



LarKC

The Large Knowledge Collider

a platform for large scale integrated reasoning and Web-search

FP7 – 215535

D2.2.2, 2.5.2. Month 24 Selection Components (report accompanying two software deliverables)

Coordinator: Danica Damljanovic

**With contributions from: D. Damljanovic, J. Petrak,
M.Greenwood, A.Roberts, H.Cunningham (University of
Sheffield), I. Peikov, A. Kyriakov (Ontotext), Mihai Lupu
(IRF), Jose Quesada (MPI)**

Quality Assessor: V. Tresp, B. Bishop

Quality Controller: D. Damljanovic

Document Identifier:	LarKC/2008/D2.2.2, 2.5.2./V2.0
Class Deliverable:	LarKC EU-IST-2008-215535
Version:	2.0
Date:	March 23, 2010
State:	final
Distribution:	public



EXECUTIVE SUMMARY

This report accompanies the Month 24 software deliverables D2.2.2. (baseline components) and D2.5.2 (geometrical semantics components v2). Since Month 12, we have updated the Month 12 baseline plugins by adding a result set size limit option and implemented a more sophisticated IR Selector. With regards to geometrical semantic models, we have studied various advanced vector space models, and applied Random Indexing to RDF, as the most scalable in comparison to other similar methods. We have developed LarKC plugins for experimentation with various lexicalisations of RDF, and application of Random Indexing for the task of selection.



DOCUMENT INFORMATION

IST Project Number	FP7 – 215535	Acronym	LarKC
Full Title	The Large Knowledge Collider: a platform for large scale integrated reasoning and Web-search		
Project URL	http://www.larkc.eu/		
Document URL			
EU Project Officer	Stefano Bertolo		

Deliverable	Number	2.2.2, 2.5.2.	Title	Month 24 Selection Components (report accompanying two software deliverables)
Work Package	Number	2	Title	Identifying and Selecting

Date of Delivery	Contractual	M24	Actual	31-Mar-10
Status	2.0		final	<input checked="" type="checkbox"/>
Nature	prototype <input checked="" type="checkbox"/> report <input type="checkbox"/> dissemination <input type="checkbox"/>			
Dissemination Level	public <input checked="" type="checkbox"/> consortium <input type="checkbox"/>			

Authors (Partner)	University of Sheffield, Ontotext			
Resp. Author	Danica Damljanovic		E-mail	d.damljanovic@dcs.shef.ac.uk
	Partner	University of Sheffield	Phone	+44 114 22 21 931

Abstract (for dissemination)	This report accompanies the Month 24 software deliverables D2.2.2. (baseline components) and D2.5.2 (geometrical semantics components v2). Since Month 12, we have updated the Month 12 baseline plugins by adding a result set size limit option and implemented a more sophisticated IR Selector. With regards to geometrical semantic models, we have studied various advanced vector space models, and applied Random Indexing to RDF, as the most scalable in comparison to other similar methods. We have developed LarKC plugins for experimentation with various lexicalisations of RDF, and application of Random Indexing for the task of selection.
Keywords	Selection, baseline, geometrical methods, Vector space model, WP2

Version Log			
Issue Date	Rev No.	Author	Change
25/Jan/2010	1	Danica Damljanovic	0.1 – the first draft
01/March/2010	2	Danica Damljanovic	0.2 – adding description of plugins
01/March/2010	3	Danica Damljanovic	0.3 – adding more into Geometrical models chapter
09/March/2010	4	Ivan Peikov	0.4 – input for baseline plugins
09/March/2010	5	Johann Petrak	0.5 – review and comments on geometrical models part
09/March/2010	6	Danica Damljanovic	0.6 – integrating baseline plugins part and applying comments from Johann
13/March/2010	7	Danica Damljanovic	0.7 – comments from Mihai
18/March/2010	8	Danica Damljanovic	0.8 – comments from QA (Barry)
22/March/2010	9	Danica Damljanovic	0.9 – few minor edits



PROJECT CONSORTIUM INFORMATION
















Participant's name	Partner	Contact
Semantic Technology Institute Innsbruck, Universitaet Innsbruck	 	Prof. Dr. Dieter Fensel Semantic Technology Institute (STI), Universitaet Innsbruck, Innsbruck, Austria Email: dieter.fensel@sti-innsbruck.at
AstraZeneca AB		Bosse Andersson AstraZeneca Lund, Sweden Email: bo.h.andersson@astrazeneca.com
CEFRIEL - SOCIETA CONSORTILE A RESPONSABILITA LIMITATA		Emanuele Della Valle CEFRIEL - SOCIETA CONSORTILE A RE- SPONSABILITA LIMITATA Milano, Italy Email: emanuele.dellavalle@cefriel.it
CYCORP, RAZISKOVANJE IN EKSPERI- MENTALNI RAZVOJ D.O.O.		Michael Witbrock CYCORP, RAZISKOVANJE IN EKSPERIMEN- TALNI RAZVOJ D.O.O., Ljubljana, Slovenia Email: witbrock@cyc.com
Höchstleistungsrechenzentrum, Universitaet Stuttgart		Georgina Gallizo Höchstleistungsrechenzentrum, Universitaet Stuttgart Stuttgart, Germany Email : gallizo@hlrs.de
MAX-PLANCK GESELLSCHAFT ZUR FOERDERUNG DER WISSENSCHAFTEN E.V.		Dr. Lael Schooler, Max-Planck-Institut für Bildungsforschung Berlin, Germany Email: schooler@mpib-berlin.mpg.de
Ontotext AD		Atanas Kiryakov, Ontotext Lab, Sofia, Bulgaria Email: naso@ontotext.com
SALTLUX INC.		Kono Kim SALTLUX INC Seoul, Korea Email: kono@saltlux.com
SIEMENS AKTIENGESELLSCHAFT		Dr. Volker Tresp SIEMENS AKTIENGESELLSCHAFT Muenchen, Germany Email: volker.tresp@siemens.com
THE UNIVERSITY OF SHEFFIELD		Prof. Dr. Hamish Cunningham, THE UNIVERSITY OF SHEFFIELD Sheffield, UK Email: h.cunningham@dcs.shef.ac.uk
VRIJE UNIVERSITEIT AMSTERDAM		Prof. Dr. Frank van Harmelen, VRIJE UNIVERSITEIT AMSTERDAM Amsterdam, Netherlands Email: Frank.van.Harmelen@cs.vu.nl
THE INTERNATIONAL WIC INSTI- TUTE, BEIJING UNIVERSITY OF TECHNOLOGY		Prof. Dr. Ning Zhong, THE INTERNATIONAL WIC INSTITUTE Mabeshi, Japan Email: zhong@maebashi-it.ac.jp
INTERNATIONAL AGENCY FOR RE- SEARCH ON CANCER		Dr. Paul Brennan, INTERNATIONAL AGENCY FOR RE- SEARCH ON CANCER Lyon, France Email: brennan@iarc.fr
INFORMATION RETRIEVAL FACILITY		Dr. John Tait INFORMATION RETRIEVAL FACILITY Vienna, Austria Email: john.tait@ir-facility.org



TABLE OF CONTENTS

LIST OF ACRONYMS	7
1 INTRODUCTION	8
2 GEOMETRICAL SEMANTIC MODELS FOR SELECTION	10
2.1 Random Indexing and RDF	10
2.2 Generating subgraphs	10
2.2.1 RDF molecules vs. RDF subgraphs	12
2.3 Lexicalisation of subgraphs	12
2.4 Encode	14
2.5 Search	15
3 TRANSITION OF RESULTS INTO THE LARKG PLATFORM	17
3.1 Extended baseline plugins	17
3.2 IR Selector	18
3.3 Random Indexing Selector	20
3.3.1 Implementation	21
3.3.2 Generating vectors	22
3.3.3 Adding refinement blocks to the SPARQL query	23
3.4 RDFToText Transformer	24
4 EXPERIMENTS	26
4.1 Evaluation of baseline plugins	26
4.2 Experiments with Random Indexing and Linked Life Data	27
5 CONCLUSION AND NEXT STEPS	33
5.1 Graph collapsing based on Random Indexing	33
5.2 Further Application of Random Indexing	34
6 APPENDIX: RANDOM INDEXING FOR SEARCHING LARGE RDF GRAPHS	35
REFERENCES	38



LIST OF ACRONYMS

Acronym	Description
RI	Random Indexing
VSM	Vector Space Model
IR	Information Retrieval
LSA	Latent Semantic Analysis
HAL	Hyperspace Analogy to Language
BEAGLE	Bound Encoding of the Aggregate Language Environment
LDSR	Linked Data Semantic Repository
LLD	Linked Life Data
SPARQL	SPARQL Protocol And RDF Query Language
RDF	Resource Description Framework



1. Introduction

This report accompanies the Month 24 software deliverables D2.2.2. (baseline components) and D2.5.2 (geometrical semantics components v2). Since M12, we have

- Improved existing baseline plugins and developed a new Information Retrieval (IR) selector
- Extended our vision of RDF Molecules to RDF subgraphs, when experimenting with geometrical models for selection. Our approaches use RDF subgraphs as proxies for documents and look for applications of the proximity and ranking operations that advanced IR methods provide.
- Developed an experimental setup for the application of advanced geometrical models to RDF graphs, particularly Random Indexing (RI)
- Started exploitation of large datasets developed in LarKC:
 - **LinkedLifeData**(<http://www.linkedlifedata.com>): we have initiated collaboration with AstraZeneca and Ontotext, on the Early Clinical Drug Development use case.
 - **Linked Data Semantic Repository (LDSR)** (<http://ldsr.ontotext.com>): we have started experiments with LDSR and a set of SPARQL queries released by Ontotext.

At Month 24, we are delivering the following LarKC plugins:

Extended baseline plugins Baseline and Key Phrase Selectors delivered at Month 12 are updated with an additional parameter for the selection size limit. This parameter enables control over the selection result set size and is introduced as a general mechanism to enforce a limit on the number of molecules in the result set.

IR selector is a new selector plugin which combines RDF PageRank with Lucene. In contrast to the Month 12 Baseline and Key Phrase plugins, Lucene's internal relevance score based on the vector space model (VSM) and RDF Rank boosting is used to rank results and order them by relevance.

RDFToText Transformer This is a transformer plugin which we have developed in order to experiment with document based IR methods, such as advanced geometrical models for selection. This transformer generates *virtual* documents, from an RDF graph, and can save the result in various formats. The output of this plugin is used by Random Indexing selector described next.

Random Indexing selector This is a selection plugin which uses RDF subgraphs as proxies for documents and applies the proximity and ranking operations that RI provides. It is based on Semantic Vectors (<http://code.google.com/p/semanticvectors/>) Random Indexing library and AirHead S-Space Package which contains a collection of algorithms for building Semantic Spaces (<http://code.google.com/p/airhead-research/>).



This deliverable is structured as follows. In Section 2 we discuss the task of applying document based IR methods based on vector space models, to RDF graphs. In Section 3 we describe an updated version of the M12 baseline plugins, a newly developed IR selector, the RDFToTextTransformer, which generates documents from RDF graphs, and finally Random Indexing selector. In Section 4 we present our initial experiments: evaluation results of running IR selector with Wordnet and LDSR (Section 4.1), and the initial experimental setup and the experiments of applying Random Indexing to the Linked Life Data repository (Section 4.2). We conclude in Section 5 and draw future directions. One aspect of the work reported in this deliverable is submitted for the Poster track of the ESWC 2010 conference, and is attached as an Appendix in Section 6.



2. Geometrical semantic models for selection

In this chapter, we discuss the task of applying document based IR methods based on vector space models, to RDF graphs.

The standard way of generating a Vector Space Model (VSM) in Information Retrieval (IR) is to generate a term-document matrix, which can be huge for large corpora. In the case of random indexing [5], each keyword is assigned a sparse random vector with a pre-specified dimension. If an object contains multiple keywords, the vector of the object is the sum of the random vectors for the keywords that the object has. In comparison to the conventional method, having predefined vector dimension is an advantage for software implementation.

In D2.5.1., we described various statistical distribution methods such as Bound Encoding of the Aggregate Language Environment (BEAGLE) [3], Hyperspace Analogy to Language (HAL) [4], Latent Semantic Analysis (LSA) [2] and Random Indexing [6]. We proposed to use Random Indexing as it seems to have the greatest potential given the size of the datasets we are faced with in LarkKC.

2.1 Random Indexing and RDF

Application of document based IR methods to RDF graphs requires generating *virtual* documents from these graphs. These documents can contain keywords from a triple, from an RDF molecule, or any other set of statements which we decide to use as an abstraction. After we decide on the abstraction, a very important question to be answered is *what is the context?*. Statistical distribution methods such as LSA or Random Indexing have been proven to work well with human-understandable text. When generating *virtual* documents from RDF, we must simulate human readable text. Generation of these documents is not trivial and based on how we observe the nodes in an RDF Graph, and how we place the 'boundaries' between various subgraphs, results can significantly vary. This process consists of the following two steps:

- derive a set of subgraphs from a huge RDF graph
- lexicalise these subgraphs (generate *virtual* documents)

Table 2.1 compares the application of Random Indexing for a corpus of documents, and an RDF graph.

2.2 Generating subgraphs

The task of deriving a set of subgraphs from a huge RDF graph includes generating a *representative subgraph*, for each RDF node (i.e. a resource identified with a URI,

Document Corpus	RDF graph
Documents	Lexicalisations of RDF subgraphs
Terms	URIs, literals
Find similar words; words in similar context	Find similar URIs/literals; proximate nodes

Table 2.1: Random Indexing and RDF



not a blank node or literal). These representative subgraphs hold relevant information about a particular RDF node. That is, these subgraphs can overlap for nodes which are related.

In the simplest case, each triple could be seen as a subgraph; also, each RDF molecule is a subgraph. According to the definition in D2.2.1 a molecule for the specific node *S* is a set of statements where *S* is a subject – if any of these statements have a blank node as an object, then the molecule is extended recursively. However, while this definition stands for various tasks such as those performed by baseline selection components, for the task of generating subgraphs in preparation for the application of document based IR methods, we have encountered several problems:

- **Blank nodes are generally not used in the ABOX for OWL-Lite.**
- **Blank node often replaced by *code nodes*.** A *code node* is an RDF node which has an artificially generated URI such as for instance *umls-concept:C0278504*¹. They are usually introduced to model complexities, and are roughly equivalent to blank nodes in RDF. Such nodes are often used in, for example, UMLS dataset, which is a part of Linked Life (a dataset generated for the LarKC project use cases as a part of WP7a). In the case of the above mentioned URI, firstly, this URI is not descriptive and to discover what it refers to, we have to investigate labels. For example, this node would have a *label: Stage I Non-Small Cell Carcinoma of the Lung*. Therefore, Literals such as labels of the *code nodes* often seem to be *one step too far*, while they appear to be important and should be included in the molecule.

The question is whether a triple or an RDF molecule can capture enough of the semantics which is represented through an RDF graph (and in particular, the relation of a particular RDF node with the others), and whether we could extend the notion of molecules in order to capture more information. We investigate this question as a part of the experiments presented in this deliverable.

Another interesting question is whether we want to treat all RDF nodes equally. In particular, how shall we treat the RDF nodes in the TBox? Some of the possible options are:

- make no distinction in treating the TBox and the ABox; that is, generate representative subgraphs for all RDF nodes in both ABox and TBox without any special treatment of the TBox
- generate representative subgraphs for all RDF nodes in the ABox only, and include in these links to the relevant information in the TBox

One way to make a decision about the TBox is to look at the structure of the data. Another way is to experiment with both treatments which have been described and make the decision followed by investigating results. This approach makes sense given the various design decisions which can affect the quality of data, especially due to the trend of automatically generating RDF graphs.

In the experiments we report in this deliverable, we have considered getting either nothing from the TBox or getting classes (defined with `rdf:type` property) of instances only.

¹We use *umls-concept* instead of the full namespace: <http://linkedlifedata.com/resource/umls/id/>



In addition, generating representative subgraphs from multi-language large datasets such as LDSR (developed within WP2 in LarKC), emphasises the importance of language filtering mechanisms. The question is, whether we should consider all languages at once or not, and, if not, how to make the distinction in treating languages? One of the reasons for making such a distinction is, for instance, the application of tokenisation and stemming, as these might differ for different languages.

2.2.1 RDF molecules vs. RDF subgraphs

The semantics in an RDF graph is represented through definitions in the TBox and relations specified between nodes in the ABox. For the task of generating representative subgraphs, we need to decide what is the right level of abstraction of each URI.

If we start with an RDF molecule, even with its modified definition as stated above (with code nodes included), several questions still remain:

- *Original definition of RDF Molecules*: Is the original set of triples as defined by an RDF Molecule the right level of abstraction?
- *Special treatment for labels*: For each triple in an RDF Molecule, shall all labels of all nodes in the triple be included? That is, shall we add to the original definition of RDF Molecule, all those statements which describe literals associated with URIs already included in the RDF Molecule?
- *Transitive size of the subgraph*: Starting from a URI, how many nodes we should include, connected by $N > 1$ edges? Given a URI we need to decide how big the transitive size of a subgraph needs to be in order to properly represent this URI.
- *Special treatment for literals*: In addition, shall nodes connected by $N + 1$ edges be included, if literals? That is, for each node included in the RDF subgraph, shall an additional set of statements which indicates the relation of that node with literals, be included?

Finally, the remaining challenge when generating representative subgraphs is handling properties and deciding on whether a representative subgraph needs to be generated for each property. Properties are used to define relations between concepts in an RDF graph, and an obvious solution is including them, while preserving a distinction from other RDF nodes, as will be discussed later.

Challenges outlined in this section are the base for the experiments which have been conducted before developing the configurable components described in Chapter 3.

2.3 Lexicalisation of subgraphs

Once *representative subgraphs* are generated, they need to be verbalized into *virtual documents*. While these documents are artificial, all semantics expressed through RDF subgraphs should be preserved, and this makes this task very difficult for several reasons:

- *URIs/literals are not words*: An important issue is distinguishing URIs from literals, and also text literals from all the others. While URIs or non text literals should not be changed (e.g., lowercased or stemmed), standard tokenisation



measures should be applied to text literals. In addition, it is often the case that incorrect capitalisation is used in RDF graphs, e.g. *proper nouns* are lowercased, and the stemming algorithm usually relies on the first capital letter in order to distinguish a *proper* from a *common* noun. All this makes the task of lexicalising RDF subgraphs very challenging.

- *Triples are not sentences*: while we could argue that a triple represented in a (Subject, Predicate, Object) form, looks like a very basic human understandable sentence, it is obvious that information contained in triples is often too explicit in comparison to what is usually used in human understandable text.
- *RDF subgraph is not a document*: this subgraph contains a set of triples, and its verbalisation or lexicalisation makes the point of explicitness of a triple even more obvious when these are merged together into one document;

Having these challenges in mind, we have initially adopted the following approach:

- consider URIs and literals as words, and separate them with white spaces
- consider each statement (i.e. each triple) as a sentence and separate sentences by a new line within a document

However, the URI which is being represented this way might be neglected because statistical distribution methods take into account the frequency of words appearing in context. In order to simulate the right frequency we have modified our initial approach as the following:

- consider URIs and literals as words, and separate them with white spaces
- consider each statement as a sentence and separate sentences by a new line within a document;
 - if the statement does not have the *representative* URI² as subject, then include all statements starting with the one where that URI is a subject

We illustrate this with an example.

Given that for the URI `geo:colorado` we identify the following set of triples³ to serve as a representative subgraph for this node:

```
geo:colorado rdf:type geo:State
geo:colorado rdfs:label colorado
geo:colorado geo:abbreviation co
geo:colorado geo:statePopulation 104000
geo:colorado geo:stateArea 2889000
geo:colorado geo:statePopDensity 0.035998616
geo:colorado geo:borders geo:utah
geo:utah rdf:type geo:State
geo:utah rdfs:label utah
geo:utah geo:abbreviation ut
```

²*Representative* URI is the URI for which the representative subgraph is generated.

³We use *geo* as a prefix for namespace <http://www.mooney.net/geo#>



```

geo:utah geo:statePopulation 84900
geo:utah geo:stateArea 1461000
geo:utah geo:statePopDensity 0.05811088
geo:utah geo:borders geo:nevada

```

We could take this as a lexicalisation of the *representative* subgraph for *geo:colorado*. However, if we want to emphasize the fact that this subgraph indeed represents *geo:colorado*, we might want to lexicalise the statements differently, by assuming that each sentence in the document must always begin with the *representative URI* (i.e. *geo:colorado* in this case). For the above set of triples, the lexicalisation following our slightly modified approach looks as the following:

```

geo:colorado rdf:type geo:State
geo:colorado rdfs:label colorado
geo:colorado geo:abbreviation co
geo:colorado geo:statePopulation 104000
geo:colorado geo:stateArea 2889000
geo:colorado geo:statePopDensity 0.035998616
geo:colorado geo:borders geo:utah
geo:colorado geo:borders geo:utah rdf:type geo:State
geo:colorado geo:borders geo:utah rdfs:label utah
geo:colorado geo:borders geo:utah geo:abbreviation ut
geo:colorado geo:borders geo:utah geo:statePopulation 84900
geo:colorado geo:borders geo:utah geo:stateArea 1461000
geo:colorado geo:borders geo:utah geo:statePopDensity 0.05811088
geo:colorado geo:borders geo:utah geo:borders geo:nevada

```

The difference is that now the last eight triples which present the relation with *geo:utah* are preserving the relation between *geo:utah* and *geo:colorado*.

One remaining question when lexicalising RDF subgraphs is handling properties. We have decided to include properties by default, while it is possible to supply the list of properties as a list of *stop words* which would then be excluded from the computation.

2.4 Encode

As with applying any statistical distribution method, an important aspect of generating a semantic space is preprocessing of the text which includes tokenisation, stemming, and stop words. As we have mentioned previously, there is a distinction which we need to make when processing URIs as opposed to processing string literals.

Another important aspect is parameterisation of the method. As we have decided to use Random Indexing, we need to experiment with various parameters such as *vector dimensionality*, *seed length*, and the others (see Section 3.3.2 for the complete list of parameters which we considered). These parameters affect the way the vectors are generated, and consequently can yield different results. As these vectors are relying on random initialisation, an important aspect is stability. Sitbon and Bruza [7] investigated the stability of term representations through document representations, on the TASA corpus of 44 486 documents containing 148 221 different non stop words and the total of 8 605 497 words. They have investigated the effect of dimension reduction, stabilisation with cycles and the size of the context window.



For the minimum frequency of terms set to 2, and the values of initial seeds set to +1 or -1, they have compared the results of 80 TOEFL questions to the gold standard. They have reported experiments with 5 runs.

Their conclusion is that the average results for non-stemmed data are well under the accuracy of stemmed data. With regards to the number of random vectors, the best score is achieved with 1800 dimensions, which is in-line with what was recommended in [1]. The results for each run on stemmed data exhibit large variations suggesting that vector representations are not that stable [7].

Further on, [7] investigated the size of the context windows, using random vectors computed with 1800 dimensions. The window size refers to the total number of words taken into account including the target word. The best results were achieved with a sliding window of the smallest size (3). Therefore, the conclusion is that the model constructed based on the co-occurrences of words only with the previous and the next word (these not being stop words) performs best for the synonym test. Their experiments further reveal that the results with the context window size of up to 9 reached the higher accuracy in comparison to the accuracy reached when using the whole document as context.

This encouraged us to experiment with triples as documents as described in Section 2.3, as these documents contain 3 words exactly (Subject, Predicate, Object). However, as it is mentioned in [7], context based models are more computationally expensive, and therefore we consider experimenting using whole documents as context.

With regards to the technical details, these parameters are passed through the configuration of the components, and details are described in Chapter 3.

2.5 Search

Once we have generated term and document vectors, we can perform search in several ways:

- **Term term:** given a URI/literal find similar URIs/literals.
- **Document term:** given a URI representing a subgraph, find URIs/literals.
- **Term document:** given a URI/literal find similar RDF subgraphs.
- **Document Document:** given an RDF subgraph find similar RDF subgraphs.

Depending on the task, we need to decide on the search method. So far, our application of RI in the context of selection is twofold:

- *finding similarities between two terms:* finding similar terms can be used for refinement of the SPARQL queries, and therefore could be seen as a complementary to already existing selection methods, as it can help with finding more complete and relevant results, in addition to those which would be retrieved without using RI-based refinement. Note that this method is limited to certain SPARQL queries only (see Section 3).
- *finding documents (RDF subgraphs) related to the specific term:* this task is in line with the baseline selection methods, where for a given SPARQL query, we first extract the keywords, and then return a set of RDF molecules in response

to the set of the identified keywords; application of RI is expected to improve the results of the boolean selection model, by extending the list of keywords with a set of proximate ones, according to RI.



3. Transition of results into the LarKC platform

This section describes the plugins delivered as D2.2.2 and D2.5.2. Each subsection gives a brief description of each plugin. The plugins themselves are published in the LarKC source code repository, together with full documentation. Table 3.1 shows the name and location of each plugin.

3.1 Extended baseline plugins

The currently available baseline selection plugins attempt to accomplish their selection task by simplifying the SPARQL query. Another approach for dataset reduction was explored in D2.4.1 through applying cognitively-inspired graph priming techniques. Both simplifying directions yield approximations of the original query process that under different circumstances could skip lots of important results or fetch overwhelmingly many redundant ones, sometimes orders of magnitude more than the supposedly more complicated original query would. This behavioral diversity motivates the need for some means of control over the selection result set size. Although many ad-hoc solutions exist in the context of every particular selection method (e.g. using a list of stop-words for the Key-Phrase Selector) we introduce a general mechanism that allows one to enforce a limit on the number of molecules in the result set. We believe that such a mechanism could be used as a development tool and could also improve the selection process whenever there is some form of relevance ranking available (e.g. in the IR Selector plugin to be introduced below). Additionally, this is a part of the LarKC platform's general vision of passing non-functional parameters as plugin contract constraints.

We implement the size limiting mechanism inside the Baseline Selector which is also a base class of all the keyword selection plugins. Therefore not only the Baseline Selector but all plugins that inherit the Baseline Selector's class (e.g. Key-Phrase Selector and IR Selector) will make use of the size limiting functionality.

The size limit is passed to the plugin's select method through the plugin contract. The plugin contract is extended to contain an optional integer molecule size limit in addition to the SPARQL query for selection. Further on, the selection implementation selects no more molecules than the contracted number or behaves as before if no size limit was set in the contract.

We should note that limiting the number of molecules in the result set is only one of the possible result limiting strategies considered. For instance one could limit **by number of resulting selected statements** or **by number of distinct RDF nodes** to be present in the selection. However, we chose to implement the strategy limiting by number of molecules because we intuitively consider an RDF molecule to be the minimal semantically-complete unit of RDF information available for selection. The other size limiting strategies are prone to cutting off meaningful parts of the result set. Finally, it is easy to show that limiting the number of resulting molecules will always select a superset of the result sets limited by number of statements or by number of distinct RDF nodes. Thus from all limiting strategies considered, the one limiting by number of molecules seems least likely to cut off important pieces of knowledge, the limit size being equal.

From another perspective, limiting by number of molecules could only be adequate for plugins working in terms of molecules. As this will not always be the case for



Plugin	Status	Availability
BaselineFT selector	Updated	https://larkc.svn.sourceforge.net/svnroot/larkc/trunk/plugins/select/SPARQLKeywords
Keyphrase selector	Updated	https://larkc.svn.sourceforge.net/svnroot/larkc/trunk/plugins/select/SPARQLKeywords
IR selector	New	https://larkc.svn.sourceforge.net/svnroot/larkc/trunk/plugins/select/SPARQLKeywords
Random Indexing selector	New	https://larkc.svn.sourceforge.net/svnroot/larkc/trunk/plugins/select/RandomIndexingSelector
RDFToText Transformer	New	https://larkc.svn.sourceforge.net/svnroot/larkc/trunk/plugins/transform/RDFToTextTransformer

Table 3.1: New and updated LarkC plugins and their availability

selection plugins, we’ll consider implementing other result set size limiting strategies for comparison purposes as future work.

3.2 IR Selector

The Month 12 selection plugins (baseline selector and key-phrase selector described in D2.2.1) were based on keyword search and followed the following scheme:

1. Extract keywords from the literals of the input query
2. Retrieve nodes in the RDF graph that contain in their molecule another node for which the keywords are *descriptive* (different meanings of *descriptive* are used in different plugins, e.g. the baseline selector would retrieve nodes in whose molecule there is a literal which contains any of the keywords)
3. Select the molecules of the nodes retrieved in the previous step

The motivation behind such a scheme is that the molecule centers retrieved in step 2 tend to be characterized by all the nodes in their surrounding RDF molecule (built in step 3). Thus if the keywords extracted from the SPARQL query (in step 1) prove descriptive for any of these nodes the descriptiveness is expected to transfer over to the molecule centers.

An obvious deficiency of the baseline full-text search selector is that the bare matching of a keyword rarely guarantees its relevancy. For instance, the fact that someone mentioned “umbrella” in the description of a node doesn’t mean that this node has something to do with umbrellas at all or even if it is topically similar to the notion of “umbrella”. The key-phrase selector that is built on top of the baseline full-text search selector attempts to solve this problem by ranking the keywords using their TF.IDF score and matching against only the highest ranking ones. This at least guarantees



that “umbrella” will be matched against only if it is specific enough to the node it describes.

Another problem common to both the baseline full-text search and the key-phrase selection strategies is that all keywords extracted from the original query are searched for independently, without a notion of relevance. However, in a real world situation a document matching more keywords in a more specific fashion is expected to be more relevant to the query. This and other relevancy issues bring us to the implementation of the current IR selector inspired by classical methods from information retrieval that improve significantly the results of a keyword based retrieval system.

The IR selector plugin requires the following off-line preprocessing steps to be executed over the underlying RDF graph:

1. Compute RDF Rank using the PageRankRDF component (presented in D2.4.1)
2. Compute the textual molecules of all RDF nodes in the graph
3. Index molecules using the Lucene IR engine¹ using RDF Rank as a boosting factor

The RDF Rank computed by PageRankRDF component provides a graph analysis measure of importance for all nodes in the graph which is encoded in the Lucene index and used during query processing to adjust the ranking of results. The PageRankRDF component is implemented inside BigOWLIM and the results from its execution are the rank numbers for all repository nodes. The rank numbers are stored inside the repository and can be accessed and used for ordering from a SPARQL query. For further reference on how RDF Rank is computed and why it reflects the nodes’ importance in graph context see D2.4.1 and its citations.

In order to compute the RDF Rank of an OWLIM repository one starts the GeneratePageRank program (distributed inside BigOWLIM jar):

```
java -DrepositoryPath=<path> com.ontotext.trree.GeneratePageRank
```

(class path specification is omitted for brevity here but needs to include the path to BigOWLIM jar as well as the paths to all dependency jars).

The textual molecule of a node in the RDF graph is a string built as a concatenation of all the literals found in the node’s molecule. Local names of the URIs in the molecule are also added to the textual molecule because very often they name the object represented by the node which becomes important information if the textual molecule is otherwise empty. The textual molecule string is then treated as a Lucene document that needs to be mapped to the molecule center. The boosting factor of this Lucene document is set to the RDF Rank value computed by the PageRankRDF component.

In order to generate Lucene indices for an OWLIM repository one starts the GenerateLuceneIndex program (again inside BigOWLIM jar):

```
java -DrepositoryPath=<path> com.ontotext.trree.GenerateLuceneIndex
```

Both GeneratePageRank and GenerateLuceneIndex programs are system utilities of BigOWLIM and are merely reused in the context of the IR Selector plugin. Once the Lucene index is created the IR selector behaves similarly to other baseline selection

¹<http://lucene.apache.org/>



plugins and only differs in step 2 of the schema described above. Instead of matching the keywords extracted from the input SPARQL query (in step 1), the IR selector builds one big Lucene query using the keywords as terms and lets Lucene find all the nodes whose textual molecule matches the query. Lucene index is queried through OWLIM's special `http://www.ontotext.com/luceneQuery` predicate:

```
select ?s
where
{?s <http://www.ontotext.com/luceneQuery> 'term1 term2 ...'}
```

The results are followed to molecule centers and then expanded to whole molecules in step 3 as in the other Month 12 plugins. In contrast to the keyword selection plugins, Lucene's internal relevance score based on the vector space model (VSM) and RDF Rank boosting is used to rank results and order them by relevance. Note that such an ordering would make the result set size limit feature introduced in the previous section even more useful. Indeed, selecting the most relevant N results is far more likely to select what is needed by the following stages in the LarkC pipeline than the almost random selection of the first N results.

3.3 Random Indexing Selector

Our work on geometrical semantic models for RDF repositories has continued in two main strands, one based on the Semantic Vectors (<http://code.google.com/p/semanticvectors/>) Random Indexing library and the other one based on AirHead S-Space Package which contains a collection of algorithms for building Semantic Spaces (<http://code.google.com/p/airhead-research/>). We have wrapped both libraries for two reasons:

- While SemanticVectors library does only Random Indexing, AirHead contains other statistic distribution methods such as LSA.
- To further demonstrate that *everything* can be used as a plugin within the LarkC platform.

In addition, as our goal in LarkC is to deal with vast amount of data, we are working with HLRS on parallelisation of random indexing methods; as the two libraries are different (SemanticVectors relies on Lucene indexes for example), the parallelisation of the two might differ as well.

While we have not changed AirHead library code, we had to develop additional components in order to use SemanticVectors library. This is due to the difference between the human understandable text and the RDF graphs – these differ in many aspects as many characters in URIs are considered 'special' when processing human text.

While MPI used this library to process human-readable articles (as reported in D2.3.2 Cognitive memories components v2), our version is different in that

- it is adapted to support processing of URIs,
- it uses a newly developed Lucene analyser to generate indexes; these indexes are mandatory input for generating vectors with SemanticVectors library.



The Random Indexing selector works as follows:

- *extracts keywords* from the SPARQL query; here we reuse the code written previously for the baseline plugin described in D2.2.1; this method is now separated from the plugin in order to be reused by other selector plugins if necessary
- find *similar* terms to these keywords, by looking into the vector space previously generated by Random Indexing method (details in Section 3.3.2)
- for each similar term dynamically add a *refinement block* to the initial SPARQL query (details in Section 3.3.3)
- execute the *refined* SPARQL query and return statements

3.3.1 Implementation

We pass the SPARQL query, and the relevant implementation of the RandomIndexing class through the Contract:

```
public static class PluginContract implements Contract {
private static final long serialVersionUID =
4964585541551132712L;
String theQuery;
RandomIndexing randomIndexing;

public RandomIndexing getRandomIndexing() {
return randomIndexing;
};

public void setRandomIndexing(RandomIndexing randomIndexing) {
this.randomIndexing = randomIndexing;
}

public PluginContract(String query, RandomIndexing
randomIndexing) {
theQuery = query;
this.randomIndexing = randomIndexing;
}

public String getInputQuery() {
return theQuery;
}
}
```

As we have implemented two wrappers for Random Indexing method, one based on SemanticVectors, and the other one based on Airhead, this is specified through the contract as well.

To illustrate this, we will give an example. Let's start with a sample SPARQL query:



```
String sparqlQuery =
"SELECT ?s ?p ?o WHERE { ?s ?p ?o . filter (?o='alaska') . }";
```

Then, we need to decide which implementation of Random Indexing interface to use, and configure its parameters. In this example, we will use AirHead implementation:

```
//user AirHead implementation of random indexing
RandomIndexing randomIndexing = new RandomIndexingAH();
```

Vectors are generated the first time this method is called. However, the path to the vectors file must be specified in the `build.properties` file of the plugin. It is then passed on to the `randomIndexing` object.

```
ResourceBundle rb = ResourceBundle
    .getBundle("eu.larkc.plugin.select.RandomIndexingAHSelector");
String filePath = rb.getString("vectorsFilePath");
randomIndexing.setFilePath(filePath);
//number of terms to be returned as a result of search
randomIndexing.setNumberOfSimilarWords(5);
```

Now, we can pass on these two to the Contract and do the selection as follows:

```
// pass the SPARQL query and the random indexing object to the
// contract
Contract contract = new PluginContract(new String(sparqlQuery),
randomIndexing);
SetOfStatements sos = new HTTPRemoteGraph(
    new URIImpl("http://www.mooney.net/geo#"));
// Do selection
LabelledGroupOfStatements ts = s.select(sos, contract, null);
```

By default, the plugin uses the cosine function for calculating similarity between the input query term and the vectors.

3.3.2 Generating vectors

As we have discussed in Section 2.4, vector representations vary significantly based on the parameters used. The important decision to be made at this step is optimal parameters for the method. These parameters are passed through the `randomIndexing` object, and include:

SeedLength Number of +1 and number of -1 entries in a sparse random vector.

Dimensionality Dimension of semantic vector space – number of dimensions to use for the sparse random vectors.

MinTermFrequency Minimal frequency of terms.

Window size The size of the context window.

In addition, for Semantic Vectors library, we have customised a Lucene Analyser in order to be able to index URIs. This analyser is based on:



- **Regular Expression based Tokeniser:** this is to allow a full control over the tokenisation process. The default regular expression is the white space, but any other pattern can be used, for example for tokenising punctuation. Tokenisers which perform quite well on human-understandable text are not suitable for this task, as many words (URIs) would be ignored or wrongly tokenised. In addition, this tokeniser is portable across different languages.
- **Lowercasing:** non-URI strings (literals) are lowercased
- **Stop words:** this list is used for excluding RDF annotation properties such as `rdfs:label` or `rdfs:type`;

In the case of the SemanticVectors library, vectors are generated after the documents are indexed using Lucene.

For AirHead, we decide to lowercase all non URIs, and use White Space tokeniser by default. This tokeniser is more limited than the one mentioned above as it does not perform tokenisation based on a regular expression, but based on the white space only. In future we will investigate using a more sophisticated tokeniser.

3.3.3 Adding refinement blocks to the SPARQL query

The selection method calls the Random Indexing method for finding similar terms. The call to this method is parameterized with `numOfSimilarWords` parameter, which indicates the number of terms to be returned. The returned list is given to `SparqlGenerator` class, which will add the corresponding refinement blocks to the initial SPARQL query. Depending on the type of the term, the refinement blocks will be different. For example, if the returned term is URI, the refinement block would be similar to:

```
{?s ?p ?o . filter (?s=<similar-uri-string>) .}
```

while in case of literals, the refinement block could be more flexible:

```
{?s ?p ?o . filter (regex(str(?o),'similar-literal-string*', 'i')) . }
```

To give an example, if the SPARQL query looks like:

```
String sparqlQuery =
"SELECT ?s ?p ?o WHERE { ?s ?p ?o . filter (?o='ultrasound') . }";
```

The only keyword in this SPARQL query is *ultrasound*. For `numOfSimilarWords` parameter set to 5, the Random Indexing search would return the following list as the most similar terms to *ultrasound*:

1. ultrasound
2. reflections
3. sonography
4. (son-o-gram)
5. bounced



Therefore, if the selection method started with the above mentioned SPARQL query, after refinement, it will look as the following:

```
SELECT ?s ?p ?o
WHERE { ?s ?p ?o . filter ( ?o='ultrasound') .
      UNION
      {?s ?p ?o . filter (regex(str(?s),'ultrasound*', 'i')) .}
      UNION
      {?s ?p ?o . filter (regex(str(?s),'reflections*', 'i')) .}
      UNION
      {?s ?p ?o . filter (regex(str(?s),'sonography*', 'i')) .}
      UNION
      {?s ?p ?o . filter (regex(str(?s),'(son-o-gram)*', 'i')) .}
      UNION
      {?s ?p ?o . filter (regex(str(?s),'bounced*', 'i')) .} }
```

3.4 RDFToText Transformer

This plugin was developed as a precondition for the Random Indexing selector which wraps two different libraries, both requiring different document format. While the AirHead library takes as an input one text file where each line is a document, the SemanticVectors library requires a folder where each file represents a document. Having this in mind, we have implemented a configurable transformer which could generate various formats and also various options with regards to the size of RDF subgraphs and their lexicalisations.

This plugin *transforms* an RDF Graph into Natural Language documents as follows:

- Given an RDF Graph, a set of RDF subgraphs is generated: one for each URI in the graph
- RDF subgraphs are lexicalised into *virtual* documents, according to the specified parameters (see below)
- *Virtual* documents are saved in formats required by Random Indexing selector;

As the RDF triple is the simplest form of an RDF subgraph, the first decision to make is whether to consider each triple as a document, or use more complex RDF subgraphs. Table 3.2 summarises the input and output options. This is configurable through the *documentForTriples* parameter, which indicates whether each triple should be treated as a document or not. The default value is true; if false, the documents will be generated per RDF subgraph.

Additional parameters which could be passed to this transformer are the following:

- *SubjectsQueryString*: the SPARQL query used to generate the first round of RDF subgraphs; the default value is: `SELECT DISTINCT ?X WHERE ?X ?Y ?Z . FILTER isURI(?X)`
- *Depth*: RDF subgraph maximum number of intermediate nodes. Default value is 0. This means that for the default values of this and the previous parameter,



		Output	
		Folder	File
Input	a triple	a folder with files each representing a triple	a file with each line representing a triple
	a set of triples	a folder with files each representing a lexicalised subgraph	a file with each line representing a lexicalised subgraph

Table 3.2: Output of transforming an RDF graph into *virtual* documents, using RDFToText Transformer

the result would be a set of RDF subgraphs which are limited extensions of RDF molecules. Namely, they include statements which define labels and literals for all URIs which are included in the list of statements representing RDF molecules.

- *Lexicaliser*: the class which would lexicalise the selected RDF subgraphs
- *TheDirectoryURL*: path to the file or folder where *virtual* documents will be saved

The intention behind allowing various configuration options for this transformer is the reuse from other plugins. We believe that this component is of significant value for various other tasks in LarKC, and especially selection experiments. Any plugin which uses document based methods, such as those from IR, would benefit from this transformer.



4. Experiments

In this chapter, we present an initial evaluation of the baseline plugins (Section 4.1) and our experiments with the application of Random Indexing on the Linked Life Data (Section 4.2).

4.1 Evaluation of baseline plugins

In order to evaluate the new IR Selector plugin in combination with the result size limit feature we devised two SPARQL queries with different number of keywords (that is, different complexity and selectivity).

The first query (Q1) contains only a single keyword (London):

```
SELECT ?s ?p ?o
  WHERE { ?s ?p ?o
    FILTER( ?o = 'London' )
  }
```

The second one (Q2) extends the first (Q1) with several more keywords, high-traffic ones, rarely used and stop-words:

```
PREFIX dbp-ont: <http://dbpedia.org/ontology/>
PREFIX dbp-prop: <http://dbpedia.org/property/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?Company ?LL ?PL
WHERE {
  ?Company dbp-prop:products [rdfs:label ?PL] ;
    dbp-ont:foundationPlace [rdfs:label ?LL] .
  FILTER ( ( regex(str(?LL), "london", "i") ||
    regex(str(?LL), "madrid", "i") ||
    regex(str(?LL), "berlin", "i") ) &&
    ( regex(str(?PL), "the largest
    european capitals", "i")))
}
```

We executed the IR selector on a version of the WordNet dataset (842K resources, 7.6M statements). The test included two configurations of the IR selection algorithm: limiting the result set size to 10 molecules and to 1000 molecules respectively. The results are summarized in Table 4.1.

	time (ms)	#molecules	#statements
Q1/limit 10	755	10	92897
Q1/limit 1000	25968	1000	3800761
Q2/limit 10	1211	10	49158
Q2/limit 1000	39394	1000	5613736

Table 4.1: Experiments with Wordnet and IR Selector



	time (ms)	#molecules	#statements
Q1/limit 10	372812	10	2628720
Q1/limit 1000	3639892	1000	3119667
Q2/limit 10	92558	10	216146
Q2/limit 1000	4923725	1000	3856657

Table 4.2: Experiments with LDSR and IR Selector

The *#molecules* column shows the number of molecules that were selected in the particular setup and *#statements* column shows the number of statements that were selected as a part of those molecules.

It becomes immediately visible that putting a limit on the number of selected molecules is crucial for keeping the execution time of the selectors within a reasonable range. The results set size limit can and should be adjusted to a value matching the requirements of any particular application using the plugin. We should also note that even though the above results suggest that adding complexity to the query also increases the selection time, this is only due to the increased diversity of the selected subgraph. For instance, the number of selected statements in 1000 molecules in the Q1 case is just about 2/3 of the number of selected statements in 1000 molecules in the Q2 case. Such a great difference in the average molecule size is explained by the greater structural complexity of the graph selected by Q2. Indeed, the first (most relevant) 1000 molecules having anything to do with London will include a lot more information-poor molecules than the first 1000 molecules about London, Madrid or Berlin together. This observation suggests that relevancy (as provided by the combination of VSM and RDF Rank) may directly be related to the average size of the selected molecules, if the literals mentioned in the query are very popular. Relevance to terms of low popularity is likely to reduce the size of the average molecule. RDF Rank should generally result in boosting the average molecule size, if the assumption is that larger molecules have higher RDF Rank. The latter is however quite questionable.

Exploring this relation is a matter of future work and is outside the scope of the current deliverable.

We also repeated the above experiment on a version of the LDSR dataset (136M resources, 1B statements). Table 4.2 summarizes the results.

Although the “geometry” of the results is not drastically different from that of WordNet, one notable tendency becomes visible: while the first 1000 molecules for both queries select similarly sized result sets, the first 10 molecules contain a lot more statements in comparison to the same number of molecules from the experiment with the WordNet dataset. This shows that the more complicated and bigger the dataset becomes, the more important it is to consider some sort of relevancy in the selection.

4.2 Experiments with Random Indexing and Linked Life Data

While we had performed the initial setup of our experiments based on conclusions given in [7], an important weakness of our current experiments is the lack of a gold standard. Here we present the experimental setup and the plan on how to generate the gold standard by involving scientists from the LarkC use case partners such as AstraZeneca and IARC.



	LLD 1	LLD2
number of representative subgraphs	5001	50001
number of statements	372795/595798	4573668/2324130
number of documents	64644	473742
number of terms	417753	1713349
time to generate lucene indexes	9.13 min	65.4 min
time to generate vectors	4.1 min	45min
number of term vectors	258914	1062440
finding similar terms for <i>lung</i>	11.74s	46s
finding similar terms for <i>ultrasound</i>	11.43s	46s
finding similar terms for <i>Raymond</i>	11.3s	46.06 s
finding similar terms for <i>migraine</i>	11.88s	45.48 s
finding similar terms for <i>magnesium</i>	11.31s	44.14 s

Table 4.3: Sizes of LLD subsets used in experiments: with 1800 dimensions, seed length=4, window size=5 and min frequency of the term=2

We have generated two subsets of LLD which both cover MEDLINE articles from December 2009. These are generated with 5001 and 50001 seed URIs respectively, and we refer to these as LLD1 and LLD2. Table 4.3 shows the sizes of these datasets, the number of triples, documents, terms, and the time for generating lucene indexes and vectors. These experiments were performed using the SemanticVectors library, and were executed on a CentOS 5.2 Linux virtual machine running on a AMD Opteron 2431 2.40GHz CPU with 2 cores and 20G RAM allocated to that particular instance. This is the instance set up by Ontotext, in order to support MPI and USFD in the exploitation of the large datasets, and in particular LDSR and LLD.

Our intention with these subsets is to investigate whether *sampling* huge datasets such as LLD can yield reasonable results, before we conduct the experiments on the whole LLD dataset which currently has 4,179,999,703 statements. Our other intention is to limit the scope of the subgraph to a certain number of topics in order to easily perform experiments with end users.

Table 4.4 shows sample results for the exactly same search terms, executed using LLD1 and LLD2 vectors respectively. For the brevity of presentation, we use the following abbreviations:

- *umls* instead of <http://linkedlifedata.com/resource/umls/id/>
- *umls-label* instead of <http://linkedlifedata.com/resource/umls/label/>
- *drugs* instead of <http://www4.wiwiss.fu-berlin.de/dailymed/resource/drugs/>
- *pubmed-mesh* instead of <http://linkedlifedata.com/resource/pubmed/mesh/>



LLD1	LLD2
<i>lung</i>	
0.9999982:lung	0.99999946:lung
0.8043262:non-small	0.7971309:lung.
0.7727439: umls:C0242379	0.75357807:pulmonary
0.7674137: umls:C0684249	0.66324264:alveoli,
0.7665758:(bronchogenic	0.6597513:airways
0.75594306: umls:C0007131	0.6384461:alveoli
0.7551168:dealt	0.6337491:(bronchogenic
0.75140786:non-small-cell	0.6326602:lungs.
0.730418: umls:C0149925	0.6276034:lungs
0.72947603:sclc1	0.6257749:non-small
0.72947603:hematoxyphilic	0.6204259:bronchioles,
0.7190478:nonsmall	0.616685:emphysema.
0.6976836:bronchogenic.	0.6132897:obstructive
0.6855592:lung.	0.6063821:emphysema/copd
0.66824996:adenocarcinoma;	0.6037152:(tiny
0.6564659:sccl	0.6037152:gas-exchange
0.65381706:surface),	0.6037152:sacs)
0.65381706:(costal	0.6037152:breathe.
0.65381706:basal).	0.59502864:(copd).
0.65381706:basal),	0.5922818:airspaces
<i>ultrasound</i>	
1.0:ultrasound	1.000004:ultrasound
0.96097344:reflections	0.93754905:reflections
0.94079155:sonography	0.92070985:(sonogram).
0.939101:(son-o-gram)	0.92070985:(son-o-gram)
0.939101:bounced	0.92070985:bounced
0.939101:(dctd-dip)	0.92070985:ultrasound)
0.939101:(ul-tra-son-og-ra-fee)	0.92070985:(dctd-dip)
0.939101:xays,	0.92070985:xays,
0.939101:ultrasound)	0.92070985:megahertz.
0.939101:megahertz.	0.92070985:(ul-tra-son-og-ra-fee)
0.939101:(sonogram).	0.92053723:ultrasonogram
0.9374894:ultrasonogram	0.9197309:sonogram
0.93461853:echotomographies	0.91946274:echotomographies
0.93412834:sonogram	0.9080941:echography
0.9134829:(us)	0.90598756:echoes
0.9088713:dyes,	0.88815176:bounce
0.90674514:echography	0.8728361:echotomography
0.90409994:transducer.	0.87112284:sonography
0.87819326:waves	0.86464053:ultrasonic
0.87094223:ultrasonography	0.8285665:fluoroscopy.
<i>migraine</i>	



Table 4.4 – Continued

LLD1	LLD2
1.0000001:migraine	1.0000011:migraine
0.9392936:aura)	0.95913523:mgau
0.9392936:noises.	0.95875674:noises.
0.9374501:mgau	0.9585649:mgr1
0.9374048:mgr1	0.9564931:aura)
0.935861:migraines	0.947905:symptoms).
0.9176903:loud	0.94761807:migraines
0.91356856:symptoms).	0.9428572:loud
0.89396656:2004:	0.9261631:2004:
0.8934265:lights	0.9223649:ed.
0.8818337:ed.	0.9160708:aura,
0.86041343:suppl	0.90922475:ma
0.8452679:(international	0.9064741:(international
0.8105476:headaches.	0.89782655:lights
0.7863382:aura,	0.89162:(without
0.77662176:aura	0.8628451:pulsatile
0.7739831:(without	0.82879156:suppl
0.7618793:bright	0.8005382:migraine,
0.75807947:ma	0.79893017:photophobia,
0.75656563:photophobia,	0.78821385:disturbances;
<i>magnesium</i>	
0.9999905:magnesium	1.0000014:magnesium
0.41371858:hydroxide	0.754989: umls:C0024473
0.37817737: drugs:3496	0.67077523: pubmed-
0.35977152:bicarbonate/magnesium	mesh:magnesium+deficiency
0.32650134:stearate,	0.6600064:d008275
0.3217894:inscribed	0.6447856:magnesium-centered
0.31527534: umls:C0024476	0.6447856:seeds; 0.6091367:mag-
0.31241718:defecate.	nesia 0.59572154: umls-
0.31241718:(peristalsis)	label:A0483217 0.59456575: umls-
0.31241718:(mg(oh)2)	label:A14839438 0.5936411: umls-
0.31241718:brucite.	label:A0082408 0.5917917: umls-
0.2990279:magnesium-centered	label:A0045887 0.59132934: umls-
0.2990279:seeds;	label:A14941991
0.29109553: umls:C0373675	0.59109825: umls-
0.28516296:28:08.04.24	label:A14880543
0.2767124:(omeprazole/sodium	0.5883242: umls-label:A0082409
0.2767124:zegerid.) geriatric	0.58786184: umls-label:A0045839
0.2767124:4.) gastric	0.5876306: umls-label:A0082407
0.2767124:95%. metabolism:	0.58556384: umls-label:C0991706
0.2767124:(three-fold	0.58503854: umls:C0717898
	0.5827761: umls-label:A14962657
	0.58001375:sulfa
<i>Raynaud</i>	



Table 4.4 – Continued

LLD1	LLD2
1.0000011:raynaud	0.9999999:raynaud
0.84478855:prickling;	0.9367573:prickling;
0.8388415: umls:C0034734	0.9123951:raynauds
0.7704591:raynauds	0.60944057:raynaud's
0.49496746: umls:C0040021	0.36096582: umls:C0034735
0.48361167:d013919	0.34411094:200,000
0.46680346: umls:C0264995	0.33015952:c50724
0.4652673:c6038	0.31754243: umls-
0.46381205: umls-label:A0125716	label:A10814020
0.45014036:443.1	0.3100607: umls-label:A7567641
0.4373745:isoxsuprine	0.30683875: umls-label:A7828038
0.4294775:thromboangitis	0.30438593: umls-
0.42879438: umls-label:A0587378	label:A10769478
0.4268641:paleness	0.3027457:tissue/autoimmune
0.4259581: pubmed-	0.30070782:paleness
mesh:thromboangiitis+obliterans	0.29970297: umls-
0.42172796:raynaud's	label:A14963441
0.41982332:tolazoline	0.29613507: umls-label:A8376821
0.418172:buenger	0.29268482:fingers,
0.41442454:c35070	0.28917918: umls:C0034734
0.4134803: umls:C0029822	0.2889993: umls-label:A0006991
	0.2889993: umls-label:A0450709
	0.2854314: umls-label:A8339791

Table 4.4: Finding *similar* terms based on the vectors from LLD1 and LLD2

While it is interesting to see how results vary when used with different subsets of LLD, one being the subset of the other, in order to make a reasonable comparison between the two, we need a gold standard. In order to create a gold standard, we plan to involve the scientists as follows:

1. Generate a list of keywords, e.g. 10 keywords of interest to scientists and also covered by LLD
2. Run RI term-term search and retrieve the list of top N relevant keywords (this could also be done by setting the threshold, e.g., 0.8) for each of the 10 keywords initially selected in step 1
 - if our intention is comparing results given with two subsets of different size, we should run this step twice: first, with vectors from the smaller and then with the vectors from the bigger subset
3. We give this list to the scientists and ask them to cross out those that are irrelevant; optionally they can add some of the synonyms which are not included in the list



4. We assume that those which are not crossed are relevant and this is our golden standard.

There are plenty of variations of the above mentioned scenario such as tweaking the parameters of the RI and generating vectors several times with different parameters and then comparing results. Another interesting variation is changing the way the documents are generated before generating vectors. We could also compare how the size of the dataset affects the results, and whether scalability can be achieved by using *sample* subgraphs from huge datasets such as those we are using in LarkKC.



5. Conclusion and next steps

We have described M24 plugins developed for the task of selection, which have been delivered as D2.2.2. and D2.5.2. More specifically we have updated baseline plugins with a new size limit parameter, and developed a more sophisticated baseline selector which combines PageRank with Lucene. With regards to geometrical semantic components, we have experimented with the application of Random Indexing for RDF repositories, and have developed one transformer plugin (RDFToText Transformer) in order to apply document based methods such as RI to RDF. Finally, we have wrapped Semantic Vectors (<http://code.google.com/p/semanticvectors/>) Random Indexing library and AirHead S-Space Package which contains a collection of algorithms for building Semantic Spaces (<http://code.google.com/p/airhead-research/>), into the Random Indexing Selector plugin. This plugin uses RI based refinement of the SPARQL query in order to improve the results of selection.

The approach we described in this deliverable appears interesting, and we did some initial experiments to investigate its utility. Unfortunately, the main weakness is the lack of the gold standard to which we could compare our results. We are currently working with AstraZeneca in order to select a set of keywords and perform evaluation with their scientists as described in 4.2. Depending on the evaluation results, we will consider some other ways of applying RI to RDF, some of which we outline next in Section 5.1.

In parallel, we plan to continue with the experiments explained in Section 4.2, and variations of the parameter settings for RI, in order to monitor how they influence results. Our other intention is to experiment with *RDF graph sampling* which could be generated and used for generating vectors, without having to process huge graphs. With regards to the huge dataset exploitation, such as LLD, we are currently investigating the parallelisation of Airhead RI method, in order to be able to run the experiments with the whole dataset.

In the next two subsections we outline some interesting scenarios of application of RI to RDF Graphs, some of which we will investigate in future. We will base our decision on the availability of the evaluation resources, and the possibility to evaluate our experiments within the LarKC usecases.

5.1 Graph collapsing based on Random Indexing

One interesting future direction could be applying RI as an off-line step that compacts an RDF space based on the lexical proximity of the term set derived from its molecular structures. This would work as follows:

- Derive a set of lexemes from each RDF molecule. In each case the lexeme set is associated with a unique identifier for the molecule from which it is derived.
- We then calculate clusters of lexeme sets that are maximally proximate and for each set of URIs associated with their molecule we replace them with their cluster id. This allows us to reduce the size of the RDF graph proportionally to the average size of the clusters.
- Evaluation of the properties of the approach is then relatively straightforward:



- We make queries on the full knowledge base.
- We make the same queries on the compacted knowledge base.
- We measure the degree of inclusion of the result set URIs in the former set in the latter. (This can be applied in both cases where we do not know the part of the result that the user is interested in and also in cases where we have a gold standard result.)

5.2 Further Application of Random Indexing

In addition to the scenario explained in the previous subsection, we envisage the potential use of RI and similar methods in several other settings:

1. Improved user experience: suggestions for query refinement/expansion: we can limit or expand the set of results using suggestions created by RI. This is something we have already done, however it is still not exposed to the LinkedLifeData interface for example.
2. RI provides a similarity measure between all combinations of nodes/subgraphs and nodes/subgraphs, so we can do clustering: we could generate clusters, and then execute the SPARQL query against the most relevant one incrementally until we get results
3. Setting up the weights for Spreading Activation: $\text{Weight} = \text{Sim}(\text{URI1}, \text{URI2})$



6. Appendix: Random Indexing for Searching Large RDF Graphs

This section contains the short paper on the topic described in this deliverable, which is submitted for the poster track of the ESWC 2010 conference.

Random Indexing for Searching Large RDF Graphs

Danica Damljanovic, Johann Petrak, and Hamish Cunningham

Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield, UK

Querying large RDF spaces with traditional query languages such as SPARQL is challenging as it requires a familiarity with the structure of the RDF graph and the names (URIs) of its classes, properties and relevant individuals. In this paper, we propose a complementary approach based on Vector Space Models (VSM), more concretely Random Indexing (RI) [1] for building a *semantic index* for a large RDF graph. Traditionally, a *semantic index* captures the similarity of *terms* based on their contextual distribution in a large collection of documents, and the similarity between *documents* based on the similarities of the terms contained in the documents. By creating a semantic index for an RDF graph, we are able to determine contextual similarities between graph nodes (e.g., URIs and literals) and based on these, between arbitrary subgraphs. These similarities can be used for any task where a ranked list of *similar* URIs/literals can be of use given an input term (URI or literal). One of these tasks is the automatically refining and enriching a SPARQL query, as we describe in this paper.

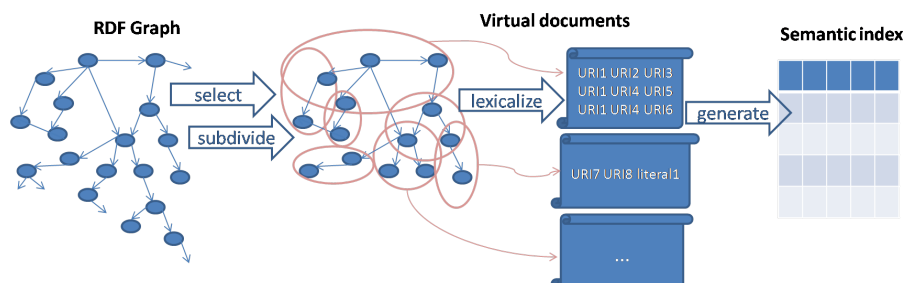


Fig. 1. Generating vectors from an RDF graph

In order to apply RI to RDF graphs, we first *select* the relevant part of the original graph and *subdivide* it into a set of potentially overlapping small subgraphs. The next step is *lexicalization* in order to create a set of *virtual documents* from these subgraphs. Finally, we generate the semantic index from the set of virtual documents (see Figure 1). The details of how each of these steps is performed significantly influences the final vector space model. For example, in the *selection and subdivision* step, all or just a part of the TBOX could be selected; the subgraphs could be individual *triples*, or *RDF molecules* (the set

of triples sharing a specific subject node), or more complex/bigger subgraphs. In the *lexicalization* step, the URIs, blank nodes, and literals of the triples from the RDF subgraph are converted to a sequence of terms. In the final step for generating the semantic index, different strategies for creating tokens and performing normalization have to be applied to typed literals, string literals with language tags, and URIs.

Once the semantic index has been created, it can be used to find similarities between URIs, literals, and RDF subgraphs. We use the ranked list of similar terms for literals that occur in certain kinds of SPARQL queries to make the query more general and also return results for entities that are semantically related to those used in the original query (see Figure 2).

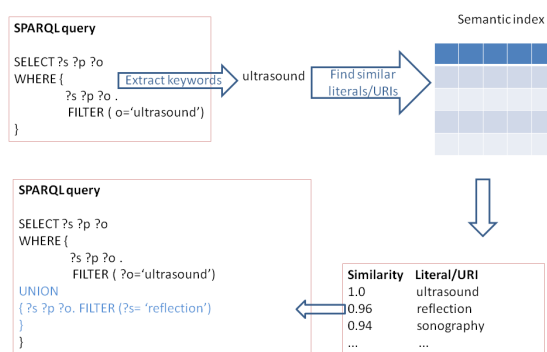


Fig. 2. Refinement of the SPARQL query using Random Indexing

We have applied our approach to parts of the Linked Life Data (LLD¹) RDF graph and are in the process of adding a search facility based on RI and RI-based SPARQL query enrichment to the LLD website. The code is available as a plugin of the LarKC platform from <http://sourceforge.net/projects/larkc/>.

Acknowledgments This research has been supported by the EU-funded LarKC² (FP7-215535) project.

References

1. Karlgren, J., Sahlgren, M.: From words to understanding. In: Uesaka, Y., Kanerva, P., Asoh, H. (eds.) Foundations of Real-World Intelligence, pp. 294–308. Stanford: CSLI Publications (2001)

¹ <http://linkedlifedata.com/>

² <http://www.larkc.eu/>



REFERENCES

- [1] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Knowledge Discovery and Data Mining*, pages 245–250, 2001.
- [2] S. Deerwester, S. Dumais, G. Furnas, T. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.
- [3] M. N. Jones and D. J. K. Mewhort. Representing word meaning and order information in a composite holographic lexicon. *Psychological Review*, 114:1–37, 2007.
- [4] K. Lund and C. Burgess. Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior Research Methods, Instruments, and Computers*, 28:203–208, 1996.
- [5] M. Sahlgren, A. Holst, and P. Kanerva. Permutations as a means to encode order in word space. In *Proceedings of the 30th Annual Meeting of the Cognitive Science Society (CogSci'08)*, Washington D.C., USA, 2008.
- [6] M. Sahlgren and J. Karlgren. Buzz monitoring in word space. In *European Conference on Intelligence and Security Informatics (EuroISI 2008)*, Esbjerg, Denmark, 2008.
- [7] L. Sitbon and P. Bruza. On the relevance of documents for semantic representation. In *Proceedings of the 13th Australasian Document Computing Symposium*, Hobart, Australia, December 2008.