A short tutorial on rrdf

Egon L. Willighagen

1 Introduction

After RDF and SPARQL integration into Bioclipse was finished, I realized that impact would even be larger if I would add support for that to R. That was in 2011. This package does just that, and allows to work with RDF data and perform SPARQL queries. I am indepted to Rajarshi Guha, whos rcdk package provided me with a template to create this package from [1].

This tutorial will demo in two sections how RDF can be handles and how SPARQL [2] functionality can be used.

But before we start, we first have to load the library:

```
> library(rrdf)
```

BTW, this package makes extensive use of Jena, and without this great Open Source project this package would have had to use an alternative.

2 Handling triples

To handle triples, we first need a triple store. At this moment only in-memory stores are supported, though the code shared with Bioclipse was later been extended with on-disk stores, and I still need to port that code to this package.

The package supports two kinds of stores, one that has minimal ontology support, and one basically just handles triples. Both can be created with the *new.rdf* command:

```
> ontStore = new.rdf()
> store = new.rdf(ontology=FALSE)
```

The store is in fact a Jave object form the Jena library, so that simply printing the object is somewhat uninformative:

```
> store

[1] "Java-Object{<ModelCom   {} | >}"
```

But, we can use the *summarize.rdf* method instead:

```
> summarize.rdf(ontStore)

[1] "Number of triples: 40"

> summarize.rdf(store)

[1] "Number of triples: 0"
```

Possibly, we will get a store prefilled with some 40 triples, defining core schema. When we later serialize the RDF into file formats, we normally no longer see these triples.

With a store at hand, we can start playing with triples.

2.1 Minting triples

This package has two methods for adding triples. One has a resource-typed object, the other is for triples with Literal values. Adding a triple with a resource use this method:

```
> add.triple(store,
+   subject="http://example.org/Subject",
+   predicate="http://example.org/Predicate",
+   object="http://example.org/Object"
+ )
> summarize.rdf(store)

[1] "Number of triples: 1"
```

We can see now that we have one more triple than we had earlier. The other method is for adding data triples, and now the last parameter is a Literal:

```
> add.data.triple(store,
+   subject="http://example.org/Subject",
+   predicate="http://example.org/Predicate",
+   data="Value"
+ )
```

This method also allows you to set the type of the Literal, but at this time only using XML Schema data types using their local names:

```
> add.data.triple(store,
+   subject="http://example.org/Subject",
+   predicate="http://example.org/Predicate",
+   data="Value",
+   type="string"
+ )
```

2.2 Serialization

A store is nice, but to interoperate with other tools, we commonly have serialize the data. For this, we can use the *save.rdf* method (the filename is POSIX-based):

```
> tmpfile = tempfile(fileext=".rdfxml")
> save.rdf(store, tmpfile)

[1] "/tmp/RtmpXDdlU9/file402a20275c9a.rdfxml"
```

This example shows that the methods takes the RDF/XML format as default. That is, the verbose RDF/XML. But you can also save as abbreviated RDF/XML and as Notation3:

```
> save.rdf(store, tempfile(fileext=".ardfxml"), format="RDF/XML-ABBREV")

[1] "/tmp/RtmpXDdlU9/file402a5efa8405.ardfxml"
```

```
> save.rdf(store, tempfile(fileext=".n3"), format="N3")

[1] "/tmp/RtmpXDdlU9/file402a1f76b9c9.n3"
```

To customize the output, we can define prefixes:

```
> add.prefix(store, "ex", "http://example.org/")
> add.prefix(store, "xsd", "http://www.w3.org/2001/XMLSchema#")
> save.rdf(store, tempfile(fileext=".n3"), format="N3")

[1] "/tmp/RtmpXDdlU9/file402a39b9ed60.n3"
```

The output will then resemble:

```
\@prefix ex:      <http://example.org/> .
\@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .

ex:Subject
      ex:Predicate "Value" , "Value"^^xsd:string , ex:Object .
```

2.3 Loading triple files

Minting triples one by one can be fun for a while, but soon you will want to just load a set of triples from a data file. This is done with the *load.rdf* method. Let's read this file:

```
\@prefix rdfs:    <http://www.w3.org/rdf/schema/#> .
\@prefix ex:      <http://example.org/> .
\@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .

ex:Methane
  rdfs:label "methane"^^xsd:string ;
  ex:formula "CH4"^^xsd:string ;
  ex:carbonCount "1"^^xsd:integer .
ex:Ethane
  rdfs:label "ethane"^^xsd:string ;
  ex:formula "C2H6"^^xsd:string ;
  ex:carbonCount "2"^^xsd:integer .
ex:Propane
```

```
  rdfs:label "propane"^^xsd:string ;
  ex:formula "C3H8"^^xsd:string ;
  ex:carbonCount "3"^^xsd:integer .
ex:Butane
  rdfs:label "butane"^^xsd:string ;
  ex:formula "C4H10"^^xsd:string ;
  ex:carbonCount "4"^^xsd:integer .
```

This file contains information about a series chemical compounds, called alkanes. For that, we use this code:

```
> exData = load.rdf("ex.n3", format="N3")
> summarize.rdf(exData)
```

```
[1] "Number of triples: 52"
```

Note that we have the additional 40 triples from the ontology model. It therefore read 12 triples from the file.

And with this ability, we are ready to do the basic stuff: create, load, and save RDF. We are ready to start analyzing the data contained in the RDF, which starts with extracting the data we are interested in.

3 Querying RDF stores

To query RDF stores, the SPARQL query language was developed. The rrdf package allows you to query local and remote stores. This tutorial will not introduce the SPARQL language itself, for which there are plenty of resources available on the web.

We demonstrate the SPARQL functionality with the alkane data loaded in Section 2.3. For example, when we want to find all alkanes with more then 2 carbons we can do:

```
> sparql.rdf(exData,
+   paste(
+     "SELECT ?name ?carbons WHERE {",
+     " ?alkane <http://www.w3.org/rdf/schema/#label> ?name ; ",
+     " <http://example.org/carbonCount> ?carbons ",
+     "}"
+   )
+ )
```

```
     name      carbons
[1,] "butane"  "4"
[2,] "propane" "3"
[3,] "ethane"  "2"
[4,] "methane" "1"
```

The results will be stored in a matrix, and the variable names in the query will be used as column names. We can even say which column should be used as row names:

```
> sparql.rdf(exData,
+   paste(
+     "SELECT ?name ?carbons WHERE {",
+     " ?alkane <http://www.w3.org/rdf/schema/#label> ?name ; ",
+     " <http://example.org/carbonCount> ?carbons ",
+     "}"
+   ),
+   rowvarname="name"
+ )

        carbons
butane  "4"
propane "3"
ethane  "2"
methane "1"
```

The latter has the added value that now the matrix is numeric.

3.1 Remote SPARQL

Similarly, we can do remote queries. We just replace the local store with a string representation of the SPARQL end point URI, and slightly change the method name:

```
sparql.remote(
  "http://rdf.farmbio.uu.se/chembl/sparql",
  paste(
    "SELECT DISTINCT ?predict WHERE {",
    " ?subject ?predict ?object ",
```

```
    "}"
  )
)
```

References

[1] R. Guha. rcdk. http://cran.r-project.org/web/packages/rcdk, 2010.

[2] A. Seaborne and S. Harris. SPARQL 1.1 query. W3C working draft, W3C, Oct. 2009. http://www.w3.org/TR/2009/WD-sparql11-query-20091022/.