, test cases for `rdfs:domain` and `rdfs:range` restrictions are the largest group, at 43.8%, followed by tests for disjointness (37.4%) and cardinality restrictions (18.8%). The existing *lemon* validator contains 24 test cases for some structural criteria of the *lemon* ontology. 14 of these tests are natively covered by the existing RDFUnit TAGs. Out of the 10 remaining cases, four where on warning and info level, based on recommendations from the ontology's guidelines. They are thus not explicitly stated in OWL, because they don't constitute logical errors and can not be covered by automatic test generation. Of the six remaining errors, two where expressed via `owl:unionOf` and two could not be expressed by the ontology's author because OWL is not able to express them. Additionally, the *lemon* validator reported undeclared properties under the *lemon* namespace. Although this test case can be expressed in SPARQL, it was not implemented at the time of writing.

The last error case not covered was due to an error in the ontology itself. *Lemon* defines that every instance of `lemon:LexicalEntry` may have a maximum of one `lemon:canonicalForm` property. Yet, the validator fails if the instance has no `lemon:canonicalForm`, thus suggesting that instead of the `owl:maxCardinality`, a `owl:cardinality` restriction was intended in this case. These kind of semantic subtleties are usually very hard to detect in the complex domain of ontology engineering. It shows that the intensive engagement necessary to write the test cases already serves to debug the ontologies underlying the datasets. This extends the test-driven approach to the ontology development, apart from the quality assessment.

These test cases could directly be translated into SPARQL queries for testing with RDFUnit. For example, it is suggested that a `lemon:LexicalEntry` should contain an `rdfs:label`. As there is no possibility to express these optional constraints in OWL, this test case was added manually to log matching resources as an info-level notice.

Beyond the implementation of the *lemon* validator as test cases, some additional test cases were added to test for semantic correctness or properties that could be added. For example, the `lemon:narrower` relation, which denotes that one sense of a word is narrower than the other, must never be symmetric or contain cycles.

```
SELECT DISTINCT ?s WHERE {
  ?s lemon:narrower+ ?narrower .
  ?narrower lemon:narrower+ ?s . }
```

[9] https://github.com/jmccrae/lemon-model.net/blob/master/validator/lemon-validator.py

Similarly, if one resource is `lemon:narrower` to another resource, the inverse relationship (`lemon:broader`) should exist in the database.

From the total of ten manual test cases that were defined for *lemon*, five were described as *PatternBasedTestCase*s, using the existing pattern library, and five as *ManualTestCase*s using custom SPARQL queries. However, for brevity we described the test case with the final SPARQL queries.

4.2 NIF

Almost 50% of automated *NIF* test cases were for `rdfs:domain` constraints, 27% for `rdfs:range`, 11% for `owl:FunctionalPropery` restrictions, 7% for disjointness and 5% for proper datatype usage. The early prototype of RDFUnit that is used as the *NIF* validator did not cover any schema constraints and consists of 10 test cases. There exists one test case on the warning level that reports classes of the old namespace.

Other manual test cases include the following restrictions:

– An occurrence of `nif:beginIndex` inside a `nif:Context` must be equal to zero (0).
– The length of `nif:isString` inside a `nif:Context` must be equal to `nif:endIndex`.
– A `nif:anchorOf` string must match the substring of the `nif:isString` from `nif:beginIndex` to `nif:endIndex`. For example:

```
SELECT DISTINCT ?s WHERE  {
  ?s  nif:anchorOf ?anchorOf ;  nif:beginIndex ?beginIndex ;
      nif:endIndex ?endIndex  ;
      nif:referenceContext [ nif:isString ?referenceString ]  .
  BIND (SUBSTR(?referenceString,
               ?beginIndex , (?endIndex - ?beginIndex) ) AS ?test ) .
  FILTER (str(?test) != str(?anchorOf )) . }
```

– `nif:CString` is an abstract class and thus a subclass such as `nif:CStringImpl` or `nif:RFC5147String` must be used.
– All instances of `nif:CString` that are not `nif:Context` must have a `nif:referenceContext` property.
– All instances of `nif:Context` must also be instances of a `nif:CString` subclass.
– Misspelled `rdf:type` declarations for class names, for example `nif: RFC5147-String`.
– All instances of `nif:CString` must have the properties `nif:beginIndex` and `nif:endIndex`.
– all `nif:Context` must have an explicit `nif:isString`, `nif:isString` can only occur with `nif:Context`.

5 Evaluation

[10] http://lemon-model.net/lexica/uby/WktDE/WktEN.nt.gz

Name	Description	Ontology	Type
lemon datasets			
LemonUby Wiktionary EN [10] [5]	Conversion of the English Wiktionary into UBY-LMF model	lemon, UBY-LMF	Dictionary
LemonUby Wiktionary DE [11] [5]	Conversion of the German Wiktionary into UBY-LMF model	lemon, UBY-LMF	Dictionary
LemonUby Wordnet [12] [5]	Conversion of the Princeton WordNet 3.0 into UBY-LMF model	lemon, UBY-LMF	WordNet
DBpedia Wiktionary [13] [9]	Conversion of the English Wiktionary into lemon	lemon	Dictionary
QHL [14] [15]	Multilingual translation graph from more than 50 lexicons	lemon	Dictionary
NIF datasets			
Wikilinks[15] [10]	sample of 60976 randomly selected phrases linked to Wikipedia articles	NIF	NER
DBpedia Spotlight dataset[16] [18]	58 manually NE annotated natural language sentences	NIF	NER
KORE 50 evaluation dataset[17] [18]	50 NE annotated natural language sentences from the AIDA corpus	NIF	NER
News-100[18] [16]	100 manually annotated German news articles	NIF	NER
RSS-500[19] [16]	500 manually annotated sentences from 1,457 RSS feeds	NIF	NER
Reuters-128[20] [16]	128 news articles manually curated	NIF	NER

Table 2. Tested datasets

For evaluation purposes we gathered a representative sample of *lemon* and *NIF* datasets in Table 2. We loaded all the datasets in an open-source edition of

[11] `http://lemon-model.net/lexica/uby/WktDE/WktDE.nt.gz`
[12] `http://lemon-model.net/lexica/uby/wn/wn.nt.gz`
[13] `http://downloads.dbpedia.org/wiktionary/dumps/en/wiktionary_en_2013-09-17_dump-20130726.ttl.bz2`
[14] `http://linked-data.org/datasets/qhl.ttl.zip`
[15] `http://mlode.nlp2rdf.org/datasets/wikilinks-sample.ttl.tar.gz`
[16] `http://www.yovisto.com/labs/ner-benchmarks/data/dbpedia-spotlight-nif.ttl`
[17] `http://www.yovisto.com/labs/ner-benchmarks/data/kore50-nif.ttl`
[18] `https://raw.github.com/AKSW/n3-collection/master/News-100.ttl`
[19] `https://raw.github.com/AKSW/n3-collection/master/RSS-500.ttl`
[20] `https://raw.github.com/AKSW/n3-collection/master/Reuters-128.ttl`

	Size	SC	FL	TO	ER	AErrors	MErrors	MWarn	MInfo
WiktDBp	60M	177	5	-	-	3.746.103	7.521.791	-	3.582.837
WktEN	8M	168	14	-	-	752.018	394.766	-	633.270
WktDE	2M	170	12	-	-	273.109	66.268	-	155.598
Wordnet	4M	166	16	-	-	257.228	36	-	257.204
QHL	3M	170	11	-	1	433.118	538.933	-	538.016
Wikilinks	0.6M	91	4	-	1	141.528	21.246	-	-
News-100	13K	91	2	-	3	3.510	-	-	-
RSS-500	10K	91	2	-	3	3.000	-	-	-
Reuters-128	7K	91	2	-	3	2.016	-	-	-
Spotlight	3K	92	3	-	1	662	68	-	-
KORE50	2K	89	6	-	1	301	55	-	-

Table 3. Overview of the NLP datasets test execution. For every dataset, we provide the size in triples count, the number of test cases that were successful, failed, timed-out and did not complete due to an error. Additionally, we mention the total the number of the individual violations from automated test cases along with errors, warnings and infos from manual test cases.

Virtuoso server (version 7.0)[21] and ran RDFUnit for each one of them. The results of the dataset evaluation are provided in Table 3.

Looking at the results of Table 3 we observe that manual test cases can be of equal importance to the schema restrictions. Additionally we notice that the *lemon*-based datasets were more erroneous than the *NIF*-based datasets. This may be attributed to the following reasons:

– the *NIF* datasets were smaller in size and, thus, better curated.
– the DBpedia Wiktionary datasets is derived from a crowd-sourced source, which makes it more prone to errors.
– the *lemon* ontology is stricter than the *NIF* ontology.
– [16] already used the early prototype of RDFUnit and fixed all data errors found by manual test cases.

All *lemon* datasets failed the info level test case that required at least one and unique `lemon:language` in a `lemon:LexicalEntry`. The existence of a `lemon:subsense` or exactly one `lemon:reference` also failed in all datasets with a high number of violations, except Wordnet that had only 33. Additionally, all datasets had a high number of violation on the `owl:minCardinality` of 1 constraint of `lemon:lexicalForm` on the `lemon:LexicalEntry` class. However, all datasets had the appropriate number of `lemon:canonicalForm` properties, which is a sub-property of `lemon:lexicalForm` and invalidates these errors. This constraint of RDFUnit, stems from the fact that transitive sub-property checking is not implemented at the time of writing. Except from the DBpedia Wiktionary dataset, all other *lemon* datasets had many reports of a `lemon:LemonEntry` without a label.

[21] http://virtuoso.openlinksw.com

The DBpedia Wiktionary dataset had only five failed test cases. With an addition to the previous three, the dataset returned 163K violations due to the disjointness of the `lemon:LexicalEntry` class with the `lemon:LexicalSense` class constraint and 3.5M violations of missing a required `lemon:lexicalForm` property in a `lemon:LexicalEntry`. The same query returned 270K errors in the QHL dataset.

The Uby Wiktionaries had many failed test cases with a very low (less than 10) number of violations except from `owl:minCardinality` of one in `lemon:Form` class for the `lemon:representation` property. This test case returned 430K errors on the English version and 200K errors on the German. Wordnet also failed this test case with 130K violations. Finally, other high in number of violation test cases are found in the QHL dataset and regard incorrect domain (30K) and range (68K) of `lemon:entry` and wrong range of `lemon:sense` (67K).

The most common test case that failed in all *NIF* datasets is the incorrect datatype of `nif:beginIndex` and `nif:endIndex`. Both properties are defined as `xsd:nonNegativeInteger` but were used as string Literals. This is due to a recent change of the *NIF* specification but also showcases the usefulness of our methodology for data evolution. The correct datatype of `nif:beginIndex` and `nif:endIndex` are also the reason for the *NIF* test cases that returned an error. In these cases, substrings based on these properties were calculated on the query (cf. Section 4.2) and non-numeric values did not allow a proper SPARQL query evaluation. This case also expresses the need for chained test cases execution (*TestCaseDependency* in cf. Section 3). The existence of a `nif:beginIndex` and `nif:endIndex` in a `nif:CString` also return violation in spotlight (68) kore50 (51) and Wikilinks (21K) datasets. Finally 21K objects in a `nif:wasConvertedFrom` relation did not have `nif:String` as range.

A direct comparison or our results with the results of the implemented validators cannot be provided in a consistent way. The *NIF* validator contained only 10 test cases while our approach had a total of 96 test cases. The *lemon* validator on the other hand could not finish after 48 hours for the DBpedia Wiktionay dataset and resulted in a multitude of non-RDF logging messages that were hard to filter and aggregate.

6 Related Work

There exist several approaches for assessing the quality of Linked Data. They can be broadly classified into (i) automated (e.g. [8]), (ii) semi-automated (e.g. [6]) or (iii) manual (e.g. [3,14]) methodologies. These approaches are useful at the process level wherein they introduce systematic methodologies to assess the quality of a dataset. However, the drawbacks include a considerable amount of user involvement, inability to produce interpretable results, or not allowing a user the freedom to choose the input dataset. In [11] errors occurring while publishing RDF data along were detected with a description of effects and means to improve the quality of structured data on the web. In a recent study, 4 million RDF/XML documents were analysed which provided insights into the level of conformance these documents had in accordance to the Linked Data guidelines. On the one hand, these efforts contributed towards assessing a vast amount of Web or RDF/XML data, however, most of the analysis was performed automatically, therefore overlooking the problems arising due to contextual discrepancies.

The approach described in [7] advocates the use of SPARQL and SPIN for RDF data quality assessment and shares some similarity with our methodology. However, a domain expert is required for the instantiation of test patterns. *SPARQL Inferencing Notation* (SPIN) [22] is a W3C submission aiming at representing rules and constraints on Semantic Web models. SPIN also allows users to define SPARQL functions and reuse SPARQL queries. The difference between SPIN and our pattern syntax, is that SPIN functions would not fully support our *Pattern Bindings*. SPIN function arguments must have specific constraints on the argument datatype or argument class and do not support operators, e.g. '=', '>', '!', '+', '*', or property paths.[23] However, our approach is still compatible with SPIN when allowing to initialise templates with specific sets of applicable operators. In that case, however, the number of templates increases. Due to this restrictions, with SPIN we can define fewer but more general constraints. One of the advantages of converting our templates to SPIN is that the structure of the SPARQL query itself can be stored directly in RDF, which is, however, very complex and difficult to manage. From the efforts related to SPIN, we re-used their existing data quality patterns and ontologies for error types.
Pellet Integrity Constraint Validator [17](ICV)[24] translates OWL integrity constraints into SPARQL queries. Similar to our approach, the execution of those SPARQL queries indicates violations. An implication of the integrity constraint semantics of Pellet ICV is that a partial unique names assumption (all resources are considered to be different unless equality is explicitly stated) and a closed world assumption is in effect. We use the same strategy as part of our methodology, but go beyond it by allowing users to directly (re-)use DQTPs not necessarily encoded in OWL.

7 Conclusion and Future Work

In this article, we extended a previously introduced methodology for test-driven quality assessment. In particular, a data engineering ontology was described in detail. We applied the RDFUnit Suite implementing this methodology to the NLP domain, an area in which RDF usage is currently rising and there is a need for quality assessment. In particular, we devised 277 test cases for NLP datasets using the Lemon and *NIF* vocabularies.

In future work, we aim to extend the test cases to more NLP ontologies, such as MARL, NERD and ITSRDF. We also plan to further increase the scope of the framework, e.g. for the recently changed namespaces of *NIF* and *lemon* deprecation warnings should be produced. Another extension is the modeling of

[22] http://www.w3.org/Submission/spin-overview/
[23] http://www.w3.org/TR/2010/WD-sparql11-property-paths-20100126/
[24] http://clarkparsia.com/pellet/icv/

dependencies between test cases, which is currently done manually and could be automated. Furthermore, we also want to apply our methods on services: Usually, semantically enriched NLP services use text as input and return annotations in RDF, which could then be verified by RDFUnit to validate their output.

Acknowledgments. We would like to thank John McCrae for his support regarding lemon. This work was supported by grants from the EU's 7th Framework Programme provided for the projects LIDER (GA no. 610782), LOD2 (GA no. 257943) and GeoKnow (GA no. 318159).

References

1. R. Angles and C. Gutierrez. The expressive power of sparql. In *The Semantic Web-ISWC 2008*, pages 114–129. Springer, 2008.
2. K. Belhajjame, J. Cheney, D. Corsar, D. Garijo, S. Soiland-Reyes, S. Zednik, and J. Zhao. Prov-o: The prov ontology. Technical report, 2013.
3. C. Bizer and R. Cyganiak. Quality-driven information filtering using the WIQA policy framework. *Web Semantics*, 7(1):1 – 10, Jan 2009.
4. L. Bühmann and J. Lehmann. Pattern based knowledge base enrichment. In *ISWC*, 2013.
5. J. Eckle-Kohler, J. P. McCrae, and C. Chiarcos. lemonuby - a large, interlinked, syntactically-rich resource for ontologies. Submitted to the Semantic Web Journal.
6. A. Flemming. Quality characteristics of linked data publishing datasources. Master's thesis, Humboldt-Universität of Berlin, 2010.
7. C. Fürber and M. Hepp. Using sparql and spin for data quality management on the semantic web. In W. Abramowicz and R. Tolksdorf, editors, *BIS*, volume 47 of *Lecture Notes in Business Information Processing*, pages 35–46. Springer, 2010.
8. C. Guéret, P. T. Groth, C. Stadler, and J. Lehmann. Assessing linked data mappings using network measures. In *ESWC*, 2012.
9. S. Hellmann, J. Brekle, and S. Auer. Leveraging the crowdsourcing of lexical resources for bootstrapping a linguistic data cloud. In *JIST*, 2012.
10. S. Hellmann, J. Lehmann, S. Auer, and M. Brümmer. Integrating nlp using linked data. In *ISWC*, 2013.
11. A. Hogan, A. Harth, A. Passant, S. Decker, and A. Polleres. Weaving the pedantic web. In *LDOW*, 2010.
12. D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, and R. Cornelissen. Test-driven evaluation of linked data quality. In *WWW*, 2014 (to appear).
13. J. McCrae, G. Aguado-de Cea, P. Buitelaar, P. Cimiano, T. Declerck, A. Gmez-Prez, J. Gracia, L. Hollink, E. Montiel-Ponsoda, D. Spohr, and T. Wunner. Interchanging lexical resources on the semantic web. *LRE*, 46(4):701–719, 2012.
14. P. N. Mendes, H. Mühleisen, and C. Bizer. Sieve: linked data quality assessment and fusion. In *EDBT/ICDT Workshops*, pages 116–123. ACM, 2012.
15. S. Moran and M. Brümmer. Lemon-aid: using lemon to aid quantitative historical linguistic analysis. In *LDL*, 2013.
16. M. Röder, R. Usbeck, S. Hellmann, D. Gerber, and A. Both. N3 - a collection of datasets for named entity recognition and disambiguation in the nlp interchange format. In *LREC*, 2014.
17. E. Sirin and J. Tao. Towards integrity constraints in owl. In *Proceedings of the Workshop on OWL: Experiences and Directions, OWLED*, 2009.
18. N. Steinmetz, M. Knuth, and H. Sack. Statistical Analyses of Named Entity Disambiguation Benchmarks. In *NLP and DBpedia WS @ ISWC*, 2013.