# Toward Automatic RDF Property Tagging

Author: Andrea Dessi

Supervisor: Maurizio Atzori

Department of Mathematics and Computer Science
Ph.D. school in Computer Science
Ph.D. Coordinator: G. Michele Pinna

University of Cagliari
S.S.D. INF/01

A thesis submitted for the degree of

*Doctor of Philosophy*

Cycle XXVIII
Final examination academic year 2015/2016

“When ones expectations are reduced to zero,
one really appreciates everything one does have.”
(Stephen William Hawking)

# Acknowledgements

First and foremost, I would like to thank my Ph.D. supervisor Maurizio Atzori, who has been a model and an inspiration for providing me with the opportunity to complete my Ph.D. thesis at the Penn State University as a research assistant and as a visiting student. I want to also thank my on the spot supervisor Anna Cinzia Squicciarini and my colleagues Emanuele and Andrea, who helped and supported me.
These last years have been intense and stimulating, both professionally and humanly.
Finally, I would like to thank my family for all their strength and encouragement, who lived with me throughout this journey. For my parents who raised me with a love of science and supported me in all my pursuits. To all my friends, who encouraged me to take the Ph.D. challenge. And most of all for my loving, supportive, encouraging, and patient girlfriend Zuleica whose faithful support during this Ph.D. is so much appreciated. Thank you.

Andrea Dessi gratefully acknowledges INPS for the financial support of his PhD scholarship (Doctor J, Homo Sapiens Sapiens Operational Programme 2012).

Andrea Dessi
University of Cagliari
February 2017

# Publications

The research reported in this dissertation has been contributed by the following publications:

- Andrea Dessi, Andrea Maxia, Maurizio Atzori, Carlo Zaniolo: Supporting semantic web search and structured queries on mobile devices. (SSW@VLDB 2013: 5:1-5:4).
- Maurizio Atzori, Andrea Dessi: Ranking DBpedia Properties. (Wetice 2014: 441-446).
- Andrea Dessi, Maurizio Atzori: Computing On-the-Fly DBpedia Property Ranking. (ICSC 2014: 260-261).
- Andrea Dessi, Andrea Maxia, Maurizio Atzori, Carlo Zaniolo: Supporting Semantic Web Search and Structured Queries on Mobile Devices. (SEBD 2014: 361-368).
- Andrea Dessi and Maurizio Atzori: Schema-Agnostic Ranking of RDF Properties. (SEBD 2015: 104-111).
- Andrea Dessi, Maurizio Atzori: A machine-learning approach to ranking RDF properties. (Journal Elsevier, Future Generation Comp. Syst. 54: 366-377 (2016))

# Acronyms

| | |
|---|---|
| **RDF** | Resource Description Framework |
| **RDF/S** | Resource Description Framework/Schema |
| **XML** | Extensible Markup Language |
| **HTML** | Hyper Text Markup Language |
| **OWL** | Web Ontology Languages |
| **SBE** | Search By Example |
| **URI** | Uniform Resource Identifier |
| **SPARQL** | SPARQL Protocol and RDF Query Language |
| **SQL** | Structured Query Language |
| **URI** | Uniform Resource Identifier |
| **KR** | Knowledge Representation |
| **KB** | Knowledge Base |
| **QBE** | Query By Example |
| **IR** | Information Retrieval |
| **IE** | Information Extraction |
| **QA** | Query Answering |
| **MLR** | Machine Learning to Rank |
| **NLP** | Natural Language Processing |
| **QALD** | Question Answering over Linked Data |
| **LARQ** | Lucene + ARQ |

# Abstract

This work investigates some problems about semantic properties (also known as predicates) of Knowledge Bases, as part of the Semantic Web, for querying and ranking them toward a new system to tag automatically RDF Property over parts of free-text in Natural Language.
The main insights and contributions are:

- a contribution to develop a system called *Qpedia*, inspired by *SWiPE*, to make difficult query on schema-agnostic Knowledge Bases with a simple and intuitive mobile-user interface;
- the creation of the first approach exploiting Machine-Learning to rank RDF predicates;
- the creation of a possible approach to tagging free-text with RDF predicates, with a case study of possible backend;

The proposed methods have been evaluated with the most popular Knowledge Bases (DBpedia, WikiData, MusicBrainz and Freebase), obtaining encouraging results. Thus, this work is a first step towards the RDF Property Tagging of natural language, as reflected in Chapter 5, needed to pave the way providing a resolution of sub-problems related to Question Answering over RDF properties, which are not typically addressed in literature through this way.

**Keywords**

Semantic Web, DBpedia, Ranking Algorithms, Graphical User Interface, Human Computer Interaction, Fast Property Ranking, Tagging, User Experience.

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 The Context

The World Wide Web is the greatest repository of information, with virtually unlimited potential. It is made from millions of interlinked web pages and contains resources concerning almost every imaginable topic, instantaneously available to anyone with an Internet connection.

However, its size has also become one of big research problems. Due to the volume of available information, it is becoming increasingly difficult to locate useful information. Furthermore, users often want to use the Web to do more than just locate a document, they want to perform some special purpose task. For example, a user might want to find the best answer on a specific question.

The main obstacle results that the Web was not designed to be processed by machines. Thus, to process a web page intelligently, a computer must understand the text, but natural language understanding is known to be another extremely difficult and unsolved research problem. Tim Berners-Lee, inventor of the Web, has coined the term Semantic Web [10] to describe this approach.

The exact definition according to him of this concept, is

*"The Semantic Web is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."*

The Semantic Web is impacting on a number of fields, showing huge potential of having the Web as a collaborative space for storing and querying structured data in a decentralized way where everyone can access and contribute.

The Semantic Web consists primarily of three W3C technical standards:

- Resource Description Framework [39] (RDF), the data modeling language for the Semantic Web, where the Semantic Web information is stored and represented in this format.
- Web Ontology Language [8] (OWL), the schema language of the Semantic Web, which enables you to define concepts carefully defined, called Ontologies, so that these concepts can be reused as much and as often as possible.
- SPARQL Protocol and RDF Query Language [51] (SPARQL), the query language of the Semantic Web, which adds querying capabilities to RDF;

In this thesis I will examine some of the problems that directly regards it, in particular searching, ranking and tagging over it, and I will try to solve them with some original proposals in an efficient way. The following Section 1.2 provides more details about theoretical background of Semantic Web and its technologies.

## 1.2 Semantic Web overview

This section describes a brief introduction into the background technologies used. As I said, these technologies are standards and W3C recommendations, the basic building blocks of the approach presented in this thesis is based on.

### 1.2.1 Knowledge Bases used

A knowledge base (KB) is a machine-readable resource and in fact a centralized repository used for the dissemination of information (knowledge) and management. It contains a set of concepts, instances, and relationships. Over the past decade, numerous KB have been built, and used to power a growing array of applications. For example a public library, a database of related information about a particular subject. Islands of RDF, technology described in the following subsection 1.2.2, and possibly related ontologies form a Knowledge base. Well-known examples of KB include DBLP, Google Scholar, Internet Movie Database, YAGO, DBpedia, Wolfram Alpha, MusicBrainz and Freebase. In recent years, numerous KBs have been built, and the topic has received significant and growing attention, in both industry and academic areas.

### 1.2.2 RDF

RDF is the data-model for representing metadata in the Semantic Web. RDF triples representing facts, and made of entities, properties and values. In RDF is possible to express the meaning of fact unambiguously. The RDF data-model is based on subject, predicate and object triples, so called RDF statements, to formalize meta-data. RDF is domain independent in that no assumptions about a particular domain of discourse are made. It is up to the users to define their own ontologies for the users domain in an ontology definition language such as RDF/Schema (RDF/S), which defines the vocabulary used in the RDF data-model. Ontology is the core of the Semantic Web, which is used to explicitly represent our conceptualizations. In the RDF data-model the statements are represented as nodes and arcs in a graph. In this notation, a statement is defined as:

- a node for the subject ($s$)
- an arc for the predicate ($p$)
- a node for the object ($o$)

Thus, a triple can be graphically represented by two nodes ($s$ and $o$) and a directed edge (representing the $p$) from the subject to the object node. A collection of RDF triples forms an RDF graph. Before we are able to express the fact above as RDF statement we have to introduce the concept of a resource which is identified by a Uniform Resource Identifier (URI). Most of these elements are represented as URIs ($U$), forming a huge graph sometimes referred to as Linked Data. Each data publisher provides a part of the Semantic Web graph, and through endpoints, these subgraphs can be easily queried by means of an effective pattern-based query language, the well-known SPARQL. Linked Data and the number of triples it is composed by is experiencing a continuous growth in recent years. Semantic web is organized to form a huge distributed knowledge based system. A knowledge base is a database used for knowledge sharing and management. In this thesis the mainly work is based on four Knowledge Bases well-known like DBpedia, Wikidata, Freebase and MusicBrainz, which we will see after.

In a model, all components will be modelled in the following way. Given a set of URIs $U$ and a set of literals $L$, an RDF triple is defined whit the known short notation as $\langle s, p, o \rangle$, where $s \in U$ is the subject, $p \in U$ is the predicate (or property), and $o \in (U \in L)$ is the object. Exists the case in which $s$ and $o$ are blank nodes but their usage is discouraged (Heath and Bizer 2011).

For instance, statements can be represented as a graph in RDF as we said. Consider a simple example as "Barack Obama is the leader of the United States" using DBpedia. This sentence has the following parts

- $s$: `http://dbpedia.org/page/Barack_Obama`
- $p$: `http://dbpedia.org/ontology/leader`
- $o$: `http://dbpedia.org/resource/United_States`

The same concept is possible to define as

- $s$: `http://dbpedia.org/resource/United_States`
- $p$: `http://dbpedia.org/ontology/leader`
- $o$: `http://dbpedia.org/page/Barack_Obama`

In this example, the ontology to express the fact above has to define the concept of a "United States" and the relationship "leader" in its vocabulary. The figure 1.1 shows a possible representation with nodes and arcs.

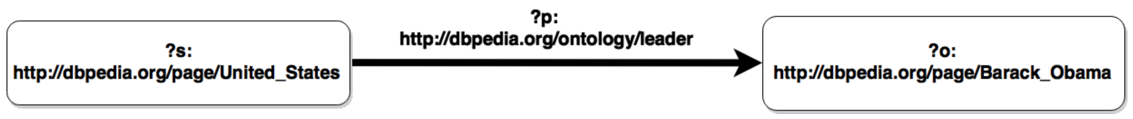

?s:
http://dbpedia.org/page/United_States
?p:
http://dbpedia.org/ontology/leader
?o:
http://dbpedia.org/page/Barack_Obama

Figure 1.1: A RDF triple sample

Another example would be "Barack Obama was born on 1961-08-04". In this case the concept is defined as follow

- $s$: `http://dbpedia.org/page/Barack_Obama`
- $p$: `http://dbpedia.org/ontology/birthDate`
- $o$: `1961-08-04`

where the object $o$ is a constant.

### 1.2.3 DBpedia

DBpedia [7] is a well-known semantic web project focused on extracting structured information from Wikipedia and making triples available as free datasets, had in the last year a 7.5 edition, from 4.26 to 4.58 million. In the same period of time, properties describing those entities increased of 8.2 passing from 51,736 to 55,986 raw properties. Similar growth have been reported in other public Knowledge bases, such as Freebase which now contains ten times the entities of DBpedia and a total of 2 billion triples. Recapping the structure of an example[1] about DBpedia Uri we have:

- the first part "dbpedia.org"
- an entity (i.e. Rho)

DBpedia is backed by the Virtuoso triplestore[2] and it is available through W3C standards for the Semantic Web and it stores its data as Resource Description Framework Schema (RDF/S) triples [45]. The DBpedia dataset has been extracted from Wikipedia and currently has more than 3.77 million "things" with 400 million facts. It also features labels and short abstracts in 15 different languages, 588,000 links to images and 3,150,000 links to external web pages.

### 1.2.4 Freebase

Freebase is a Web-based database that allows you to create and edit data entries for any entity of general interest. We can also say, like a graph database, Freebase uses several sources to provide broad coverage. Freebase contains about 22 million entities and 390 million facts in more than 100 domains. It has two main advantages compared to Wikipedia. First, it has rich types and well defined schemas for the entities, thus is considered more as a structured database. Second, it contains a lot more entities. According to the current statistics on each site, Freebase is several times bigger than English Wikipedia. The structure of an example[3] about Freebase Uri is the following:

- the first part "freebase.com"
- an id (i.e. /m/02mjmr which corresponds to entity Barack Obama)

[1] `http://dbpedia.org/page/Rho`
[2] `http://virtuoso.openlinksw.com/`
[3] `http://www.freebase.com/m/02mjmr`

### 1.2.5 Wikidata

Wikidata is the community-created knowledge base of Wikipedia, and the central data management platform for Wikipedia. The goal of Wikidata is to overcome some problems. For instance in Wikipedia, the same information often appears in articles in many languages and on many articles within a single language. Population numbers for Rome, for example, can be found in the English and Italian article about Rome, but also in the English article Cities in Italy. All of these numbers are different. How to solve these problem? By creating new ways for Wikipedia to manage its data on a global scale. The result of these ongoing efforts can be seen at its site[4]. The structure of an example[5] about Wikidata Uri is the following:

- the first part ”wikidata.org”
- an id (i.e. Q1897 which corresponds to entity Cagliari)

Wikidata uses for its properties a string with an increasing number (i.e. Property:P17 that corresponds to property country). A complete SQLite database was carried out which contains all of mapping between property plus code and label.

### 1.2.6 MusicBrainz

MusicBrainz is an open source community-maintained database of music information project that provides a wealth of crowd-sourced structured data about music. In this knowledge base there are all of the various pieces of information collected about music, from artists and their releases to works and their composers, and of course much more. MusicBrainz was founded as an open system that allows registered users to update and edit the database. The structure of an example[6] about MusicBrainz Uri is the following:

- the first part ”musicbrainz.org”
- the category (i.e. artist)
- an id (i.e. 0de4d19f-05c8-4562-a3c0-7abdc144f1d5 which corresponds to entity Barack Obama)

[4]wikidata.org
[5]`http://www.wikidata.org/wiki/Q1897`
[6]`https://musicbrainz.org/artist/0de4d19f-05c8-4562-a3c0-7abdc144f1d5`

Here it is possible to choose many different categories like Artist, Release Group, Release, Recording, Work, Label, Area, Place, Annotation, CD Stub, Editor, FreeDB, Tag, Instrument, Series, Event and Documentation.

### 1.2.7 SPARQL

SPARQL is the W3C language which allows to query for triples from an RDF database (or triple store). It features a set of constructs very similar to those provided by Structured Query Language (SQL). A triple store stores only triples, and it permits to pile the triples while describing a thing. As mentioned above RDF uses URIs, having the potential to link to any other data in any triple store. SPARQL uses RDF graphs expressed in Turtle syntax as query patterns and can return as output variable bindings (SELECT queries), RDF graphs (CONSTRUCT and DESCRIBE queries) or yes/no answers (ASK queries). SPARQL has already been proved to be as expressive as relational algebra.

Anatomy of a Query is composed to three parts:

- prefixes[7]
- select dataset and the query patterns
- modifiers

An example of query SPARQL in DBpedia to find what is the birth date of Barack Obama is the following:

```
select ?o
where {dbpedia:Barack_Obama <http://dbpedia.org/ontology/birthDate> ?o}
```

## 1.3 Contribution

This thesis aims at facing challenges in the context of Question Answering, through the instruments described above, starting from three parts of studies, being each part of study devoted to cope with a specific part of solution proposed in line with recent advances in this field.

The first part of study focuses on the task of unearthing and manipulating information from knowledge bases, each having its own organization, terminology and

[7] `http://prefix.cc/`

data formats in order to provide user-friendly mobile graphical interface for accessing and querying the above resources and smartly exploring their content. The study explores the potential of Search By Example paradigm as an enabling technology to understand how to obtain new information from existing information. Specifically, it presents Qpedia, which allows for searching information made available by public open databases.

The second part of study regards RDF properties with special focus on Machine Learning To Rank, where properties from the most famous open Knowledge Bases, are ranked through features specially made to give them an useful order. The study presents a service application called RankProp which deals with supporting users in the choice of the semantic properties on the basis of context.

Finally, the third part of study concerns the tagging properties from sentence in natural language. The study investigates a possible solution to tag text with RDF properties, and not only. In particular, the study faces the problem of question answering indirectly.

Almost all the approaches at the state of the art, used to tag and rank sentences, are basically based on Entities, without taking into account any factor of Properties. This thesis sets out to give two contributions summarized as follows. First, the potential of SPARQL queries and ranking properties has been evaluated for developing applications that support the access to knowledge bases and the easily to query them by improving presentation of results and fostering users interaction to find new information.Second, the positive impact of tagging properties to identify the important parts of sentences in natural language enriching them with the most probable properties through the ranking system. Experiments are presented and results are then analyzed in order to draw guidelines about how to reduce the above contribution.

## 1.4 The Approach

The research performed in the Ranking and Tagging RDF properties, focuses primarily on the fundamental QBE, MLR and NLP processes that allow the querying and sorting of properties and the identification of them in natural language sentence. The approach taken is generic, as it addresses the interplay of Mobile User Interfaces, heterogeneous knowledge bases, and sentences in natural language (English), and involves in addition, the use of advanced NLP tools. The results of this research are

even of immediate interest to many scientific areas. Figure 1.2 shows how interact the parts of thesis between them. RankProperties is the core of system because is used by TagProp and Qpedia. TagProp is the aim and Qpedia basically is an application which exploits the above methods. The experience obtained in this research allows researchers to advance new theories in semantic web scenarios, as well as in other fields including Question Answering, in which there is at the base also interplay of similar and/or analogous processes.
In the next three subsection is illustrated a summary of the major components of this thesis.

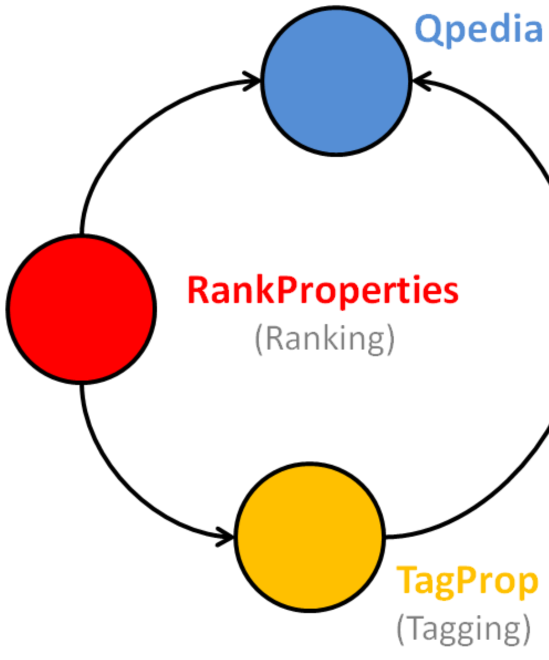


Figure 1.2: Heart of Thesis

### 1.4.1 Query By Example: Qpedia

The first work proposed regards a novel cross-platform system called QPedia which supports querying SPARQL endpoints dynamically without previous knowledge of web semantics from a mobile device. The aim is to address for the first time the problem of accessing and querying semantic web data coming from any endpoint (not necessarily DBpedia, and no assumptions on the schema or the content), using the search by example approach in [5] and adapting it to mobile devices. This work is therefore motivated [60] by the need for an easier way of using semantic web resources, such as DBpedia, for casual users accessing from a mobile device, therefore with a small screen, no proper pointing device and without knowledge of the ontology behind the Semantic Web. Such proposal and its related prototype QPedia described in the next chapter introduce a novel approach to display, query and interact with the Semantic Web from the mobile using well-known gestures, voice recognition, a simple way of introducing constraints and enabling location-based queries based on the user position.

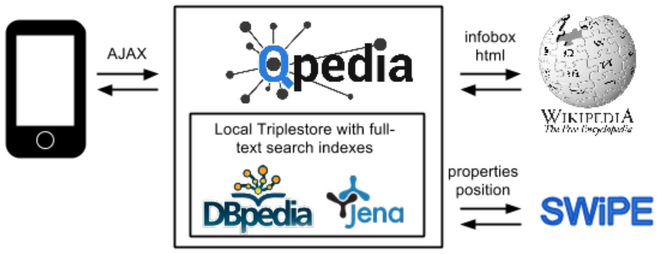


Figure 1.3: The QPedia System

## 1.4.2 RankProperties

RankProperties is the core of this thesis where it analyses the problem of computing the ranking of entity properties in a fast and effective way, where the ranking is personalized depending on the entity viewed by the user. A general property ranking, not conditioned by a given entity, is also feasible with this approach. In Chapter 5 I will propose a number of specifically designed numerical features that measure different aspects of each property, two of these are new compared to [4, 19]. Then, by using a supervised machine-learning approach, an existing learning-to-rank (MLR) algorithms to a number of classified properties has been applied, automatically constructing ranking models that reflect a given classification. The proposed features are easily computable on the fly, allowing the application of a previously-learned ranking model to any query result or to an entity. As thoroughly shown in the previously chapter, the problem of ranking properties seems not specifically taken into account by the large literature on RDF ranking, which focused on sorting entities and queries instead. Figure 1.4 illustrates RankProperties framework and the process of modeling, evaluating and ranking organized in three main tasks (denoted by {task}).
In order to run the experiments we accomplished the following steps:

I. It has been implemented implemented a service {3} that given an optional entity automatically computes the feature metrics discussed in the previous section –

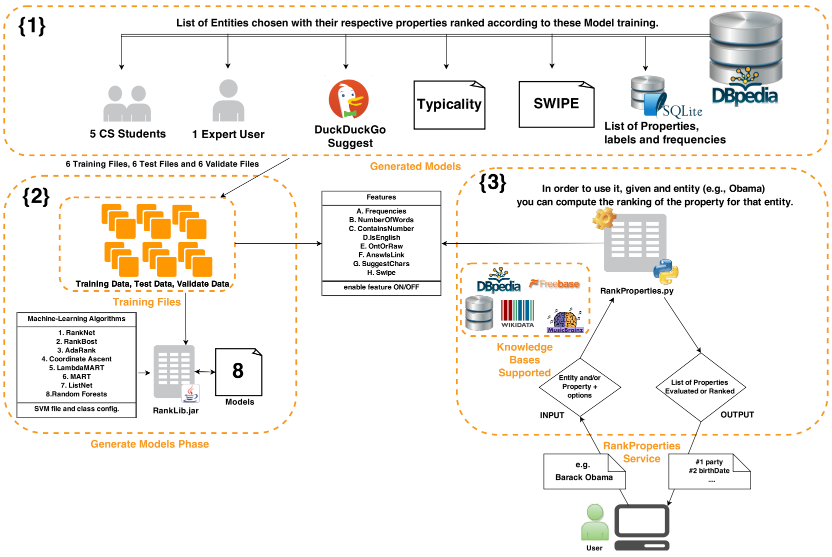


Figure 1.4: The Ranking RDF Properties framework

including a tool in Python based on the *Natural Language Toolkit (NLTK)*[8];

II. by using that service, features values have been computed for a number of properties, created all necessary input files {1} (training, test, and validate set) and generated a model for each MLR algorithm {2};

III. MLR-based evaluation: it has been evaluated all machine-learned ranking models against real RDF properties.

[8]`http://www.nltk.org/`

### 1.4.3 TagProp

TagProp is the last work of this thesis where it analyses the problem of computing the tagging of entity properties, given a sentence in natural language, in a fast and effective way. Previous works described and in particular RankProperties, is a key part of the tagging result. This is because when performing a search of all properties, it is possible to obtain them more than one, for this it is necessary to have a ranking of them. In Chapter 5 I will propose a possible engine devised with a graphical user interface that permit you to obtain the best results easily. Figure 5.2 illustrates an example of how TagProp works.

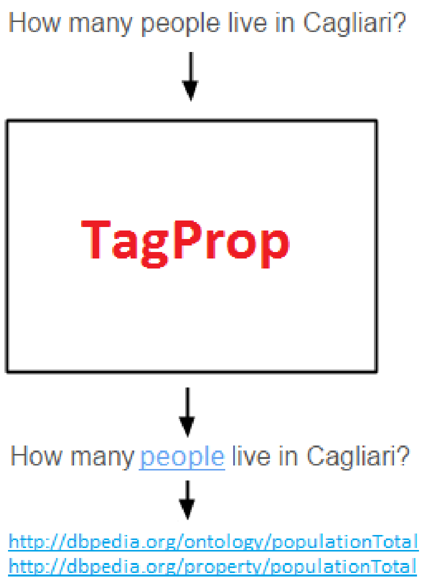

How many people live in Cagliari?
TagProp
How many people live in Cagliari?
http://dbpedia.org/ontology/populationTotal
http://dbpedia.org/property/populationTotal

Figure 1.5: TagProp System

The proposed solution is novel and still performing work. Currently, a service that given a natural language sentence automatically computes the possible relevant properties has been implemented. In Appendix B it is possible to follow instructions to install this system. The problem of tagging properties could be very interesting to improve Question Answering systems. Then, starting from a sentence in natural language, assuming to have its entities, and assuming to have its properties with TagProp, it is possible by exploiting the structure and nature of RDF, to be able to reply to the questions in a natural language.

## 1.5 Thesis Structure

After describing the current situation of the Semantic Web and discusses the components, standards, and technologies used in this context with a summary description of contributions, now on to the hearth of matter.
Chapter 2 presents a summary and the state of art of Querying, Tagging and Ranking RDF Properties.
Chapter 3, 4, and 5 will show the chosen methodologies to these critical questions, and Chapter 6 how they have been resolved them describing the main experiments.
Chapter 7 concludes the thesis and gives an outlook on future work.
Appendix A contains information about how to install and configure RankProperties tools, providing interesting technical details and useful examples.
Appendix B describes technical details about TagProp tools, with a step by step guide on how to install these tools.
Appendix C is useful to give definite indications to install Qpedia.
Finally, Appendix D proposes a series of experiments were performed to determine the mechanism and performance about Jena backend.

# Chapter 2

# State of the Art

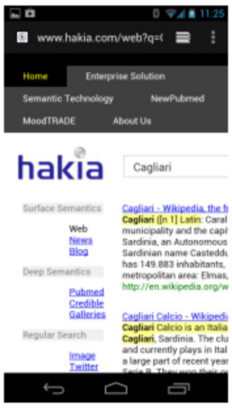
This chapter overviews the background material of the thesis starting from the studies and improvements on the topic of Semantic Web, than showing existing framework and application for querying, ranking and tagging about it. The chapter is organized as follows: Section 2.1 overviews the frameworks present in literature which address the search systems for Knowledge Bases. Section 2.2 gives and overview of the most known systems for ranking problem and finally Section 2.3 describes the few frameworks present to problem about tagging Natural Language with Knowledge Base components.

## 2.1 Search Systems

The state of the art of searching the web is defined largely by the capabilities and shortcomings of the various available search engines. Currently the discussion of the future of searching the web is dominated by the term semantic web. The concept is based on annotated metadata (XML/RDF). For a general approach of searching using semantics into the web search, new methods are emerging, but the mainly focus of this section regards mobile world. The advent of smart phones and thus mobile computing confirm that the future of the Web is to create more transparency and simplicity, to allow an easy use though there exist problems such as low interoperability with the devices, small screens and more. In parallel, the recent evolution of the web, namely the Semantic Web, is growing rapidly, and contains a large amount of data and knowledge. The challenge thus will be to join Semantic Web technology and the mobile world to provide new additional supports for knowledge-based, location and context-aware information. For all the work an excellent testing ground it was DBpedia. There has been a number of useful web interfaces to navigate and query DBpedia and they are discussed subsequently. Unfortunately they are based on interfaces that require a standard monitor and mouse, handling specific user events such as the `mouseover` event. Fig. 2.1 shows how four existing interfaces are given on a recent smartphone screen, drastically reducing the usability on such devices. The existing proposals, such as SWiPE [5], Faceted Wikipedia Search [33] (Fig. 2.1a) and Virtuoso Faceted Web Search (Fig. 2.1b), allow users to ask complex queries only with a desktop user interface. In more detail SWiPE generates automatically semantic queries for DBpedia using the Search by Example approach, helping people who do not have knowledge about SPARQL to pose their desired query. The system provides an interface like Wikipedia which has, on the infobox, editable fields to input the query. The user can choose which fields to modify in order to start a new query using shown information about the underlying related DBpedia page. Another example of semantic web search engine is Hakia (see Fig. 2.1c), that brings relevant results based on concept match rather than keyword match or popularity ranking. A few others try to address the problem of making web semantic data useful in a mobile context, such as DBpedia Mobile [9] (Fig. 2.1d) , that provides a map view annotated with DBpedia entities and information from other knowledge bases. This application, based on geographic location, generates a map that contains information of the surrounding locations contained in the DBpedia dataset. It works on desktop browsers, while for mobile devices, the application is optimized for QVGA display (320x240 pixels) therefore not specifically focused on current devices (featuring full HD displays). Other than being designed for low-resolution screens, DBpedia mobile is a system that tackles only a specific search need, by focusing on locations. Therefore it is not addressed to the general problem of accessing and querying large datastore of (possibly) unknown domains. In Chapter 3 will be described in detail a solution, called Qpedia [20], to solve the problem of the state of art, which is an important part of this thesis regards semantic system search.

(a) Faceted Wikipedia Search

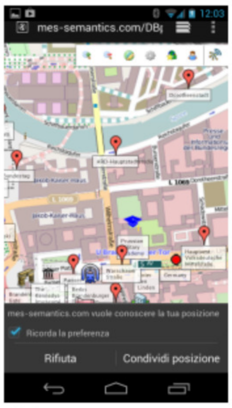
(b) Faceted Search and Find service



Cagliari - Wikipedia, the f
Cagliari ([n 1] Latin: Caral
municipality and the capi
Sardinia, an Autonomous
Sardinian name Casteddu
has 149.883 inhabitants,
metropolitan area: Elmas,
http://en.wikipedia.org/w

Cagliari Calcio - Wikipedi
Cagliari Calcio is an Italia
Cagliari, Sardinia. The clu
and currently plays in Ital
a large part of recent yea

(c) Hakia



(d) DBpedia Mobile

Figure 2.1: Existing approaches rendered on a recent mobile browser

## 2.2 Ranking Systems: entities and properties

As mentioned above, the Semantic Web is impacting on a number of fields, showing huge potential of having the Web as a collaborative space for storing and querying structured data in a decentralized way where everyone can access and contribute. Since many Semantic Web searches through SPARQL queries look for interesting entities, much work has been done that deals with the problem of ranking entities, but very few work faced the problem of ranking the properties of a given entity. While many triples contains useful information for each entity, a large amount of them may be insignificant for some applications, and users are therefore overwhelmed with irrelevant data. For instance, on DBpedia 2014 a user looking for the city *Rome* will be in front of 601 RDF triples containing 164 distinct properties. Supposing she is interested in the number of people living in the capital of Italy, she will find difficulties reaching the appropriate `populationTotal` property, since the list also contains many other unuseful attributes such as `wgs84_pos#geometry` or `wikiPageID`.

In such frequent scenarios, the user experience degrades and casual users are not able to easily find the desired information. In many Semantic Web applications, including the HTML pages of DBpedia's entities and Qpedia, the user is shown with all the triples of a predefined entity. The list of these applications included from semantic browsers and faceted navigators, entity viewers such as the DBpedia and DBpedia Live [41] HTML representations of each entity, semantic Knowledge base aggregators such as IBKB [49, 50], mobile semantic querying tools such as Qpedia. The user usually focuses on some particular information about a resource, but she gets overwhelmed by plenty of results generated by those systems. For instance, there are many results after querying those interfaces by only providing a common keyword. Even when the set of resources is determined and small, the number of attributes to deal with is still too large. In such frequent scenarios, the user experience degrades and casual users are not able to easily find the desired information. With the notable exception of IBKB, the default is sorting attributes and values in lexicographical order by the attribute name. Even the new advanced DBpedia interface[1], which allows instant keyword searches on property names, is of little help considering that users usually are unaware of the exact attribute names. A feature that highlights the most important attributes of the entity at hand is missing. An original approach to evaluating importance, focusing on data quality, is the one described in [56], where the system *"WhoKnows?"* exploits collaborative, crowd-sourced reviews of RDF data through on a quiz game based on DBpedia data. The paper is focused on data cleansing, that is, detecting inconsistencies and doubtful facts that may arise because of misspelled human-provided data, faulty text parsing and other buggy automated methods. Although quite different on purposes w.r.t. my work, the paper contributes an evaluation of property relevance heuristics on DBpedia data, provided by the game players. The ranking of properties may change depending on the context (a DBpedia class), for instance the attribute `keyPerson` is ranked *2nd* for the DBpedia ontology class *Company*, but only *6th* if related to the ontology class *Organisation*. The approach is definitely interesting, although the methodology limits its applicability since it requires a large community of players and an ad-hoc personalized game if used in other contexts different from DBpedia. Instead, a completely automatic system has been developed that only requires some classified instances to

[1]available, for instance, on `http://live.dbpedia.org/page/Rome` by clicking on the right yellow corner

learn how to rank properties. It should be noted that the crowd-sourced classification could be used as a useful feature in our framework, taking advantage of both approaches.

The work in [21] describes a novel navigation model, introduced in the *Swoogle* Semantic Web search engine, that supports ranking based on the data quality of RDF data. It proposes ranking ontologies at various levels of granularity to promote reusing ontologies, and introduces a the *OntoRank* algorithm which is based on the rational surfer model, emulating an agents navigation behavior at the document level. The work appears to be mainly focused on classes and ontologies, while the last part defines the *TermRank*, an approach to rank ontology classes based on the so-called *class-property bond*, that is, relations between classes and predicates. In [36] an algorithm inspired by the PageRank provides a scalable ranking scheme for RDF datasets. The work takes advantage of property values to rank entities, each forming a subgraph with properties and values. According to authors, the work did not conduct an extensive evaluation of the quality of the ranking results, while providing a step towards a unified RDF ranking scheme. With respect to our work, they do not focus on RDF property ranking. Also work in [32] focuses on ranking entities, as well as [35], that proposes a way to sort Semantic Web resources based on importance, relevance and query length, and provides an *overall ranking* that considers all these three dimensions together. Another approach that takes account of different dimensions for the ranking is the *DBpediaRanked* in [48], where measures regards: the graph-based nature of the underlying RDF structure, the context independent semantic relations in the graph and the external information sources such as classical search engine results and social tagging systems. It is an advanced system that exploits external Web sources such as Google, Yahoo, Bing, Delicious and Wikipedia in order to reduce query results based on entity ranking. Unfortunately, the methods do not seem to be applicable to the problem addressed in our paper, namely the ranking of RDF properties. In [65] authors focus on ranking very large number of entities exploiting the resulting graph and entity relevance based on search engines. One of our features is also based on exploiting search engine results, plus our novel prefix-count suggest feature, but again, that's being used these source of metrics to sort predicates.

In [23] authors focus on a research area that aims to avoid overwhelming users by ranking the results of a SPARQL query. In particular, they develop a novel form of language models that allows both structured and keyword search, extending ranking measures, that are already well-defined for keywords search only. In order to obtain RDF knowledge ranking the work proposes a generalization of entity Language Models that considers relationships (RDF properties) as first-class citizens. The use of properties is therefore instrumental to rank results of mixed structure and keyword-based queries. The problem of ranking the very properties is not taken into account. Also in [58] solutions to combine keyword search and structured querying are provided. Ranking of results is a feature provided by their methods, in contrast to database-oriented structured approaches such as those using only SPARQL.

In [12] an adaptation of the BM25F ranking function for RDF data is presented. The work demonstrates that this adaptation outperforms existing methods in ranking RDF resources. Their algorithmic contribution is based on novel indexes that supports efficient retrieval of ranked RDF data. The BM25F scoring measure is used within a method where it is possible to assign different weights to different predicates. This contribution provides a very fast way to retrieve ordered (according to the metrics) RDF results, under the assumption that the query language has some weaker expressive w.r.t. SPARQL. Anyway, it is not clear how to extend those results to rank properties only. In particular, in order to evaluate the effectiveness authors rely on weights for properties (important, unimportant, neutral), but then they state that "it is future work to look at how is it possible automatically learn these lists, i.e. based on the likelihood of the fields matching in relevant documents or domains vs. the likelihood of matching in irrelevant documents or domains". Our work mainly addresses this specific need, that is, the definition of property features automatically computable from the data, therefore having an impact and helping the contribution in [12] to be fully automatic.

Work in [1] applies previous work on inferring the information value given a graph. In particular, they use RDF Graphs as the input graph, modeling and computing impact and trust for any piece of information, with the purpose of improving web ranking. Therefore, RDF data is an instrument to obtain information value with the final aim of constructing a Web re-ranking system that personalizes the information experience of Web users.

Authors of [18] consider a learning to rank approach, as the one in this paper. The features used in that work are anyway dependent on the different setting, since they want to measure the relevance of entities: number of subjects, number of objects, ingoing and outgoing predicates and their average frequencies, number of literals. While using frequencies and counts, these features are defined on entity nodes only, not on properties.

The work in [38] addresses the problem of providing more results to a query, by relaxing its constraints with a novel RELAX clause. Results are sorted by closeness/exactness w.r.t. the original query. The ranking approach of this work seems to be centered on the relaxed queries, therefore not applicable to our problem setting. Also [22] proposes a non-exact approach to answering structured queries, proposing a language-model-based approach to ranking the results of exact, relaxed and keyword-augmented graph-pattern queries over RDF graphs such as entity-relationship graphs. The ranking model is based on the Kullback-Leibler divergence between the query language model and the result-graph language model. The work in [2] defines the problem of querying for semantic associations. Here the ranking is provided by the user as a soft constraint on which associations (between nodes) she is looking for [3].

In [29] three dimensional tensors (basically multiple adjacency matrices, one for each attribute) are used together with HITS and PARAFAC tensor analysis, developing a 3-step offline system called *TripleRank*. It allows predicates to be grouped together improving user experience, showing, e.g., that these predicates `http://dbpedia.org/property/genre` and `http://dbpedia.org/ontology/genre` usually contain similar data. Experiments show the feasibility for faceted navigation, although also similarity search can benefit from the tensor-based method. This can be considered a clustering approach that can be used within our framework, for instance by considering similar predicates as if they were the same, aggregating the feature metrics accordingly. We plan to investigate how TripleRank and our learning to rank approach can interact and enhance each other.

The work in [42] computes the RankScore values of resources by applying a top-k dominating model. In [63] big data processing is studied, investigates a variety of techniques and theories from different fields, including data mining and machine learning, information retrieval and massive processing.

Authors of [37] propose a Citation Semantic Link Network (C-SLN) to describe the semantic information over the literature citation networks. As in our proposal, they use NLP methods and other techniques. However, the focus of the work is about discovering opinion communities in a C-SLN and finding articles of high importance.

[34] proposes the ranking of web services, based on the exploitation of textual descriptions. It defines service relevance and service importance, providing techniques for ranking results, which are references to web services. The work does not specifically addresses semantic data and in particular static RDF triple datasets.

The work in [59] proposes an improved social-network based reputation ranking algorithm, called Poisonedwater, to compute accurate reputation ranks of social network related entities.

About Property Ranking, two works in literature specifically have addressed the problem of ranking RDF *properties*, not only entities. Those recent work has been done by the projects IBminer [50] and Typicality [40].

IBminer [50] is a text-mined RDF dataset where properties of each entity can be sorted by accuracy, significance and relevance. Their approach is based on hard coded weights assigned to the source of the data. Users may modify these weights providing a rating. It was developed to generate and verify structured information, and to reconcile terminologies across different knowledge bases. In particular it is interesting the tool designed to support the integration process in close collaboration with IBminer, called InfoBox Knowledge-Base Browser (IBKB)[2]. In this tool users can easily provide feedback on significance, correctness, or relevance of each summary item (RDF property), in their knowledge base. Thus users, through their feedback, provide the correct ranking of facts (that can be modeled as RDF triples), and therefore their properties. This approach has good performances in terms of precision, since ranking is provided by users. However, many properties may not be classified, leading to possible low recall, and in general it may not be applied to different KB, since ranking is provided for the facts and properties in the IBKB repository, without a way to generalize this information to other data, as it will be displayed in RankProperties. A different approach has been investigated by the work on Typicality. Its name is given by the fact that authors measure how typical each property is for a concept (such as an RDF class). They provide different kinds of typicality, that will be used in experiments later in Chapter 7. In particular the work focused on $P(c|a)$ which denotes how typical concept (ontology) $c$ is, given property $a$. Then, they compute for each property typicality score and they use this for ranking the same properties. Internally, Typicality employ Probase [62], a probabilistic knowledge base, as also IBminer does. This approach has the advantage of providing a plausible ranking in an automatic way, without user intervention. This is usable in some applications, such as automatic information extraction (the context typicality has been developed for), while others may require user personalization, such as in IBminer. In the fourth chapter will be explained the approach adopted which concerns the best of both previously approaches: a correct classification is exploited, that can be provided

[2]`http://semscape.cs.ucla.edu/mapper/ibminer.html`

by users or automatically and will be part of the training set, and then a machine-learning to rank algorithms to learn proper ranking from the given instances is applied, provided some automatically-computed features.

## 2.3 Tagging and QA Systems

First and foremost, there's been a particular interest in Tagging System, because is a good intermediate step to get to Question Answering over Linked Data. Question Answering (QA) is a fast-growing research area that brings together research from Information Retrieval (IR), Information Extraction (IE) and Natural Language Processing (NLP). The Question Answering system takes questions from natural languages as input and searches matching answer in set of documents and extracts the precise answer to natural language questions. It is different from information retrieval (IR) or information extraction (IE). IR system present the users with a set of documents that related to user questions, but do not exactly indicate the correct answers. The functioning of Tagging Systems could even be that of facilitate QA processing.

In [53] is described Ephyra, an open-source question answering system and its extension with factoid and list questions via semantic technologies. Using Wordnet as well as a answer type classifier to combine statistical, fuzzy models and previously developed, manually refined rules. The disadvantage of this system lies in the hand-coded answer type hierarchy which prohibits its multi-lingual use. In [17] is being developed ORAKEL to work on structured knowledge bases. The system is capable of adjusting its natural language interface using a refinement process on unanswered questions. GINO [11], allows users to edit and query ontologies in a language akin to English. It uses a small static grammar, which it dynamically extends with elements from the loaded ontologies. In [44] is introduced PowerAqua, another open source system, which is agnostic of the underlying yet heterogeneous sets of knowledge bases. It detects on-the-fly the needed ontologies to answer a certain question, maps the users query to Semantic Web vocabulary and composes the retrieved (fragment-)information to an answer. However, PANTO [57] accepts generic natural language queries and outputs SPARQL queries. Based on a special consideration on nominal phrases, it adopts a triple-based data model to interpret the parse trees output by an off-the-shelf parser.

In [15] is presented a demo of QAKiS, an agnostic QA system grounded in ontology-relation matches. The relation matches are based on surface forms extracted from Wikipedia to enforce a wide variety of context matches. Several industry-driven

QA-related projects have emerged over the last years. For example, DeepQA of IBM Watson [27], which was able to win the Jeopardy! challenge against human experts.

Just to go back to Tagging Systems, some important Semantic Web applications are related to tagging text with semantic URIs. While the vast majority of those tools find entities references, performing entity annotations (e.g., TagMe [25, 26] and SpotLight [46]), some other applications such as *Question Answering on Linked Data* [55] and *By-Example Structured Queries* featured by the SWiPE System [5, 6] perform tagging by linking text to *property*'s URIs, instead of entity's. DBpedia Spotlight is a tool for automatically annotating mentions of DBpedia resources in text, providing a solution for linking unstructured information sources to the Linked Open Data cloud through DBpedia. DBpedia Spotlight recognizes that names of concepts or entities have been mentioned (e.g. "Michael Jordan"), and subsequently matches these names to unique identifiers (e.g. dbpedia:Michael_I._Jordan, the machine learning professor or dbpedia:Michael_Jordan the basketball player). It can also be used for building your solution for Named Entity Recognition, Keyphrase Extraction, Tagging, etc. amongst other information extraction tasks. TagMe is a powerful tool that is able to identify on-the-fly meaningful substrings (called "spots") in an unstructured text and link them to a pertinent Wikipedia page in a fast and effective way. It is possible to perform automatic tests of this tool by using simple RESTful APIs exploited in TagProp. TagMe results one of the best topic-annotators in scientific community and its main strengths are that it can annotate also very short texts (namely composed by few tens of terms) and it is very fast. The response of TagMe includes all annotations found in the input text. An attribute to each annotation is associated , called  (rho), which estimates the "goodness" of the annotation with respect to the topics of the input text. You can deploy  to discard annotations that are below a given threshold. The threshold should be chosen in the interval $[0, 1]$ and on our datasets it resulted that the best value is 0.1. Anyway there are few works in literature related to semantic web and tagging, especially inherent to RDF properties and tagging. There is special interest in tagging system as there is the purpose of producing improvements and new point of views for Question Answering. In the five chapter will be explained the chosen ideas to implement the TagProp system.

# Chapter 3

# Qpedia: An user-Friendly Interface for RDF data

There has been much recent interest in user-friendly interfaces that support queries and searching the Semantic Web, without requiring knowledge of SPARQL and the internal structure used by DBpedia or other knowledge bases.

This chapter examines the problem of querying and searching the Semantic Web from mobile devices, by taking full advantage of their small touch-enabled screens, by exploiting an adaptation of the recently proposed concept of SBE query system.

## 3.1 The engine behind Qpedia

Qpedia allows users to show DBpedia facts and search among them in an intuitive way from smartphones and other mobile devices. Searches can be done by providing keywords, values or ranges for properties (either through a keyboard or by voice), and/or location constraints, optionally based on the user location (through GPS if available).

The way constraints can be provided by the user leverages the achievements of the Search By Example approach in [5], where the constraint is associated to a specific RDF property without requiring the user to know the name of the property (e.g., `dbpedia-owl:birthplace` or `dbpedia:placeofbirth`).

Qpedia can be accessed by a mobile phone's web browser, using as a development framework *jQuery Mobile* [28] which is compatible with all mobile browsers. The application can be used on any smartphone operating system and desktop, with an interface able to adapt to any resolution and method of interaction.

Qpedia's initial view contains a free text search and a search button. The way it works is very simple and intuitive: when the user enters some search terms (also by voice through speech recognition), and press enter or the search button, then the application will try to match those terms against DBpedia entities. In case the provided terms are too short or anyway no result is found, a dialog box will pop up warning the user inviting her to change the keywords. Otherwise, the matching results are shown.

Before starting a search, the user can flick (i.e., swiping with the finger) the page on the left, showing an map area that indicates the user position and allowing to select a location range constraint about the entities looked for. The map view can be unzoomed (and therefore the location range will update) through pinch-to-zoom, i.e., by touching screen's surface with two fingers bringing them closer together, or zoomed if moved them apart, in order to respectively increase or decrease the location range constraint. Location constraint can be easily switched off by a slide button. Figure 3.1 shows the various QPedia interface views under a recent Android web browser.

After launching the search, a new view will appear showing matched entities in DBpedia. By clicking on a result, its corresponding entity will be chosen, and its infobox (as in Wikipedia) will be shown. This is done to introduce entity data to the user in a familiar way. The user can then choose, flicking the view on the left or on the right, between the current infobox view, the advanced search view and the map view.

The advanced search view shows all the properties of the current resource, using an expandable listview instead of the infobox. In this view, by a long press on a property, it's possible to introduce a new constraint on the selected property. By further flicking, the map view is shown. If spatial information is available (such as latitude and longitude), the map will be centered on that point, also allowing to input a location constraint. Interactions among views are shown in Fig. 3.2.

By pressing the search button, it will be started a background SPARQL query generated by QPedia. Fig. 3.3 shows the raw SPARQL query, available to experienced users, for the corresponding query *"all Sardinian cities with population between* $15,000$ *and* $25,000$ *inhabitants"*. In order to write such query the user will just specify the constraints for each property of interest, by and editing dialog, such as changing `Sardinia` for the property "region" or the range `15000<> 25000` for the property "population total"; after pressing the search button in the action bar dialog, Qpedia will list all Sardinian cities with a population total between $15,000$ and $25,000$.

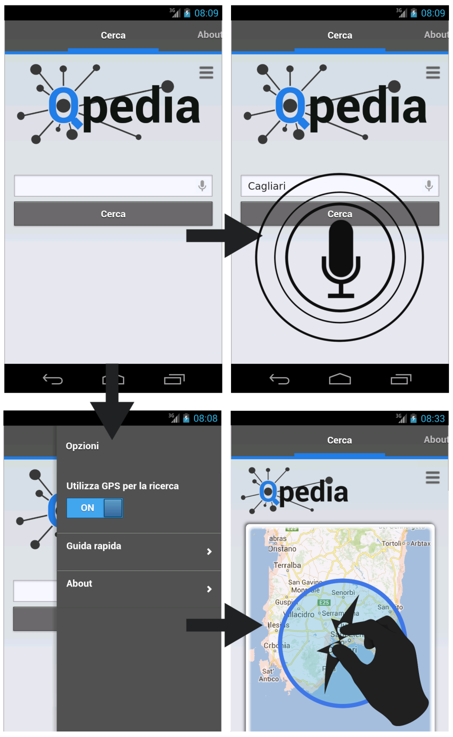


Figure 3.1: User interaction Qpedia

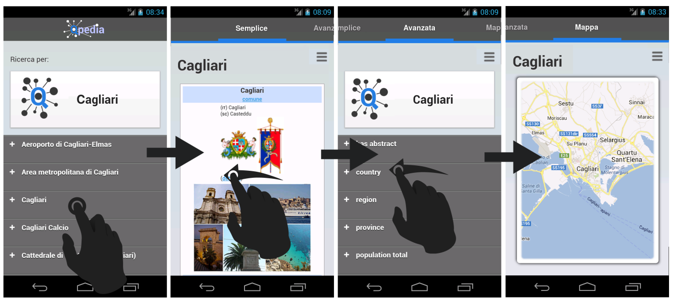
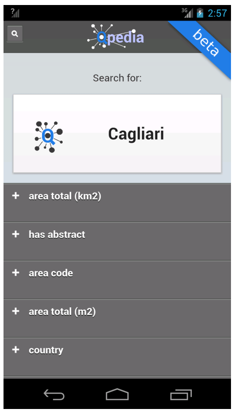
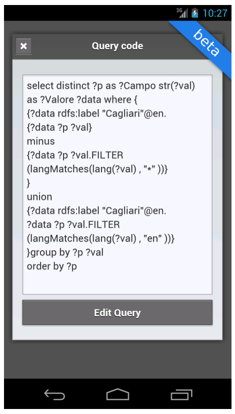


Figure 3.2: Searching and querying in Qpedia



(a) Background SPARQL Query



```
select distinct ?p as ?Campo str(?val)
as ?Valore ?data where {
{?data rdfs:label "Cagliari"@en.
{?data ?p ?val}
minus
{?data ?p ?val.FILTER
(langMatches(lang(?val) , "*" ))}
}
union
{?data rdfs:label "Cagliari"@en.
?data ?p ?val.FILTER
(langMatches(lang(?val) , "en" ))}
}group by ?p ?val
order by ?p
```

(b) Property Change

Figure 3.3: Search By Example in QPedia

## 3.2 Implementation

So far the user interface of Qpedia is described. In order to achieve such user-friendly experience, Qpedia is made of a number of modules that we review in the following. In Appendix C it is possible to follow the instruction to install Qpedia.

### 3.2.1 The UI Module

This module is responsible for showing the user interface described in the previous Section. In order to be portable and available on the majority of the devices, this part has been developed using HTML and JavaScript, therefore accessible through any mobile browser. Most of the interface has been developed with *jQuery Mobile* [28], compatible with almost all browsers in use. Qpedia should therefore be available on any smartphone operating system and desktop, with an interface self adapting to any resolution and method of interaction. The use of jQuery Mobile allows skins to be personalized depending on the device/OS used, enhancing the UX.

The UI module is also in charge of communicating with the Qpedia backend server through AJAX calls (see Fig. 1.3), sending user inputs and obtaining the elements to be shown in the interface. In particular, constraints provided by the user in the query are sent to the server, which in turn will answer with the query results. Results are graphically elaborated by the UI module before showing them to the user.

### 3.2.2 The Query Manager

This module is responsible to generate the SPARQL queries to be sent to the triplestore described next. The Query Manager is in charge of interpreting the conditions entered by the user through the user interface. These conditions can correspond to a text-based keyword search on some RDF properties (e.g., `dbpedia-owl:abstract`), a constraint on a location property (e.g., `geo:long` and `geo:lat`) of RDF entities expressed using the map view of Qpedia UI, or a constraint on other RDF properties (currently any available in DBpedia).

Other than a query translation function, the query manager also provides alias for the items shown by the interface, i.e., instead of showing raw RDF attributes, the QM sends more explicative strings such as the ones obtained querying the `rdfs:label` property of the entity at hand.

Most of the features and solutions regarding this module are part of the Search by Example approach used in SWiPE [5]. One big difference in Qpedia is that in SWiPE the user inputs the constraints within the HTML of the original infobox, which is a non trivial problem to solve given the fact that there is no markup to recognize the property position within the HTML, and RDF values do not necessarily match strings in the infobox. In our case, Qpedia shows a structured list of properties that allows the user to input a constraint, therefore it is straight-forward to know which property the user was meaning to edit. On the other side, the properties should be shown where the user will expect to be, that is, in the same order as shown in the original infobox (which is one flick away from the advanced search view).

Finding the order in which properties appears within the infobox HTML is a new non-trivial problem to solve. Fortunately is possible to leverage the tools developed in [5] to find property positions and therefore, by inspecting the `top` CSS attribute, it is possible easily recover how properties are vertically sorted in the infobox HTML.

### 3.2.3 Triplestore / Execution Manager

This module is responsible for executing the SPARQL query and returning the results to the users. In order to ensure fast response and execution times some experiments has been made with alternative query execution engines. In particular, the Virtuoso system used in DBpedia proved too slow on some keyword-based queries where multiple attributes where involved. Further, the online service freely provided by DBpedia (either standard and "live" endpoints) showed low service availability when accessed programmatically. To solve theses performance problems it has been developed a version of Qpedia backend server that uses full-text Lucene indexes on a local server, based on a modified version of the *Apache Jena* triplestore. Qpedia also features a mechanism that dynamically tries different endpoints whenever a service availability issue might occur.

### 3.2.4 Native Android Client

Generally a web application has limits and missing features, avoidable only through a native client. The restrictions primarily affect some performances, for example, using for a long time a web application the browser cache can get saturated and the UX will decrease. Other opportunities coming from a native app are the social aspects, integration with other mobile apps, sharing customized searches or new features like to save favorited searches. Therefore it also developed an Android application based on a simple *webview*, optimizing performance and implementing other features such as a bookmark of favorited searches. The main screen is a *Fragment Activity*[1] with a *PagerAdapter* that contains two sections, module search and favorites list, where it's possible to save each favorited search on smartphone physical memory. To overcome possible cache problems, every search runs on a different *webview* using a new javascript interface. This solution, after a number of tests has shown a significant improvement in device performance.

[1] `developer.android.com/reference/android/support/v4/app/FragmentActivity.html`

# Chapter 4

# RankProperties: A possible solution for Ranking RDF properties

This chapter is the heart of my thesis and illustrates the main scientific contribution made. The proposed solution is novel and according to our empirical evaluation is effective, thanks to the leveraging of well-known results in the MLR field, which will be discussed below in the following section. Instead the features proposed in the second Section have been used as input data for learning to rank algorithms [43].

## 4.1 MLR Algorithms

Machine-learned ranking (MLR) is the application of machine learning, typically supervised, semi-supervised or reinforcement learning, in the construction of ranking models for information retrieval systems. Training data consists of lists of items (in our case property) with some partial order specified between items in each list. This order is typically induced by giving a numerical or ordinal score or a binary judgment (e.g. "relevant" or "not relevant") for each item. The ranking model's purpose is to rank, i.e. produce a permutation of items in new, unseen lists in a way which is "similar" to rankings in the training data in some sense.Various systems for learning to rank have been proposed in the literature such as *LibSVM*[1], *Sofia-ml*[2] or *Ranklib*[3]. To make ranking we preferred to use RankLib, a library of learning to rank algorithms, which also supports a simplest convenient interface to employ MLR algorithms. The

[1]`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
[2]`https://code.google.com/p/sofia-ml/`
[3]`http://lemurproject.org/ranklib.php`

file format used by RankLib of the training, test and validate files is the follow[4]. In RankLib currently eight popular algorithms have been implemented:

1. **RankNet** [14] is based on a simple probabilistic cost function implementation using a neural network to model the underlying ranking function;
2. **RankBoost** [30] is another ranking algorithm that is trained on pairs, and it attempts to solve the preference learning dbpem directly, rather than solving an ordinal regression problem;
3. **AdaRank** [64] can be viewed as a machine learning method for direct optimization of performance measures, based on a different approach;
4. **Coordinate Ascent** [47] is a commonly used optimization technique for unconstrained optimization problems. The algorithm iteratively optimizes a multivariate objective function by solving a series of one dimensional searches;
5. **LambdaMART** [61] is a learning to rank algorithm based on Multiple Additive Regression Tree (MART);
6. **MART (gradient boosted regression tree)** [31] is an implementation of the gradient tree boosting methods for predictive data mining (regression and classification);
7. **ListNet** [16] is a learning method for optimizing the listwise loss function based on top k probability, with Neural Network as model and Gradient Descent as optimization algorithm;
8. **Random Forests** [13] are an ensemble learning method for classification (and regression) that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes output by individual trees.

## 4.2 Proposed Features

In this Section we describe the features we associate to each RDF property. They provide a numerical or categorical value that can be exploited by the MLR algorithm in order to predict the correct ranking.

[4]`www.cs.cornell.edu/people/tj/svm_light/svm_rank.html`

In our work, we ignore schema/ontology properties specific to each RDF dataset by design, therefore allowing our approach to work seamlessly on any RDF dataset. The only information (provided by all popular RDF KBs) that we consider of great importance is the property label. These KBs use the Resource Description Framework (RDF) as a flexible data model for representing the information and our opinion is that not only Entities on the Web of Data need to have labels in order to be exposable to humans in a meaningful way [24] but also the properties This allows us to compute any feature on property belonging to any knowledge base. We designed 9 features, on the last part of Uri (or the label) about property, with computational performances and generality in mind, some specifically addressed to the most relevant DBpedia RDF dataset. Here we describe all of them, and later in Section 4.5 we empirically study them and operate feature selection based on time performance and their contribution to the trained model.

A. *Frequency*: Frequency is generally the most used feature for ranking. We notice that the frequency of a property, say `dbpedia-owl:populationTotal` in DBpedia, can be obtained by computing statistics offline, or even online by running the following SPARQL query against a DBpedia endpoint:

   ```
   SELECT COUNT(*)
   WHERE { _:a dbpedia-owl:populationTotal _:b }
   ```

   and obtaining the number of triples in which such property is used. Another variation would be:

   ```
   SELECT COUNT(DISTINCT ?entity)
   WHERE { ?entity dbpedia-owl:populationTotal _:b }
   ```

   that is, the number of entities that have at least one triple using such property. The last frequency number can be more appropriate in some circumstances, for instance when the same property may occur several times on certain entities. It is the case of the property `abstract`, that may appear several times on DBpedia entities translated in different languages. The same also applies to the common `label` property.

   High frequency usually implies that the property is very common, and therefore related to the importance of the property. Frequencies are also used by the Typicality approach in order to estimate conditional probabilities of classes and properties.

B. *NumberOfWords*: Some information about properties can be obtained very fast, even without querying the SPARQL endpoint or dataset. One of such features, that we defined to help the ranking, is the count of the words contained in the property's name. For instance, `populationTotal` contains two words, "population" and "total". This information can be obtained by the URI itself or by the `label` property of each property. In the latter case (also applicable to meaningless URIs such as those used by DBpediaLite and Wikidata), a simple query must be run against DBpedia, possibly filtering non-english labels. The rationale is that only one or two words are necessary for important characteristics of an entity, while multiple terms may regard an over-specific attribute.

C. *ContainsNumber*: Similar to the previous feature, also the presence of numbers in the URI can give insights about the quality of the property. For instance, properties `source1Region`, `plane1Origin` and `plane2Origin` are difficult to be interpreted by users that are not very knowledgeable of the underlying schema. Therefore, the presence of numbers in the URI may be associated to a penalty in the ranking score, depending on the training model.

D. *IsEnglish*: Following the same argument used in the ContainsNumber feature, property names (or labels) can be more or less meaningful. In order to measure it, we check against an English dictionary (by using stemming capabilities of the NLTK library) and verify that the property name in the URI (or its label) has a meaning in English.

E. *OntOrRaw*: This DBpedia specific binary feature determines whether the feature is in the DBpedia ontology (prefix `dbpedia-owl`) or not. Properties belonging to the DBpedia ontology are defined by experts and through expert-defined mappings from the so-called "raw properties" extracted from Wikipedia infobox templates.

F. *AnswIsLink*: Another discriminant is given by the range of the property. Specifically, we verify the values for the property and check whether they are literals or links. Although it may depend on each entity, sometimes this feature can be decisive to sort properties. In some applications, only literal are useful, while in others, such as for join queries, links are of fundamental importance. Therefore, knowing the type of property values may be decisive in some contexts.

G. *SuggestChars*: This feature that we are describing is the most novel, and somewhat tricky. As in other approaches in literature (e.g., [48]), we also decided to use external sources, but without relying on the non-deterministic behaviour of most Web Search engines that personalize rankings based on previous keyword searches. Instead, we developed a novel measure that is based on the DuckDuckGo[5] autocomplete functionality: as the user types in the search box, DuckDuckGo suggests longer words or sentences whose prefixes match the string typed by the user. The measure that we compute by using DuckDuckGo autocomplete is obtained as in the following. To compute its value for the property of a given entity, we simulate the typing of an entity name (using the label), followed by a space, and then we see whether DuckDuckGo suggests the name of the property for which we want to compute the DuckDuckGoSuggestChars feature value. If so, we know that zero extra characters are needed in order to get the property in the list of suggestions. Otherwise, we simulate the typing of another character, the first letter of the property. For instance, if we want to compute the value of this feature for the property `areaCode` given the entity *New York*, we first check whether the string "area code" is suggested after "New York" is provided. Since DuckDuckGo does not suggest it, we provide "New York a", again without having the desired suggestion. Finally, by providing "New York ar" DuckDuckGo suggests "New York area codes", and therefore we conclude that 2 extra characters must be typed (namely "a" followed by "r") in order to get the name of the property suggested by the service. We also use lowercase and stemming to avoid singular/plural differences and other linguistic-related issues. This value measures how popular is the name of the property w.r.t. the provided entity, according to DuckDuckGo suggestions (which in turn are based on real user preferences). Notice that the same feature can be computed without any predetermined entity. For the example at hand, in this scenario (without providing "New York") at least 3 letters (not 2 as in the previous example) must be typed before getting a suggestion of "area codes", in our experimental setting.

   Thanks to this novel feature computed through DuckDuckGo, the value (number of chars) does not decrease if the same query is posed multiple times (since results are not personalized per user). The feature allows to take into account conditional relevance (such as relevance of a property given an entity) without

[5]`https://duckduckgo.com/`

expensive computations. The value may also change over time depending on the relevance changes in the DuckDuckGo user base, or cached for additional speed-up in some applications.

H. *SWiPE*: The last feature that we are describing is inspired by the *SWiPE System* [5, 49]. As mentioned earlier, in order to simplify the process of generating SPARQL queries over DBpedia, SWiPE recognizes fields within the infoboxes, making them searchable and introducing therefore a user-friendly query interface.

   For this work, we implemented an API that provides a list of the most important properties used in SWiPE, ranked according to their appearance in the infoboxes. We set every property listed in the infobox as relevant for the entity at hand, therefore assigning a value of 1, while properties not showing in the list are assigned a value of 0.

## 4.3 Training Set

In order to create our training set for RankLib we devised different approaches. Operatively, this will impact the values in the first column of a SVM file, specifying the correct classes to RankLib. Values depend on the relevance of a property w.r.t. an instance in the training set, where the score is induced by the order in the ranked training set.

The following are the approaches that we developed, some of which are completely automatic to compute, not requiring any user intervention. Hybrid methods would also be possible, merging training sets obtained with different methods.

I. Expert-based Training

   This training model is based on a judgement from a semantic web expert that evaluates all properties contained in a number of chosen entities. Features provide a numerical value, a dimension exploitable to split the property space into different categories, such as: very important, important, possibly important, relevant but not important, unuseful. A larger or smaller number of categories is also possible. These classes, provided by an expert on a sample of properties, are used to teach MLR algorithms how to rank each property based on its features After manual scoring, the properties are sorted in a descending order. Their sorting position is taken and placed like class to our training set. Each of the training set row represents one property and contains the class and the feature's results.

II. Questionnaire-based Training

This training system is similar to the previous. We administered a questionnaire on 5 CS graduate students unfamiliar with DBpedia and semantic web, and they were asked to assign a score to a number of properties, depending on their relevance w.r.t. a given concept (a DBpedia entity). We computed the average of such evaluations and sorted them in a descending order. Their sorting position is taken and placed like class to our training set. More details on the questionnaire administration mode can be found on a report available on the project website at `http://atzori.webofcode.org/projects/rankProperties`.

III. Frequency-based Training

This training mode exploits the feature (A), without user intervention. We computed this feature for each property and sorted them in a descending order. Their sorting position is taken and placed as its score to our training set.

IV. Suggest Training

This training mode exploits the feature (G), without user intervention. We computed this feature for each property and created a list of properties sorted by this value, in a descending order. The score is computed based on the position in the list, as in the Frequency-based Training.

V. Typicality-based Training

Typicality approach is very interesting and for this we use it to create a training set. We computed this approach for each property and with the score sorted them in a descending order. Their sorting position is taken and placed like class to our training set. Each of the training lines represents one properties and contains, the class just obtained and the feature's results which complete its content.

VI. SWiPE-based Training

Finally, we propose to create a training set based on our novel feature (H). This training mode was implemented in two versions: ordinal score and numerical score. For the first one, we sorted the properties by computing this feature for each property, then sorting them in a descending order and using the position as the ranking score value. In the second one, we used the a binary approach, marking all properties recognized by SWiPE as relevant, and all the other as not relevant.

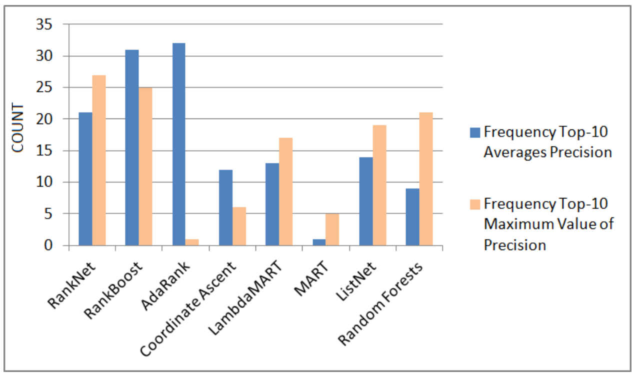


Figure 4.1: Best performing MLR algorithms w.r.t. average and best Precision.

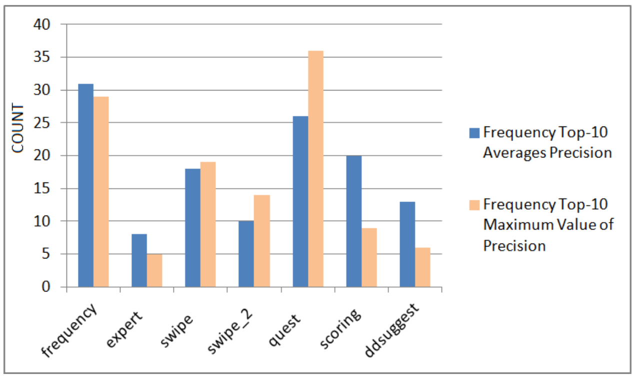
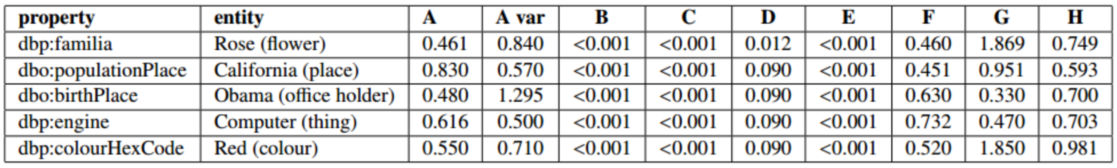


Figure 4.2: Best performing Training mode w.r.t. average and best Precision obtained by the trained models.

## 4.4 Models

In this Section we describe the application of our approach, discussing the model we computed varying the three dimensions (features, algorithms and training sets). In order to manually classify the learning set and then carefully check the outcomes, we experimented on a small set of entities picked up at random from those belonging to these different categories: Flowers, Fruits, Singers, Cities and Colors. In order to prepare the input dataset we have been assigning to each input property a class obtained from our training modes. We have split our classified properties and features into three files, respectively training set, test set and validate set.



| property | entity | A | A var | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|---|---|
| dbp:familia | Rose (flower) | 0.461 | 0.840 | <0.001 | <0.001 | 0.012 | <0.001 | 0.460 | 1.869 | 0.749 |
| dbo:populationPlace | California (place) | 0.830 | 0.570 | <0.001 | <0.001 | 0.090 | <0.001 | 0.451 | 0.951 | 0.593 |
| dbo:birthPlace | Obama (office holder) | 0.480 | 1.295 | <0.001 | <0.001 | 0.090 | <0.001 | 0.630 | 0.330 | 0.700 |
| dbp:engine | Computer (thing) | 0.616 | 0.500 | <0.001 | <0.001 | 0.090 | <0.001 | 0.732 | 0.470 | 0.703 |
| dbp:colourHexCode | Red (colour) | 0.550 | 0.710 | <0.001 | <0.001 | 0.090 | <0.001 | 0.520 | 1.850 | 0.981 |

Figure 4.3: Time of Execution (seconds) for each A-H Ranking Feature.

Therefore, we eventually ended up with a search space of $8 \cdot 9 \cdot 6 = 432$ models (8 algorithms, $2^9$ feature combinations, and 6 training sets), that we explored in large part. In order to reduce the number of feature combinations, we first analyze the set of features and operate a feature selection based on average time and precision performance. Then, we describe how we obtained good generated models, selecting them by using *Spearman's rank correlation*[54].

## 4.5 Feature Selection

We ran a set of test to measure feature performance in terms of time and f-measure. Results can be useful to operate feature selection based on the required time and precision performance.

### 4.5.1 Time Performance

We developed a script to compute the average time required to evaluate a property feature, as summarized in Table 4.3. Experiments show that features B-E are extremely fast to compute, while A, Avar, F-H may introduce some delay due to the network connection required to query the DBpedia endpoint and the DuckDuckGo Suggest service and SWiPE API, respectively. In this test we disabled any cache, used a clean system and we did not use the prefetching optimizations for any feature.

### 4.5.2 Precision Performance

Our set of experiments based on DBpedia, WikiData, FreeBase and MusicBrainz shows that the best performing features are: NumberOfWords (B), isEnglish (D) and DuckDuckGoSuggest (G). This result is based on the evidence that various performance indices such as Frequency, Recall, f-measure and Spearman's rho seem not to decrease by removing the other features. In fact sometimes performance improve by removing some features, showing that they may inappropriately overfit the data. Table 4.1 shows an optimal configuration using only the three mentioned features

| System | Configuration | F-Measure |
|---|---|---|
| RankProperties | quest_alg1_B_D_G | 67% |
| RankProperties | quest_alg1_All_Features | 58% |
| Typicality 3 | $P(c\|a)$ | 55% |
| Typicality 1 | $P(i\|a)$ | 55% |
| IBminer | default | 53% |
| Typicality 2 | $P(a\|i)$ | 41% |

Table 4.1: Best performing configurations according to f-measure, compared against Typicality and IBminer (assessment of 50 entities, totalling 1346 properties).

| Position | Property |
|---|---|
| 1 | name |
| 2 | area |
| 3 | country |
| 4 | disambiguation |
| 5 | id |
| 6 | type |
| 7 | aliases |
| 8 | ipis |
| 9 | sort-name |
| 10 | label-code |
| 11 | life-span |

Table 4.2: RankProperties on MusicBrainz's properties of "The Guardian" entity

(B, D, and G), getting better results in terms of f-measure than the Typicality and IBminer approaches. Typicality presents three configurations where $P(c|a)$ denotes how typical concept (ontology) $c$ is, given attribute $a$, $P(a|i)$ denotes how typical attribute $a$ is, given instance (entity) $i$ and finally $P(i|a)$ denotes how typical instance $i$ is, given attribute $a$.

### 4.5.3 Examples of Generated Models

Here we qualitatively examine an example of generated models. As we also quantitatively see in the Experiment Section, we tested, by varying the three dimensions features, algorithms and training systems, all possible combinations on many entities. This was done in order to find the best feature/algorithm/training mode combination, also comparing our models against the other existing approaches. Thanks to this, we observed a number of optimal combinations that requires only few features, as done mentioned above for feature selection. Histogram in Fig. 4.1 shows how many times each algorithm is in the Top-10 best performing list in terms of average

| Position | Property | Label |
|---|---|---|
| 1 | P357 | title |
| 2 | P112 | founder |
| 3 | P17 | country |
| 4 | P98 | editor |
| 5 | P407 | language |
| ... | ... | ... |
| 20 | P214 | VIAF identifier |
| 21 | P227 | GND identifier |
| 22 | P243 | OCLC control number |
| 23 | P966 | MusicBrainz label ID |
| 24 | P1438 | Jewish Encyclopedia ID (Russian) |

Table 4.3: RankProperties on Wikidata's properties of "The Guardian (Q11148)" entity

| Position | Property |
|---|---|
| 1 | issues |
| 2 | publisher |
| 3 | language |
| 4 | country |
| 5 | contents |
| 6 | quotations |
| 7 | subjects |
| 8 | properties |
| 9 | alias |
| 10 | headquarters |
| ... | ... |
| 21 | type |
| 22 | price |
| 23 | city |
| 24 | notable types |
| 25 | final issue date |
| 26 | webpage |
| 27 | weblink |
| 28 | official website |
| 29 | ISSN |
| 30 | also known as |
| 31 | frequency or issues per year |

Table 4.4: RankProperties on Freebase's properties of "The Guardian" entity

and maximum value of Precision, Recall and Spearman's Rho (label COUNT on y-axis). Algorithms [30, 14] are the best in both terms. MART and Coordinate Ascent produced the worst results, where the output was homogeneous and uninteresting ranking. Instead, RankNet and RankBoost have been able to learn the user-assigned classes, ranking other unclassified properties accordingly. Histogram in Fig. 4.2 shows how many times each Model Training is in Top-10's in terms of Average and Maximum value of Precision (label COUNT on y-axis). That is, while Fig. 4.1 shows a comparison of MLR algorithms in similar conditions, Fig. 4.2 compares instead the performance of different training modes. In particular, we realize that models obtained by using Frequency and Questionnaire Training modes produce the best results in terms of precision. Also SWiPE-based training set has good average and maximum precision performance according to our experimental setting, with the advantage of being completely automatic and fast to compute with on fast networks or by using bulk requests (i.e., a single HTTP connection for many requests).

We can also assess the level of the various ranking approaches by qualitatively analysing their outputs, as shown Table 4.5. The table presents the rankings for the UK newspaper entity "The Guardian" as computed by each system. In bold the correct property classification according to values in the first column (provided by the expert and not part of the training set). We see that our method sorting is the best to set high-quality attributes on the first ten positions. In this example, our method is able to correctly identify six important properties in the first ten, more than typicality and even more than IBminer, which is specifically trained by humans.

We can also observe the effect in the tail of the ranked list of properties, where also many insignificant properties are recognized as not important. To obtain this ranking we used the following configuration: the best performing features NumberOfWords (B), isEnglishc(D) and DuckDuckGoSuggest (G), Suggest Mode as training mode and RankBoost as MLR algorithm. This is just a randomly picked up example, and there are of course some different configurations which can obtain even better performance using other Model Training and MLR algorithms.

An important result of our work is the ability to use trained model to sort a given set of properties never seen before. That is, a set for which no ontology or frequency data is given. We implemented a python tool[6] which checks the URL of an endpoint and automatically compute RDF property ranking, applying pre-computed models on-the-fly (see installation instruction in Appendix A). To test this feature, other than DBpedia, we used MusicBrainz, Wikidata and Freebase, using

[6]An online API is available at `http://atzori.webofcode.org/projects/rankProperties/`

again entity *The Guardian*. It is important to note that the set of feature may be completely different on another knowledge base, and therefore user-defined scoring such as IBminer cannot be used. Further, frequencies are not available or very long to compute (order of minutes), and therefore typicality is not feasible in this context, while RankProperties with fast-to-compute selected features is effective. Table 4.2 shows the output (sorted list of properties) for the MusicBrainz KB. Table 4.3 show the ranking of properties found on Wikidata. To better understand this result, we associated each Wikidata property with its label according to the site[7]. Therefore the second column contains original properties (which are meaningless for humans) while the third column contains the translation useful to sort them and to understand the result. Finally, Table 4.4 contains the ranking results for properties found on Freebase. Interestingly, while models where not specifically trained on those data, rankings appear to be meaningful. Less important properties, such as those containing IDs, codes or other apparently less useful data, are correctly put low in the sorted lists.

[7]`http://www.wikidata.org/wiki/Wikidata:List_of_properties/all`

| ***Expertise/Questionn.*** | ***RankProperties*** | ***Typicality*** | ***IBminer*** |
|---|---|---|---|
| dbp:editor | dbp:circulation | dbp:sisterNewspapers | **dbp:editor** |
| dbp:format | **dbp:editor** | **dbp:opeditor** | **dbp:website** |
| dbp:foundation | **dbp:foundation** | dbo:circulation | **dbp:publisher** |
| dbp:owners | dbp:language | **dbp:political** | **dbp:owners** |
| dbp:publisher | **dbp:publisher** | dbp:issn | dbo:sisterNewspaper |
| dbp:website | **dbp:website** | dbp:circulation | dbp:name |
| dbp:political | **dbo:editor** | dbo:sisterNewspaper | dbp:circulation |
| dbo:editor | **dbo:owner** | **dbo:editor** | **dbp:foundation** |
| dbo:owner | dbo:circulation | **dbp:editor** | dbp:language |
| dbp:caption | dbp:abstract | **dbp:owners** | dbp:format |
| ... | ... | ... | ... |
| dbp:language | dbp:caption | dbp:cost | - |
| dbp:name | dbp:format | dbp:caption | - |
| dbo:wikiPageExt.. | dbp:issn | dbo:abstract | - |
| dbp:oclc | dbp:oclc | dbp:name | - |
| dbo:wikiPageWiki.. | dbp:owners | dbo:wikiPageWiki.. | - |
| dbo:wikiPageLength | dbp:opeditor | dbo:wikiPageLength | - |
| dbo:wikiPageOutD.. | dbo:format | dbo:wikiPageOutD.. | - |
| dbo:wikiPageRev.. | dbo:wikiPageRev.. | dbo:wikiPageRev.. | - |
| dbo:wikiPageID | dbo:wikiPageID | dbo:wikiPageID | - |

Table 4.5: Rankings for newspaper entity "The Guardian" as computed by each system (32 properties to the first three, 18 properties for the last column about IBminer [All orders]). In bold the correct property classification according to values in the first column (provided by the expert and not part of the training set). dbpprop is renamed to dbp and dbpedia-owl is renamed to dbo

# Chapter 5

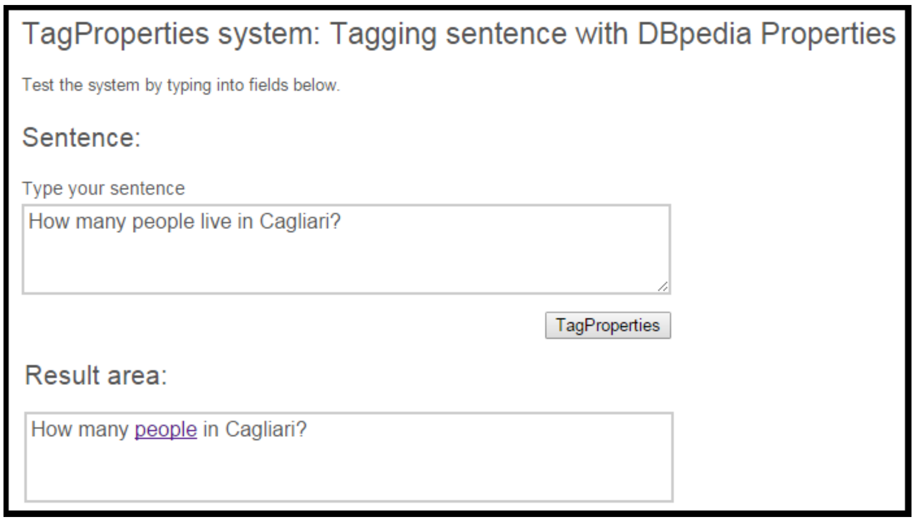
# TagProp: An idea for tagging RDF Property

This chapter contains an experimental part of my thesis, where I have been concepts and techniques also described about the previous chapters.

TagProp, an idea to automatic assignment of the appropriate RDF properties on some words in a free-text, could be right a specialised instance of the general problem of QA, even if it is not typically addressed in literature through this way. This kind of tagging can be carried out effectively by combining several simple, independent, methods and this Chapter includes the explanation and design of such tagger. A prototype of this system has been implemented, correctly tagging some of the sentences, listed in Section 6.3, thus providing evidence that this hypothesis presents promising results. However, there's still plenty to do in terms of execution time, and this point needs to be taken fully into account. At present, there were no publications for this part of work.

## 5.1 The TagProp algorithm

The TagProp tool allows users to tag DBpedia RDF properties into sentence in natural language in a fast way. In the image 5.1 is represented a Graphical User Interface. The technologies incorporated in the design and construction are HTML, Javascript and Python.


TagProperties system: Tagging sentence with DBpedia Properties

Test the system by typing into fields below.

Sentence:

Type your sentence

How many people live in Cagliari?

Result area:

How many people in Cagliari?

Figure 5.1: TagProp Grafical User Inteface

By analysing the current structure and the usage patterns of semantic RDF property tagging in free-text, there aren't many systems and aspects that still need to be improved. Problems related to synonyms, polysemy, heterogeneous lexical forms, typos, different levels of precision and different kinds of tag-to-exactly RDF property association cause inconsistencies and reduce the efficiency and the effectiveness towards Question Answering. They are mainly caused by the multiple significances of natural language sentence in the tagging process. The idea is to provide a new way to augment free-text through the semantic tagging. It allows user to identify semantic assertions and then a possible answer, if it is a question: each part of sentence expresses one or more properties of a resource (entity) associating it with concepts and properties.

The tool TagProp, a semantic tagging system, exploiting RankProperties to select the best adequate properties and to select the perfect entities it is a perfect solution TagMe or the NLTK tools. We explore the adequacy of the support offered by the entities and properties of Wikipedia and WordNet in order to access to and reference concepts.

Starting from the image of tagging systems and introducing the semantic tagging, a new way to augment free-text, it has been developed TagProp, a semantic tagging system. The following lines describe its architecture and its main organizational features, giving also some practical example of functioning. In particular, it is possible to identify the following main causes of weakness related to different ways of using words in natural language sentences:

- Polysemy : the same word can refer to different concepts (the word ‘live’ can refer to verb or adjective);
- Synonyms : the same concept can be pointed out using different words (‘people’, ‘population’, ‘citizen’ are three different words that refer to the same concept: a group of humans);
- Heterogeneous lexical form : the same concept can be referred to by different noun forms, for instance plural nouns (‘year’/‘years’), different verb conjugation (‘do’/ ‘did’), name-adjective couples (‘energy’/‘energetic’), multiple words (‘pc’/‘personal computer’) and so on;
- Typos : typing errors that occurs when we write a word (‘popluation’ in place of ‘population’) or different possible spelling of the same word (‘color’/‘colour’);
- Term Precision : the specificity of the word chosen to tag a resource (‘party’ is more specific than ‘group’);

Many of the problems described can be related to natural language sentences in the process of assigning RDF properties. Every natural language could provide multiple concepts. In order to identify a specific concept, we must process the sentence using NLTK. As a consequence we need to exploit some resource that should support the following tasks:

- given a natural language sentence it should identify all its possible entities;
- it should allow to find all properties, starting from entities and looking possible matching between the part of sentence and the same properties.

Considering these fundamental requirements, we have identified three different and may be complementary kinds of resource currently available over the Web:

- WordNet: a lexical database which is based on the concept of set of synonym words, called synset, which defines a particular concept; it is sufficiently structured and includes a lot of lexical and semantic relations between words and synsets. Wordnet is updated by a group of lexicon experts and presents quite a complex net of internal relations, in fact it has been developed in order to support text mining and information extraction. WordNet has a broad coverage of all common parts of speech (nouns, verbs, adverbs and adjectives). At present, WordNet is inside NLTK tool for python.

- TagMe: a 'topic annotator' as described above (API). Alternatively, NLTK tools or a guided query in natural language using for example capital letter for topics, can be used.
- UMBC Semantic Similarity Service: a tool which compute semantic similarity between words/phrases (API). This tool is used to compare each terms about natural language sentence with RDF properties.

Figure 5.2 illustrates an overview of TagProp framework and the process of identify possible entities and relative properties, checking synonyms and ranking them.

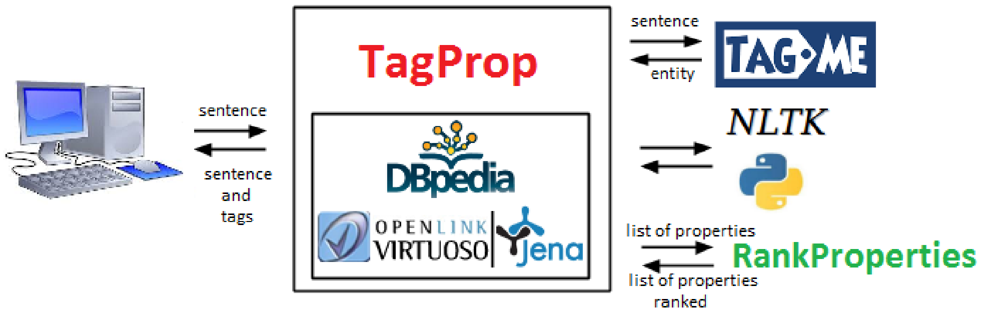


Figure 5.2: TagProp System

## 5.1.1 Choice of correct RDF property and synonyms

As mentioned above, another important tool is Ranking RDF properties can improve disambiguation, as more than a property can match, and a ranking is therefore mandatory to choose; it can also reduce time delays experienced by the users, as the most relevant are considered first, immediately, an then the rest.

The technical options for improving the searching of synonyms chosen is to create a file which contains all of synonyms, hypernyms and hyponyms about RDF properties of a Knowledge Base (i.e. DBpedia) using caching, the process of storing data in a cache.

This file is a tab-separated values file, a simple text format for storing data in a tabular structure, where in each row there is a label of RDF property and a list of synonyms. Each record in the table is one line of the text file. Each field value of a record is separated from the next by a tab stop character. Also this file includes information gathered from multiple sources like these sites [1] [2] and WordNet.

[1]`http://www.synonyms.net/`
[2]`http://www.thesaurus.com/`

### 5.1.2 A practical example

The goal of this section is to introduce the algorithm describing how the background works and it is used in tagging text. The algorithm will be presented in a number of steps in which we will elaborate one simple example:

How many people live in New York?

The first step is to analyze the 'type' of sentence. Information about type shall be provided for each RDF property. There are various types of RDF properties, which have been transcribed below:

- int - an integer number, namely is a number that can be written without a fractional component.
- double - a double number is a computer number format that represents a wide, dynamic range of values by using a floating point.
- text - traditionally a sequence of characters, either as a literal constant or as some kind of variable.
- date - a date in american format
- boolean - a boolean value (true or false)

Thanks to 'Five Ws rules plus one H', for simplicity 5Ws&H [3] and these types, it is possible to make rapid links between natural language and RDF properties. To do that, each 5Ws&H can be associated with a type.

This will be:

- what $->$ text
- when $->$ date
- who $->$ person
- why $->$ text
- where $->$ place
- how many/how much $->$ numbers (int,double)

[3] `https://en.wikipedia.org/wiki/Five_Ws`

In the example above, there is 'How many' then the type is 'int'. The second step is to remove the part of unneeded sentence. To do that, NLTK tools are used. In this case the preposition 'in' has been removed.

In the third step, one or more entities are identified. Thanks to TagMe or NLTK, or simply in this case because there is a capital letter before the word 'New Tork'. Once we find them, it is possible to have all of possible RDF properties about them through a SPARQL query. This step is optional because we can match all of RDF properties without particular list of entities.

The fourth step involves in finding a large file synonyms of RDF properties, and the remaining terms as 'people' and 'live'. To do that a list of properties are obtained and it is possible to rank these properties using RankProperties tool to improve the final results.

## 5.2 Possible backend for TagProp

In order to obtain the list of possible properties to be used for tagging free-text in natural language or to make the query SPARQL after tagging, it is necessary to have an excellent backend. During the experiments we have used mainly local endpoint including Virtuoso [4] and Apache Jena [5]. According to a very interesting benchmark [52], the best backend is Virtuoso, intensively used in this thesis, but because it is commercial system, it has been tested Jena, which it is however open-source and totally customizable.

Virtuoso is the most advanced and native RDF triple store on the market, available in both open source and commercial licenses. It provides command line loaders, a connection API, support for SPARQL and web server to perform SPARQL queries and uploading of data over HTTP.

Jena is a Java framework which is useful for building semantic web applications. It is an open source work. The Jena framework includes a SPARQL query engine, which interprets SPARQL queries against RDF data present in a back-end RDF store. SDB is a component of Jena. It provides for scalable storage and query of RDF datasets using conventional SQL databases for use in standalone applications, J2EE and other application frameworks. Jena recently introduced a non-transactional native store called TDB, a part marked in red. For our current evaluation we have Jena TDB backed with SPARQL.

[4] https://virtuoso.openlinksw.com/

[5] https://jena.apache.org/

The illustration below shows an overview of Jena used to do the test:

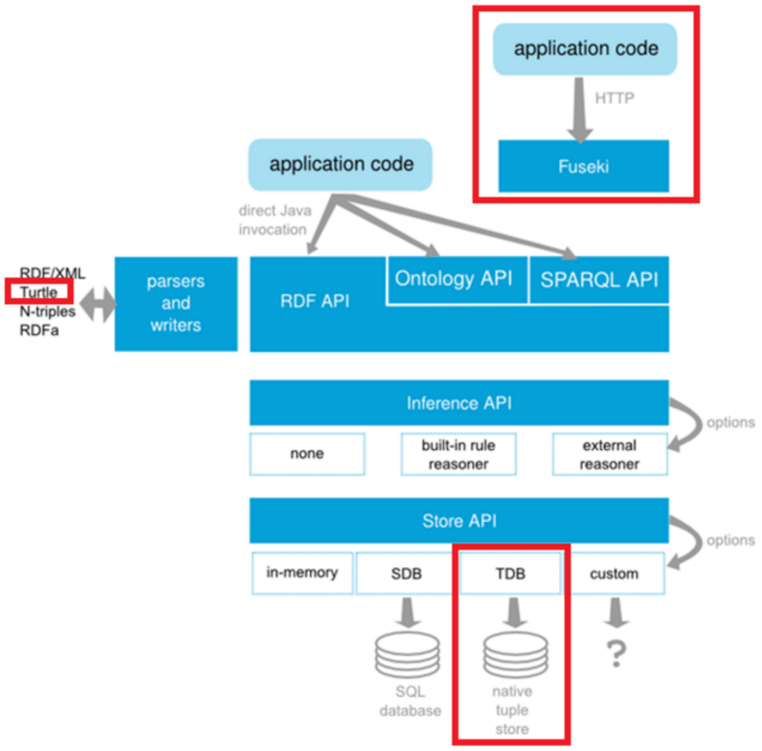


Figure 5.3: Backend Architecture

Another part marked with red marker is Turtle (Terse RDF Triple Language), a format for expressing data in the RDF data model with a syntax similar to SPARQL. And as we saw in the introduction, RDF represents information using triples, each of which consists of a subject, a predicate, and an object. Each of those items is expressed as a Web URI. Turtle provides a way to group three URIs to make a triple, and provides ways to abbreviate such information, for example by factoring out common portions of URIs. For example:

```
<http://example.org/person/Mark_Twain>
   <http://example.org/relation/author>
   <http://example.org/books/Huckleberry_Finn>
```

All the described parts were installed to perform those tests in the following hardware:

- Notebook OLIBOOK S1530 32 bits i3-2310M
- Intel Core 3 Dual core processor 2310M (2.10 GHz)
- 4 GB DDR2 RAM
- 3524236 kB Hard drive
- Apache Jena Fuseki + LARQ(2) [Lucene + Arq]

Something else that is very important is the presence of Apache Lucene into the Backend as one of its fundamental components.

Apache Lucene is a free open-source high-performance information retrieval engine written in the Java Programming language. It offers full-featured text search, based on indexing mechanisms. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform. Apache Lucene is an open source project available for free download. Lucene offers powerful features through a simple API:

- multiple-index searching with merged results
- incremental indexing as fast as batch indexing
- many powerful query types: phrase queries, wildcard queries, proximity queries, range queries and more

Example of indexes organization used in Lucene is drawn below:

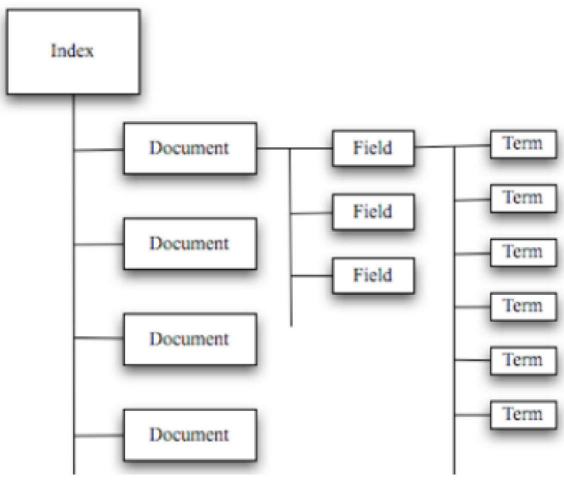


Figure 5.4: Indexes organization in Lucene

So to achieve a “Free Text Indexing for SPARQL” has been tested LARQ [6], a combination of ARQ and Lucene [7]. It gives ARQ the ability to perform free text searches. Lucene indexes are additional information for accessing the RDF graph, not storage for the graph itself.

To evaluate Apache Jena Fuseki + LARQ, have been made a series of SPARQL Query that may be consulted in Appendix D. These queries contain an useful variant to understand the index function and also to obtain a fast customizable backend. The results show that Jena is customizable and potentially efficient but Virtuoso remains faster than Jena, and this will mean more research in this concept.

[6] `https://jena.apache.org/documentation/larq/`
[7] `http://lucene.apache.org/core/`

# Chapter 6

# Experiments

This chapter describes the different experiments exploited in this dissertation, and in particular RankProperties have been covered in depth.

## 6.1 RankProperties

Now I shall give the experiments of the proposed technique about RankProperties. To compare the results obtained with this approach against other systems' ranking were used some terms of comparison presented next in the following sections. The experiments have been conducted to verify the feasibility of the proposed framework and evaluate it, both in terms of time performance and quality of ranking. In particular, it has been focused on a personalizable ranking approach that could be run on the fly, that is, at query or visualization time to solve the problem of ranking RDF properties. This supervised machine-learning framework leverages existing learning-to-rank (MLR) algorithms which are applied to a number of (almost) instantly computable property features that were described previously. Some tools have been created which are available on-line at our project website [1], in order to compute and evaluate many ranking types. The following sections first show the measures used for the evaluation and how it has been set up the testing dataset, and to discuss the outcome of the experiments.

### 6.1.1 Quantitative measures for the Evaluation

The measures chosen to quantitatively evaluate the quality of the rankings are four: Precision, Recall, F-Measure and Spearman's rho correlation. They are suitable to compare a sorted list of properties against a provided sorting, called Pivot

[1] `http://atzori.webofcode.org/projects/rankProperties`

(generally made by humans annotator), and other existing rankings. Precision, Recall and F-Measure are the basic measures used in information retrieval and evaluating search strategies.

*Precision* is the ratio of the number of relevant properties retrieved to the total number of irrelevant and relevant properties retrieved. *Recall* is the ratio of the number of relevant properties retrieved to the total number of relevant properties. These examples are based on Top-N properties [40] for the selected concepts, using the following definition for Precision:

$$Precision = \frac{\sum_{i=1}^{N} rel_i}{N}$$

where $rel(i)$ is the relevance score of the $i$-th attribute. Regarding recall Recall:

$$Recall = \frac{\#\, retrieved\, very\, typical\, prop\, in\, top\, N}{min(\#\, very\, typical\, prop, N)}$$

Finally, a measure that combines Precision and Recall is the harmonic mean of Precision and Recall, which is usually called F1 score or *F-measure*:

$$\text{F-measure} = 2 \times \frac{precision \times recall}{precision + recall}$$

In these examples N = 10 have been used, a value chosen because generally the number of attributes are considered to be very important or typical, for a singular entity, are usually around this value. This is an empirical consideration that comes from a qualitative evaluation of many instances in our experiments.

Regarding the pivot ranking, it has been realized by 5 human annotators. They assigned relevance score organized to four groups as 4 (very typical), 3 (typical), 2 (related), and 1 (unrelated), respectively. After that, using relevance score, the pivot properties were sorted. In the experiments, before computing precision, recall and f-measure, the score was normalized to $[0, 1]$. Finally, I used a technique to evaluate if two rankings were related to each other called *Spearman's Rank Correlation Coefficient* or Spearman's rho. The formula is the following:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

where $d_i^2$ is the difference between ranks, namely the position of a specific property between two different result of ranking. This framework generates four CSV files that contain therefore precision, recall, f-measure and Spearman's rho between the Pivot ranking and each ranking output (including those of existing approaches) that were assessed.

### 6.1.2 Test Dataset

In this section I describe how were generated the test datasets used as testbed for our experimentation. First of all, 18 entities were chosen, with their respective 18 ontologies, 6 for each training files (training, test and validate set of the MLR framework). To create training set Russelia, Parma, Microsoft, Dog, Enrico Berlinguer, Sandra Bullock were chosen and then as ontologies there are Species, Settlement, Public company, Animal, Politician, Agent respectively. The same operation to create validate set with Rose, Cagliari, Facebook, Cat, Aldo Moro, Angelina Jolie together with ontologies as Species, Settlement, Public company, Animal, Politician, Agent. Finally, to create test set I choose these entities Pablo Picasso, Monaco, Conus, Born to Love, The Freddie Mercury Album, Jean de Quen with Artist, Place, Work, Animal, Work and Agent respectively. Choice is not random but is based on a selection of different types (ontologies) which enclose the best known cases. To be more clear, the choice fell on a small set of entities picked up from those belonging to these different categories about Flowers, Fruits, People or Singers, Things, Cities, Animals and Colors. Therefore there are a total of 1015 properties (as shown in the table 6.1 below the allocation of each Entity's properties).

| | **Training set** | **Validate set** | **Test set** |
|---|---|---|---|
| **# Properties** | 314 | 352 | 349 |

Table 6.1: Size of the Datasets used in the experiment

Then, the models were computed for every possible combinations of MLR algorithm, feature set and training mode (ranking assignment). In order to evaluate the results, Hallway testing was applied, this is a common evaluation technique in usability. After pulling out 50 entities collected randomly by APIs wiki[2], five students were involved to evaluate these entities, and in particular their properties, assigning a score from 1 (unrelated) to 4 (very typical). This has served to compare the performance of various ranking systems with the opinion of users, in terms of the measures are

[2]`http://www.mediawiki.org/wiki/API:Random`

indicated above in the section. Finally, have been performed RankProperties tool to create all possible comparisons, producing large tables in order to find the best configuration or system according to the measures were used.

### 6.1.3 Time Performance

For time performance comparison, have been disabled all the cache. In particular, it is assumed frequencies and other dataset statistics are not stored, as they may change over time. Although in DBpedia case it is possible to save the frequencies, such as to speed up the Typicality approach or my A and Avar features, in this comparison it is assumed that were computed rankings on a new, unseen entity or dataset. In particular, this presented approach does not assume the existence of a known ontology. During models creation phase, were used each MLR algorithm and were cycled the other dimensions, that is, the features view in Chapter 5and models training view in the same section Chapter 5to cover all possible cases for use. Time required to compare all various ranking systems with the users ranking in the hallway testing was about 8-10 hours. This value may change depending on both CPU and network speed. For DBpedia frequencies, It's been used a instance of Virtuoso with the last DBpedia 2014 dump, on a 50Gb RAM linux cluster.

The results are displayed on Table 6.2, that gathers the time required by MLR-based RankProperties system to carry out an entity evaluation compared to Typicality. It may be observed from this table that without enabling cache optimizations for both systems, this approach with all features is slightly slower than typicality, since frequencies were used, a time-consuming feature, among the other features. By using only features B, D and G, as per this feature selection outcome, a major speed up was obtained, without loosing precision. This configuration is one order of magnitude faster than Typicality. These are the times required to rank all the entity's properties, on average.

| **System** | **Configuration** | **Time (sec.)** |
|---|---|---|
| RankProperties | All features | 442 |
| RankProperties | Features B, D and G | 46 |
| Typicality | $P(c\|a)$ | 397 |

Table 6.2: Time of execution without any cache. All features and frequencies computed from scratch (assessment of entity "The Guardian", totalling 31 properties).

### 6.1.4 Quality of Ranking

This section presents the experimental results in term of quality of ranking proposed. Once the average values of the results about measures described in Section 6.1.1 has been obtained, are compare them with this approaches through IBminer and Typicality systems. A fragment of results in Table 6.3 is presented where, "quest" and "freq" stand for Questionnaire-based Training and Frequency-based Training respectively, "alg5" stands for LambdaMart and "alg2" stands for RankBoost. It also compared RankProperties system against Random Sort to empathize different results obtained. It can be observed from the same table that there are different configurations about our system which obtain better performance than other existing systems. Therefore, our framework obtains better performance w.r.t. existing work, with improvements in the range of 5% to 10%. It is possible to draw two conclusions from these results. First, it is possible to use only 3 features, selected in Chapter 5with two of these Algorithms and Training Models, without variation of quality. This suggests that the use of these configurations for RankProperties is effective on ranking. Second, the criterion for configuration choosing selection can be based on choices in training phase, looking at the number of times which they obtain high results and better than competitors.

| **System** | **Configuration** | **Prec** | **Rec** | **Rho** |
|---|---|---|---|---|
| RankProperties | quest_alg5_B_D_G | 75% | 64% | 29% |
| RankProperties | freq_alg2_B_D_G | 72% | 58% | 62% |
| IBminer | default | 70% | 50% | 58% |
| Random | - | 58% | 47% | 4% |
| Typicality 3 | $P(c\|a)$ | 65% | 48% | 46% |
| Typicality 2 | $P(a\|i)$ | 65% | 43% | -36% |
| Typicality 1 | $P(i\|a)$ | 41% | 48% | 39% |

Table 6.3: Two optimals configurations against Typicality and IBminer (assessment of 50 entities, totalling 1346 properties).

### 6.1.5 Other important experiments for the Evaluation

Also, regarding this work, other considerable experiments has been made, on different machine learning-to-rank algorithms, comparing them against user data using Spearman's rank correlation coefficient.

## 6.2 Qpedia

Qpedia was evaluated in over 20 English questions of QALD-4 [55] (Task 1). The QALD-4 competition provides an RDF dataset, training and testing questions, and ground truth answers to the questions. It has been loaded the data into a Virtuoso triple store [3]. A subset of that questions have been solved by Qpedia and they are listed as follows:

- How often did Jane Fonda marry?
- What is the official website of Tom Cruise?
- Who created Wikipedia?
- What is the founding year of the brewery that produces Pilsner Urquell?
- Which river does the Brooklyn Bridge cross?
- How tall is Claudia Schiffer?
- In which U.S. state is Mount McKinley located?
- When was the Statue of Liberty built?
- Which books by Kerouac were published by Viking Press?
- Which U.S. state has the highest population density?
- How many films did Hal Roach produce?
- Give me all federal chancellors of Germany.
- Which states of Germany are governed by the Social Democratic Party?
- Which television shows were created by Walt Disney?
- Give me the websites of companies with more than 500000 employees.
- Give me all cities in New Jersey with more than 100000 inhabitants.
- Which actors were born in Germany?
- Give me all people that were born in Vienna and died in Berlin.

[3]`http://db.webofcode.org/sparql`

The remaining two questions haven't been solved because there are some limits on Qpedia system. They are:

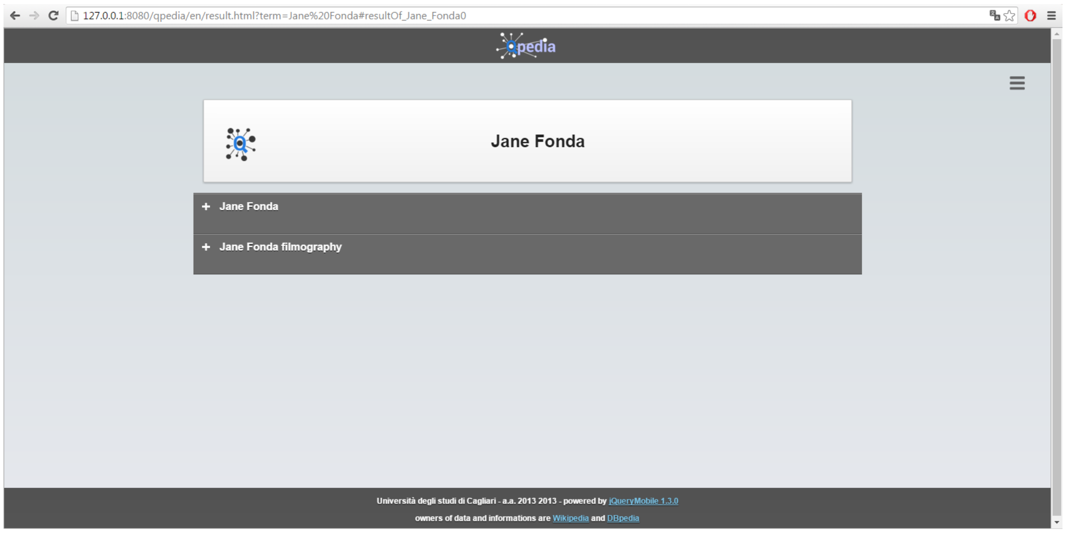
- In which country does the Nile start?
- Which countries have more than two official languages?

The overall results are shown in the following Table 6.4:

| | |
|---|---|
| **Processed Questions** | 90% |
| **Precision** | 0.90 |
| **Recall** | 0.87 |
| **F-measure** | 0.88 |

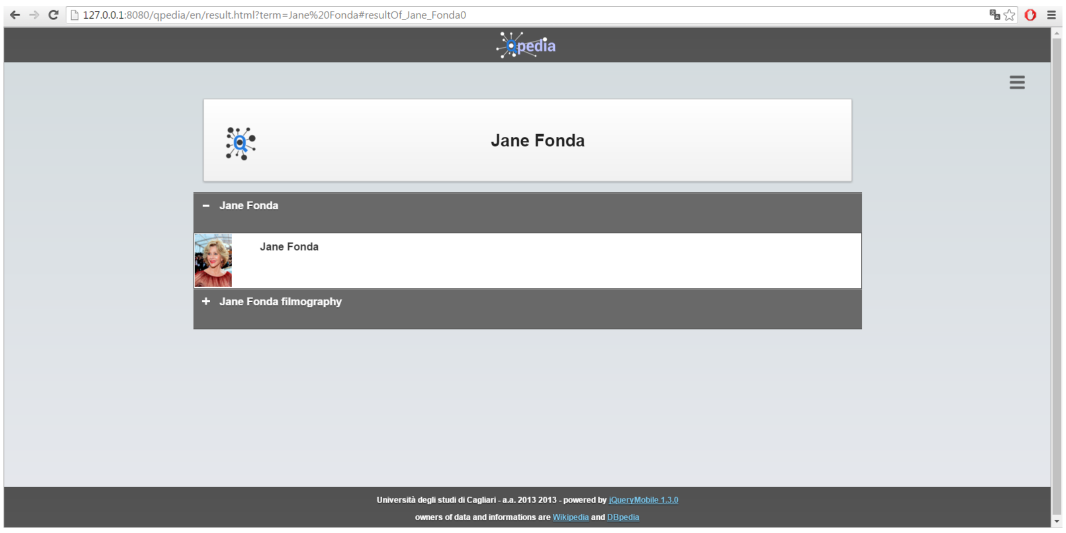
Table 6.4: Qpedia Results

Here is step-by-step method that shows the first question "How often did Jane Fonda marry?":


127.0.0.1:8080/qpedia/en/result.html?term=Jane%20Fonda#resultOf_Jane_Fonda0

Università degli studi di Cagliari - a.a. 2013 2013 - powered by jQueryMobile 1.3.0
owners of data and informations are Wikipedia and DBpedia

Figure 6.1: Qpedia 1 step: Finding the subject (Entity) of question


127.0.0.1:8080/qpedia/en/result.html?term=Jane%20Fonda#resultOf_Jane_Fonda0

Jane Fonda

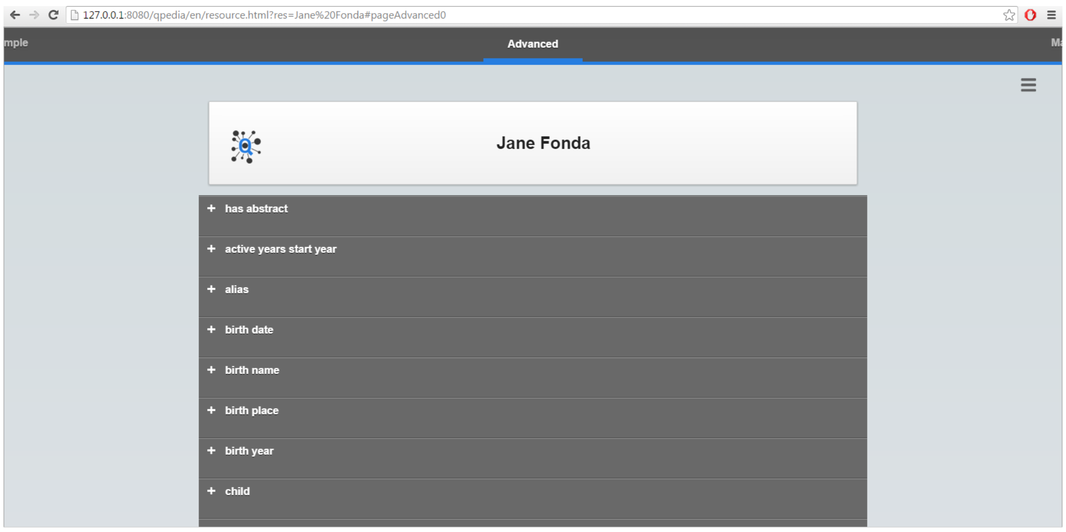
− Jane Fonda

Jane Fonda

+ Jane Fonda filmography

Università degli studi di Cagliari - a.a. 2013 2013 - powered by jQueryMobile 1.3.0
owners of data and informations are Wikipedia and DBpedia

Figure 6.2: Qpedia 2 step: Selecting the appropriate subject between results


127.0.0.1:8080/qpedia/en/resource.html?res=Jane%20Fonda#pageAdvanced0

Advanced

Jane Fonda

+ has abstract
+ active years start year
+ alias
+ birth date
+ birth name
+ birth place
+ birth year
+ child

Figure 6.3: Qpedia 3 step: Finding the desired property to answer the question

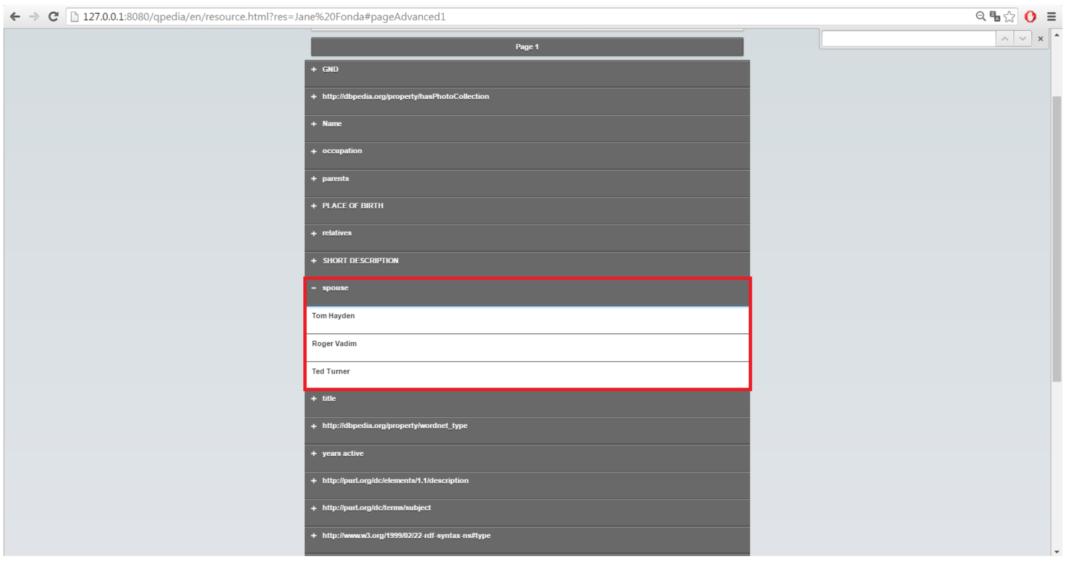

127.0.0.1:8080/qpedia/en/resource.html?res=Jane%20Fonda#pageAdvanced1
Page 1
+ GND
+ http://dbpedia.org/property/hasPhotoCollection
+ Name
+ occupation
+ parents
+ PLACE OF BIRTH
+ relatives
+ SHORT DESCRIPTION
− spouse
Tom Hayden
Roger Vadim
Ted Turner
+ title
+ http://dbpedia.org/property/wordnet_type
+ years active
+ http://purl.org/dc/elements/1.1/description
+ http://purl.org/dc/terms/subject
+ http://www.w3.org/1999/02/22-rdf-syntax-ns#type

Figure 6.4: Qpedia 4 step: Reading the answer into the contents of selected property

## 6.3 TagProp

As there was no evidence of any other systems like TagProp, it has been evaluated in over 20 English questions from different QALD, to show how this system functions. TagProp is not a real QA system, so its goal is to find the properties about text sent in input. There are a subset of questions that have been solved simply by TagProp and they are listed as follows:

- Q1: Who created Wikipedia?
- Q2: What is the religion of Tom Cruise?
- Q3: How many people live in Barcelona?
- Q4: Where was Barack Obama born?
- Q5: When did Abraham Lincoln die?

An example of output in detail about the first question:

**Q1: Who created Wikipedia?**

```
test 0 type request text
[[('Who', 'WP'),
('created', 'VBN'),
('Wikipedia', 'NNP')]]

test 1 words filtered
[u'creator', 'Wikipedia']

test 2 entity:Wikipedia
test 3 properties ok - n.58
test 4 label ok - n.58
test 5 word in properties ok - n.0
test 6 syn in properties ok - n.3
test 7 swoogle tool in properties ok - n.3

Time elapsed in seconds:  7.203

Results in Json:
```

```
[{"word": "creator", "prop": "http://dbpedia.org/property/author"},
{"word": "creator", "prop": "http://dbpedia.org/property/widthUnits"},
{"word": "creator", "prop": "http://dbpedia.org/ontology/author"}]
```

Now consider the following simple SPARQL query:

```
SELECT ?o
WHERE {
   dbr:Wikipedia <http://dbpedia.org/property/author> ?o
}
```

this query yields as result the answer of Q1 on [4].

[4]http://dbpedia.org/sparql

About the second question:

**Q2: What is the religion of Tom Cruise?**

```
test 0 type request text
[[('is', 'VBZ'), ('the', 'DT'),
('religion', 'NN'), ('of', 'IN'), ('tom', 'NN'),
('cruise', 'NN')]]

test 1 words filtered
['religion']

test 2 entity:Tom Cruise
test 3 properties ok - n.60
test 4 label ok - n.31
test 5 word in properties ok - n.1
test 6 syn in properties ok - n.1
test 7 swoogle tool in properties ok - n.1

Time elapsed in seconds:  7.806

Results in Json:
```

[{"word": "religion", "prop": "http://dbpedia.org/property/religion"}]

Now consider the following simple SPARQL query:

```
SELECT ?o
WHERE {
   dbr:Tom_Cruise <http://dbpedia.org/property/religion> ?o
}
```

this query yields as result the answer of Q2.

The result output about third question:

**Q3: How many people live in Barcelona?**

```
test 0 type request int
[[('people', 'NNS'),
('live', 'VBP'), ('in', 'IN'),
('barcelona', 'NN')]]

test 1 words filtered
['people', 'live']

test 2 entity:Barcelona
test 3 properties ok - n.318
test 4 label ok - n.58
test 5 word in properties ok - n.0
test 6 syn in properties ok - n.5
test 7 swoogle tool in properties ok - n.5


Time elapsed in seconds:  25.161

Results in Json:
```

```
[
{"word": "people", "prop":
"http://dbpedia.org/property/populationDensityKm"},
{"word": "people", "prop": "http://dbpedia.org/property/populationMetro"},
{"word": "people", "prop": "http://dbpedia.org/property/populationTotal"},
{"word": "people", "prop": "http://dbpedia.org/property/populationUrban"},
{"word": "people", "prop": "http://dbpedia.org/ontology/populationTotal"},
]
```

Now consider the following simple SPARQL query:

```
SELECT ?o
WHERE {
    dbr:Barcelona
    <http://dbpedia.org/property/populationTotal> ?o
}
```

this query yields as result the answer of Q3.

The output about fourth question:

**Q4: Where was Barack Obama born?**

```
test 0 type request place
[[('where', 'WRB'), ('was', 'VBD'), ('barack', 'JJ'),
('obama', 'NN'), ('born', 'NN')]]

test 1 words filtered
['born']

test 2 entity:Barack Obama
test 3 properties ok - n.163
test 4 label ok - n.3
test 5 word in properties ok - n.0
test 6 syn in properties ok - n.0
test 7 swoogle tool in properties ok - n.1


Time elapsed in seconds: 23.612

Results in Json:
```

[{"word": "born", "prop": "http://dbpedia.org/ontology/birthPlace"}]

Now consider the following simple SPARQL query:

```
SELECT ?o
WHERE {
   dbr:Barack_Obama
   <http://dbpedia.org/ontology/birthPlace> ?o
}
```

this query yields as result the answer of Q4.

And finally, the output about the last question:

**Q5: When did Abraham Lincoln die?**

```
test 0 type request date
[[('did', 'VBD'), ('abraham', 'NN'),
('lincoln', 'NN'), ('die', 'VB')]]

test 1 words filtered
['die']
test 2 entity:Abraham Lincoln
test 3 properties ok - n.156
test 4 label ok - n.156
test 5 word in properties ok - n.0
test 6 syn in properties ok - n.0
test 7 swoogle tool in properties ok - n.5

Time elapsed in seconds:  20.695

Results in Json:
```

[{"word": "die", "prop": "http://dbpedia.org/property/deathDate"},
{"word": "die", "prop": "http://dbpedia.org/ontology/deathDate"},
{"word": "die", "prop": "http://dbpedia.org/ontology/deathYear"}]

Now consider the following simple SPARQL query:

```
SELECT ?o
WHERE {
    dbr:Abraham_Lincoln
    <http://dbpedia.org/ontology/deathDate> ?o
}
```

this query yields as result the answer of Q5.

Thus, TagProp Algorithm can be summed up as follows:

- The first step through NLTK the sentence in input is analysed.
- The second step the type of request (Five Ws and one H) is identified.
- In the third optional step, the main entity is identified. Thanks to this mechanism the number of eligible properties can be reduced, otherwise the process is different and we consider all of properties about KB (DBpedia in this case).
- In the last step Swoogle has been used because can be not found a match with the term of sentence and the label of properties.
- The result is provided to JSON format for external uses. In many experiments it happens that the numbers of returned properties are many and between these there are not interesting properties. In cases where there are more properties will be interesting to use RankProperties, to select number one property and answer the initial question.

### 6.3.1 Quality of Tagging

This section presents the experimental results in term of accuracy of tagging proposed. The Table 6.5 shows some experiment about accuracy on the previously questions. It can be observed from the table that the time execution is not exceptional but the accuracy, for analyzed examples, is always greater than 50%.

The general formula for calculating the accuracy is the following:

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP}$$

where abbreviations stand for:

- TP = True Positive
- FN = False Negative
- TN = True Negative
- FP = False Positive

And more specifically TP and FN regard relevant elements (properties), TN and FP regard irrelevant elements and TP and FP regard found elements, while FN and TN regard not found elements.

| **Questions** | **Time in sec** | **Accuracy** |
|---|---|---|
| Q1 | 7.203 | 66% |
| Q2 | 7.806 | 100% |
| Q3 | 25.161 | 40% |
| Q4 | 23.612 | 100% |
| Q5 | 20.695 | 100% |

Table 6.5: TagProp Accuracy output

# Chapter 7

# Conclusion And Outlook

This chapter summarizes the main contributions of the thesis and discusses possible directions for further research.

## 7.1 Thesis main contributions

The work in this thesis focuses on some aspects providing a contribution in respect of three fundamental parts.

The first contribution was a new graphical user interface which combines the advantages of both mobile devices and Semantic Web. We have seen how the application works with different search modalities. This proposal and its related prototype *Qpedia* introduce a novel approach to display, query and interact with the Semantic Web from the mobile using well-known gestures, voice recognition, a simple way of introducing constraints and enabling location-based queries based on the user position.

As we have seen, we can leverage the second contribution of this thesis, a ranking property tool developed to sort property positions visualized on *Qpedia*. Different rankings for Knowledge Based properties has been investigated about the second spinneret of this dissertation where different strategies have been presented to ranking RDF properties, based on an MLR framework. Through supervised learning, this proposal provides personalization while allowing automatization once models are trained. Even the training set can be automatically computed for some learning strategies, such as the one based on *SWiPE*. In order to create appropriate models, a set of features has been proposed and evaluated for their efficacy, operating feature selection. Compared with existing approaches, there are improvements in the F-measure and Spearman's rho using the rankings proposed. The experimental results showed that the models created will be able to do better than other systems in literature. For the second problem about ranking it has been planned to explore positive outcomes of appropriate RDF ranking in different applications. Between these, beyond the aforementioned *Qpedia*, it is a possible improvement for Question Answering and disambiguation.

The third and last contribution covered about property tagging with the *TagProp* tool aims at giving a contribution in that direction. This is because thanks to *Qpedia* it is possible to find an engine to navigate with extraordinary precision into knowledge bases, thanks to *Rankproperties* it is possible to sort properties and select them with more precision, and finally thanks by *TagProp* with the last two, it is possible to give a contribution toward Question Answering.

In summary, as demonstrated above there is a clear synergy between *Qpedia*, *TagProp* and *Rankproperties.* The investigation into links between those three contributions will lead to great benefits for Question Answering area.

## 7.2 Directions for further research

Considering the novelty of the arguments covered in this thesis, the work done constitutes only the starting point of a wider research line. Indeed, many improvements and open points need to be solved.

Chapter 3 described a new graphical user interface which combines the advantages of both mobile devices and Semantic Web, detailing how this application functions with different search modalities. Future work will be devoted to extending this application with new features, such as *graph search* through constraints on multiple infoboxes, query composition, and query templates.

Chapter 4 illustrated different strategies to ranking RDF properties, and for the next future, it will be interesting to explore positive outcomes of appropriate RDF ranking in different applications and areas. It will, of course, be necessary to modify the training set, test set, and maybe the choice to prioritize to a number of features.

In Chapter 5 *TagProp* is used as an approach to demonstrate how we can develop a possible system for tagging RDF properties in a natural sentence. Wanting to continue toward a complete Question Answering system, we should apply the method of *TagProp* more quickly and more effectively, if we want to make real progress in this direction. To do this, it is necessary to increment search of synonyms in fastest way with a backend getting faster and an entity recognition always more accurate. Especially, not considering only tagging but also the concept of property ranking, because in the near future it could be interesting to explore positive outcomes of appropriate RDF ranking in other different applications in addition to Question Answering and disambiguation in the context of property tagging.

It is also necessary to look at the limitations of current approaches in order to see how we can improve in the future.
In *Qpedia*, we cannot express queries that are based on more complex pattern matching based, (unions, difference, optional matching, etc.).
Next, always in *Qpedia* we cannot express queries that contain properties not showing up in the list (property/field missing). The same applies to *TagProp* if there is no correct synonym/property for a certain word in a sentence.
Incomplete or incorrect entries in KBs is attributable to a general and frequent problem of incompleteness and inaccuracy that can be imputed to human errors or omissions, when information is generated by crowdsourcing on less-known entities and concepts. Fortunately, for these problems are being addressed by various approaches including text-mining systems such as IBminer, and massive crowdsourcing initiatives such as WikiData.

# Appendix A

# Technical Details on the RankProperties tools

The following appendixes describe the installation and basic configuration of 'RankProperties' and 'TagProp' systems. It covers information on installing these systems with a binary package such as an APT in Ubuntu Linux or a Windows executable. Topics covered here include configuration of the web servers, installation of additional Python modules and tools for other interesting experiments. Follow the detailed steps in this appendix to install RankProperties on your server or computer, and on the following to install 'TagProp'. After that you can then use our tools by localhost or online.
Here are some tools for ranking RDF properties, evaluation systems and creation models of the Knowldege Bases have already seen in the previously chapters (DBpedia, Wikidata, Freebase and Musicbrainz). RDF RankProperties architecture is composed of the following tools:

- Web Server, to evaluate and to make ranking RDF properties.
- Model Generator, to create models for our ranker systems using RankLib.
- Evaluation Tool, to evaluate results and different configurations.

## A.1 Web Server

It is possible to install Web Server within Linux using the ‘apt’ and ‘pip’ systems installer in a bash. First install ‘pip’, a package manager, and ‘virtualenv’, a tool to create isolated Python environments.

To do this digit:

```
sudo apt-get install python-setuptools
sudo apt-get install python-dev
sudo apt-get install python-devel
sudo apt-get install sqlite3 libsqlite3-dev
sudo apt-get install git
sudo easy_install pip
sudo easy_install virtualenv
```

Create a virtual environment:

```
virtualenv venv
```

This creates a copy of Python in whichever directory you ran the command in, placing it in a folder named ‘venv’.

To begin using the virtual environment, it needs to be activated:

```
source venv/bin/activate
```

After this, follow these instructions to install RankProperties dependencies:

```
pip install SPARQLWrapper
pip install numpy
pip install requests
pip install pyyaml nltk
python -m nltk.downloader all
pip install pyenchant (or sudo apt-get install libenchant1c2a)
```

You can then begin installing any new modules without affecting the system default Python or other virtual environments.

If you have concluded working in the virtual environment for the moment, you can deactivate it:

```
deactivate
```

This puts you back to the system's default Python interpreter with all its installed libraries. To delete a virtual environment, just delete its folder. After a while, though, you might end up with a lot of virtual environments littered across your system, and its possible youll forget their names or where they were placed.

If you want to use RankProperties from Python Shell follow these instructions.

The RankProperties command line interface is available via the Python Shell. See the next example to see how to discover all of the functionality.

To run three quickstart examples you'll need to make:

- Download all source code in your virtual environment directory from bitbucket repository[1].
- Open a shell, go into "web_server" folder.

If you want to test this tool, create a python simple script looks like the following scripts.

First script:

```
from rankProperties import rankProp

prop = "http://dbpedia.org/property/populationTotal"
entity = ""

print rankProp(prop,entity)
```

Second script:

```
from rankProperties import rankProp

prop = "http://dbpedia.org/property/populationTotal"
entity = "Cagliari"

print rankProp(prop,entity)
```

[1]https://andrea_dessi@bitbucket.org/semanticweb/rankproperties.git

Third script:

```
from rankProperties import rankAllDBpediaPropByJSON

entity = "Rose"
frequencyTF = True
frequencyEntTF = True
numOfWTF = True
nitTF = True
isEnTF = True
cOPTF =  True
isLinkTF =  True
goog2TF =  True
isInSwipeTF =  False
evalAll =  True

print rankAllDBpediaPropByJSON(entity,frequencyTF,frequencyEntTF,
                               numOfWTF,nitTF,isEnTF,cOPTF,
                               isLinkTF,goog2TF,isInSwipeTF,evalAll)
```

where the method's parameters are:

- entity: an entity of DBpedia
- property: a property about entity or a single property
- name_featureTF: enable (True) or not (False) a choosen features

After that, save the script, i.e., 'script.py', and executes it with python 'script.py'.

For example if you want to obtain all properties about a particular knowledge base, create a python script which contains:

```
from rankProperties import rankPropByUrl

url = "http://www.wikidata.org/wiki/Q1897"

print rankPropByUrl(url)
```

where the method's parameters are:

- url: an url between four Knowledge Bases like wikidata, dbpedia, freebase and musicbrainz

save the script, i.e., 'script.py', and executes it with python script.py

Otherwise, if you want to obtain all properties about a particular knowledge base ranked, create a python script which contains:

```
from rankProperties import rankAllPropByUrl

url = "http://www.wikidata.org/wiki/Q1897"
modal = 3
algo = 8

print rankAllPropByUrl(url,modal,algo)
```

where the method's parameters are:

- url: an url between four Knowledge Bases like wikidata, dbpedia, freebase and musicbrainz
- modal: modality about model generator
- algo: a MLR algorithm number (see under)

save the script, i.e., 'script.py', and executes it with python script.py

## A.2 Model Generator

Model Generator is a tool to create models for RankProperties. If it has been installed the web server it can jump these rows up to where there is how to use from Python Shell.

It is possible to install Web Server within Linux using the 'apt' and 'pip' systems installer in a bash. First install 'pip', a package manager, and 'virtualenv', a tool to create isolated Python environments.

To do this digit:

```
sudo apt-get install python-setuptools
sudo apt-get install python-dev
sudo apt-get install python-devel
sudo apt-get install sqlite3 libsqlite3-dev
sudo apt-get install git
sudo easy_install pip
sudo easy_install virtualenv
```

Create a virtual environment:

```
virtualenv venv
```

This creates a copy of Python in whichever directory you ran the command in, placing it in a folder named 'venv'.

To begin using the virtual environment, it needs to be activated:

```
source venv/bin/activate
```

After this, follow these instructions to install RankProperties dependencies:

```
pip install SPARQLWrapper
pip install numpy
pip install requests
pip install pyyaml nltk
python -m nltk.downloader all
pip install pyenchant (or sudo apt-get install libenchant1c2a)
```

You can then begin installing any new modules without affecting the system default Python or other virtual environments.

If you have concluded working in the virtual environment for the moment, you can deactivate it:

```
deactivate
```

This puts you back to the system's default Python interpreter with all its installed libraries. To delete a virtual environment, just delete its folder. After a while, though, you might end up with a lot of virtual environments littered across your system, and its possible youll forget their names or where they were placed.

Now it is possible to see how to use from Python Shell.

The 'Model Generator' command line interface is available via the Python Shell. See the next example to see how to discover all of the functionality.

To run a quickstart example you'll need to make:

**A.** Download all source code in your virtual environment directory.

git clone `https://atzori@bitbucket.org/semanticweb/rankproperties.git`

or

git clone `https://andrea_dessi@bitbucket.org/semanticweb/rankproperties.git`

**B.** Open a shell, go into 'model_generator' and digit 'python'

**C.** Create a python script which contains:

```
from rankProperties import testBedAJournal

entities_train = {1 : "Russelia", 2 : "Parma", 3 : "Microsoft",
4 : "Dog" , 5 : "Enrico_Berlinguer", 6 : "Sandra_Bullock"}
entities_vali = {1 : "Rose", 2 : "Cagliari", 3 : "Facebook",
4 : "Cat" , 5 : "Aldo_Moro", 6 : "Angelina_Jolie"}
# different entities possibly of different types
entities_test = {1 : "Pablo_Picasso", 2 : "Monaco",
3 : "Born_to_Love", 4 : "Conus" ,
5 : "The_Freddie_Mercury_Album", 6 : "Jean_de_Quen"}

ontologies_train = {1 : "Species", 2 : "Settlement",
3 : "Public_company", 4 : "Animal" ,
5 : "Politician", 6 : "Agent"}
ontologies_vali = {1 : "Species", 2 : "Settlement",
3 : "Public_company", 4 : "Animal" , 5 : "Politician",
6 : "Agent"}
ontologies_test = {1 : "Artist", 2 : "Place", 3 : "Work",
4 : "Animal" , 5 : "Work", 6 : "Agent"}

method = 3 # frequency

pos = True

testBedAJournal(method,entities_train,entities_vali,entities_test,
ontologies_train,ontologies_vali,ontologies_test,pos)
```

where the method's parameters are:

- entities_train, entities_vali and entities_test: three lists which contain a list of entities that we want to use to create training files.
- ontologies_train, ontologies_vali, ontologies_test: three lists which contain respectively a list of ontologies of entities chosen on previously lists (Sort elements depending on entity lists).
- method: an integer which setting method between 1.expert (expert), 2.questionnaire (quest), 3.frequency (freq), 4.ddgsuggest (ddg), 5.typicality (scor) and 6.swipe.
- pos: a boolean value which addresses how we want setting class value on training file (True sets positions, False sets values obtained with method choosen).

save the script, i.e., 'script.py', and executes it with 'python script.py'

If you want to try this example directly without parameters create a python script which contains:

```
from rankProperties import runTestBedAJournal()

runTestBedAJournal()
```

In output we obtain three file txt called dtrain_'method'.txt, dvali_'method'.txt and dtest_'method'.txt which contain files for RankLib tool (more information on [2]) to create our models in SVM format:

```
<line> .=. <target> qid:<qid> <feature>:<value> <feature>:<value> ...
<feature>:<value> # <info>
<target> .=. <positive integer>
<qid> .=. <positive integer>
<feature> .=. <positive integer>
<value> .=. <float>
<info> .=. <string>
```

The target column is obtained by 'pos' parameter.

[2]http://sourceforge.net/p/lemur/wiki/RankLib/

**D.** Finally create last python script which contains:

```
from rankProperties import testBedABJournal()

suffix = "freq"    #for example
testBedABJournal(suffix)
```

N.B. If you want to use method 1.expert (expert) or 2.questionnaire (quest) is necessary to modify methods appropriated.

Eight models, based on previously training files (dtest, dtrain, and dvali) into experiments folder, one for each learning to rank algorithms:

- RankNet (1st)
- RankBoost (2nd)
- AdaRank (3rd)
- Coordinate Ascent (4th)
- LambdaMART (5th)
- MART (Multiple Additive Regression Trees, a.k.a. Gradient boosted regression tree) (6th)
- ListNet (7th)
- Random Forests (8th)

## A.3 Evaluation Tool

Evaluation Tool is a tool to compare ranking by using Spearman's rank correlation rho, Precision and Recall of Top-10 Attributes

It is possible to install Web Server within Linux using the 'apt' and 'pip' systems installer in a bash. First install 'pip', a package manager, and 'virtualenv', a tool to create isolated Python environments.

To do this digit:

```
sudo apt-get install python-setuptools
sudo apt-get install python-dev
sudo apt-get install python-devel
sudo apt-get install sqlite3 libsqlite3-dev
```

```
        sudo apt-get install git
        sudo easy_install pip
        sudo easy_install virtualenv
```

Create a virtual environment:

```
virtualenv venv
```

This creates a copy of Python in whichever directory you ran the command in, placing it in a folder named ‘venv’.

To begin using the virtual environment, it needs to be activated:

```
source venv/bin/activate
```

After this, follow these instructions to install RankProperties dependencies:

```
pip install SPARQLWrapper
pip install numpy
pip install requests
pip install pyyaml nltk
python -m nltk.downloader all
pip install pyenchant (or sudo apt-get install libenchant1c2a)
```

You can then begin installing any new modules without affecting the system default Python or other virtual environments.

If you have concluded working in the virtual environment for the moment, you can deactivate it:

```
deactivate
```

This puts you back to the system’s default Python interpreter with all its installed libraries. To delete a virtual environment, just delete its folder. After a while, though, you might end up with a lot of virtual environments littered across your system, and its possible youll forget their names or where they were placed.

Now it is possible to see how to use from Python Shell.

The CompareAllRanking command line interface is available via the Python Shell. See the next example to see how to discover all of the functionality.

To run a quickstart example you'll need to make:

- Download all source code in your virtual environment directory.

  git clone `https://atzori@bitbucket.org/semanticweb/rankproperties.git`

  or

  git clone `https://andrea_dessi@bitbucket.org/semanticweb/rankproperties.git`

- Open a shell, go into "evaluation_tool" folder and digit *python*
- Create a python script which contains:

```
from rankProperties import CompareAllRanking

post = [{ #The_Guardian
         'http://dbpedia.org/ontology/wikiPageExternalLink': 1,
         'http://dbpedia.org/property/caption': 3,
         'http://dbpedia.org/property/circulation': 3,
         'http://dbpedia.org/property/cost': 2,
         'http://dbpedia.org/property/editor': 4,
         'http://dbpedia.org/property/format': 4,
         'http://dbpedia.org/property/foundation': 4,
         'http://dbpedia.org/property/headquarters': 3,
         'http://dbpedia.org/property/issn': 2,
         'http://dbpedia.org/property/language': 2,
         'http://dbpedia.org/property/name': 2,
         'http://dbpedia.org/property/oclc': 1,
         'http://dbpedia.org/property/owners': 4,
         'http://dbpedia.org/property/publisher': 4,
         'http://dbpedia.org/property/type': 3,
         'http://dbpedia.org/property/website': 4,
         'http://dbpedia.org/property/opeditor': 3,
         'http://dbpedia.org/property/political': 4,
         'http://dbpedia.org/property/sisterNewspapers': 3,
         'http://dbpedia.org/ontology/editor': 4,
         'http://dbpedia.org/ontology/format': 3,
         'http://dbpedia.org/ontology/headquarter': 3,
         'http://dbpedia.org/ontology/owner': 4,
         'http://dbpedia.org/ontology/wikiPageWikiLink': 1,
         'http://dbpedia.org/ontology/circulation': 3,
         'http://dbpedia.org/ontology/abstract': 3,
         'http://dbpedia.org/ontology/wikiPageLength': 1,
         'http://dbpedia.org/ontology/wikiPageOutDegree': 1,
         'http://dbpedia.org/ontology/wikiPageRevisionID': 1,
         'http://dbpedia.org/ontology/sisterNewspaper': 3,
         'http://dbpedia.org/ontology/wikiPageID': 1}]


entities_scoring = ['The_Guardian']

ontologies_scoring = ['PeriodicalLiterature']

CompareAllRanking(entities_scoring,ontologies_scoring,post)
```

where the method's parameters are:

- entities_scoring: an array which contains a list of entities that we want to analyze them.
- ontologies_scoring: an array which contains a list of ontologies of entities chosen on array entities_scoring (Sort elements depending on entities_scoring).
- post: an array which contains a list of properties belonging to entities_scoring items and ordered as you want (to create post vector use createVectorCompareAllRanking(vect), where vect is the vector which contains a list of entity chosen).

  save the script, i.e., 'script.py', and executes it with 'python script.py'

  If you want to try this example directly without parameters create a python script which contains:

  ```
  from rankProperties import testCompareAllRanking

  testCompareAllRanking()
  ```

In output we obtain four files csv called *experiments_spearman.csv*, *experiments_precision.csv*, *experiments_recall.csv* and *experiments_fmeasure.csv* which contain a table into experiments_journal directory made as follows:

```
| entity | Ty1 | Ty2 | Ty3 | Random | Lexicographic
| Lexicographic Rev | Swipe | Swipe2
| **RP_modality_NumberOfAlgorithm_NumberOfFeatures**** |
```

- entity: contains the list of entities chosen
- Ty1: Typicality $P(i|a)$
- Ty2: Typicality $P(a|c)$
- Ty3: Typicality $P(c|a)$
- Random: Random sort
- Lexicographic: Lexicographical sort
- Lexicographic Rev: Lexicographical sort reverse
- Swipe: Swipe sort
- Swipe 2: Swipe sort with different classification

**The last column is a summary of 40 columns because the name is composed of:

- RP :: RankProperties
- modality :: ['expert mode', 'frequency mode', 'quest mode', 'google suggest mode', 'scoring typicality mode', 'duckduckgo suggest mode', 'swipe mode']
- NumberOfAlgorithm :: [1-8] algorithms of RankLib
- NumberOfFeatures :: [1-9] features of RankProperties (for more details see Ranking DBpedia Properties)

In each column we will find the corresponding rho, precision, recall and f-measure respectively, than sorting chosen on the variable *post* (the third parameter of the method).

# Appendix B

# Technical Details on the TagProp tools

TagProp is a powerful tool that is able to identify parts of free-text and link them to a pertinent RDF Properties (only DBpedia now) in a fast and effective way. This annotation process has implications which go far beyond the enrichment of the text with explanatory links because it concerns a way to answer questions inside of the input typed text. Currently TagProp is available in English and it is based on last Wikipedia snapshots.

TagProp architecture is composed of the following parts:

- Web Server, to make tagging RDF properties with a GUI.
- Tools and Utilities.

## B.1 Web Server

You can install Web Server within Linux using the apt and pip systems installer in a bash. First install pip, a package manager, and virtualenv, a tool to create isolated Python environments.

To do this digit:

```
sudo apt-get install python-setuptools
sudo apt-get install python-dev
sudo apt-get install python-devel
sudo apt-get install sqlite3 libsqlite3-dev
sudo apt-get install git
sudo easy_install pip
sudo easy_install virtualenv
```

Create a virtual environment:

```
virtualenv venv
```

This creates a copy of Python in whichever directory you ran the command in, placing it in a folder named venv.

To begin using the virtual environment, it needs to be activated:

```
source venv/bin/activate
```

After this, follow these instruction to install tagProp dependencies:

```
pip install SPARQLWrapper
pip install numpy
pip install requests
pip install pyyaml nltk
sudo pip install requests beautifulsoup4 inflect
python -m nltk.downloader all
pip install pyenchant
```

You can then begin installing any new modules without affecting the system default Python or other virtual environments.

If you are done working in the virtual environment for the moment, you can deactivate it:

```
deactivate
```

This puts you back to the systems default Python interpreter with all its installed libraries. To delete a virtual environment, just delete its folder. After a while, though, you might end up with a lot of virtual environments littered across your system, and its possible youll forget their names or where they were placed.

If you want to use TagProp from Python Shell follow these instructions.

The tagProp command line interface is available via the Python Shell. See the next example to see how to discover all of the functionality.

To run three quickstart examples you'll need to make:

- Download all source code in your virtual environment directory from bitbucket repository[1].
- Open a shell, go into tagProp folder.
- if you want to test this tool, create a python simple script looks like the following.

```
from tagProp import tagClassesNewCaching

print tagClassesNewCaching("How many people live in Cagliari?")
```

After that, save the script, i.e., 'script.py', and executes it with python 'script.py'.

The tool to compute property tagging is also available as a Web API.

If you want to test this tool, open a shell, go into tagProp folder and execute a python script:

```
python httpServer.py
```

In order to use it, given a sentence (e.g., How many people in Cagliari?"), you can compute the tagging of the sentence by using our GUI which elaborates the following exemplificatory url (localhost):

```
http://127.0.0.1:9999/?sentence=[your_typed_sentence]
```

The result is provided in JSON format and visualized in our GUI, such that it can be even used within other projects online.

An example of JSON output about tagProp:

```
[{"prop":"http://dbpedia.org/property/populationTotal",
"word":"people"},
{"prop":"http://dbpedia.org/ontology/populationTotal",
"word":"people"}]
```

[1]https://andrea_dessi@bitbucket.org/semanticweb/tagproperties.git

## B.2 Tools and Utilities

- Label Extraction.
- Synonyms Extraction.
- tagProp GUI.
- Qald Extended.
- Eval TagProp over QALD.

### B.2.1 Label Extraction

Download RDF properties and their English labels:

```
python labelExtraction.py > labels.tsv
```

The content of labels.tsv will be like the following:

```
http://dbpedia.org/property/populationTotal population total
another_property_uri    label1  label2  ...
```

That is, a property URI followed by its label (or labels), if any.

### B.2.2 Synonyms Extraction

Creates a file with English synonyms

```
python synonymsExtraction.py > synonyms.tsv
```

The content of synonyms.tsv will be like the following:

```
people<tab>population   citizens    inhabitants<tab>...
another_property_uri    label1 label2  ...
```

That is, every row contains a list of words with same or correlated meaning.

### B.2.3 TagProp GUI

A Graphical User Interface for tagProp tool. After, install LAMP, copy GUI directory into the directory of apache server, enter in this folder and modify row number 10 on process.php file setting with the chosen address (ip and port) i.e. like this in localhost

```
$json = file_get_contents('http://localhost:9999/tagprop/
?sentence=%22'.$input_for_url.'%22');
```

Now, open a browser and digit:

```
http://localhost/tagprop/
```

You will see a graphical user interface where you write on text-area your sentence. E.g. For the sentence "How many people live in Cagliari?", clicking over "tagProp" button, our web service produces a JSON output like this:

```
[{"prop":"http://dbpedia.org/property/populationTotal","word":"people"},
{"prop":"http://dbpedia.org/ontology/populationTotal","word":"people"}];
```

and you'll see a sentence above with one or more links about the previously properties. If you want, you see video explication on wiki home

### B.2.4 Qald Extended

Return a JSON file from Qald file with all important component of SPARQL query. If you want to change the QALD file which it will be analyzed, go to line 19 of this script and change that code-line whenever you want. To start digit on a shell

```
python re-elaborateQald.py
```

A fragment of JSON result about this tool:

```
[{"entities": ["res:Jane_Fonda"],
"properties": ["dbo:spouse"],
"types": [],
"queries": "PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX res: <http://dbpedia.org/resource/>
SELECT COUNT(DISTINCT ?uri) \nWHERE {
res:Jane_Fonda dbo:spouse ?uri .}"}, {"entities":...
```

# Appendix C

# Technical Details on Qpedia

The following appendixes describe the installation and basic configuration of Qpedia. It covers information on installing these systems with a binary package such as an APT in Ubuntu Linux or a Windows executable. After that you can then use our tools by localhost or online. Under most circumstances, installing Qpedia is a very simple process and takes less than five minutes to complete. Only Apache server is required and a browser with javascript support. Many web hosts now offer tools to automatically install Apache for you. It is possible to install Apache with Windows using i.e. [1] or with Linux using the ‘apt’ and ‘pip’ systems installer in a bash. First install ‘git’, a version control system, to download the source code and second install Apache. To do this digit:

```
sudo apt-get install git
sudo apt-get install apache2
```

Once installed them, it is possible to download the directory using git. To run Qpedia you’ll need to make:

- Download all source code in your virtual environment directory from bitbucket repository [2].
- Open a browser and typing Qpedia (or the folder name chosen) in the location bar.

[1]http://www.easyphp.org/

[2]https://andrea_dessi@bitbucket.org/andrea_dessi/qpedia.git

# Appendix D

# Jena Experiments

This section contains a set of experiments about SPARQL Query optimezed over Apache Jena.

Let's start with a SPARQL query (A):

```
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>


SELECT DISTINCT ?res
WHERE {
?res dbpedia-owl:country ?country .
FILTER (REGEX(STR(?country), "Italy", "i") ).
?res dbpprop:mayor ?mayor .
FILTER (REGEX(STR(?mayor), "Renzi", "i") )
}
```

The estimated time is 10,541 seconds and trying to execute indexes individually take the same time.

But instead, launching the query (B) similar to query (A) with “text:query”:

```
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>

SELECT DISTINCT ?res
WHERE {
?res text:query(dbpedia-owl:country "Italy").
?res dbpprop:mayor ?mayor .
FILTER (REGEX(STR(?mayor), "Renzi", "i") )}
```

The estimated time is more long than before with his 139,803 seconds.

Other experiments performed, launching queries to estimate time, are:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?res
WHERE
{
{?res text:query (dbpprop:type 'Daily newspaper') .
?res dbpprop:type ?var_1 .
FILTER ( REGEX(STR(?var_1), "Daily newspaper" , "i") )}
UNION
{?res dbpprop:type ?var_1X.
?var_1X text:query (foaf:name 'Daily newspaper').
?var_1X foaf:name ?var_1Y.
FILTER ( REGEX(STR(?var_1Y), "Daily newspaper" , "i") )}
}
OFFSET 0 LIMIT 10
```

```
17:28:16 INFO Fuseki :: [2] exec/select
17:28:16 INFO Fuseki :: [2] 200 OK (15,266 s)
```

Same previously query with UNION inverted:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>


SELECT DISTINCT ?res
WHERE {
{
        ?res dbpprop:type ?var_1X.
 ?var_1X text:query (foaf:name 'Daily newspaper').
 ?var_1X foaf:name ?var_1Y.
        FILTER ( REGEX(STR(?var_1Y), "Daily newspaper" , "i") )
}
UNION
{
        ?res text:query (dbpprop:type 'Daily newspaper') .
        ?res dbpprop:type ?var_1 .
        FILTER ( REGEX(STR(?var_1), "Daily newspaper" , "i") )
}
}

OFFSET 0 LIMIT 10


19:21:20 INFO Fuseki :: [1] exec/select
19:24:02 INFO Fuseki :: [1] 200 OK (227,686 s)
```

Only first part of previously query (above part of UNION):

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>


SELECT DISTINCT ?res
WHERE {
{
?res dbpprop:type ?var_1X.
?var_1X text:query (foaf:name 'Daily newspaper').
?var_1X foaf:name ?var_1Y.
FILTER ( REGEX(STR(?var_1Y), "Daily newspaper" , "i") )
}
}
OFFSET 0 LIMIT 10


16:24:39 INFO Fuseki :: [4] exec/select
16:27:17 INFO Fuseki :: [4] 200 OK (223,883 s)


(without filter)


17:05:35 INFO Fuseki :: [1] exec/select
17:05:38 INFO Fuseki :: [1] 200 OK (15,862 s)
```

Second part ot the second query (below part of UNION):

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>


SELECT DISTINCT ?res
WHERE {
{
?res text:query (dbpprop:type 'Daily newspaper') .
?res dbpprop:type ?var_1 .
FILTER ( REGEX(STR(?var_1), "Daily newspaper" , "i") )
}
}

OFFSET 0 LIMIT 10

16:30:58 INFO Fuseki :: [1] exec/select
16:30:59 INFO Fuseki :: [1] 200 OK (12,444 s)

(without filter)

17:02:21 INFO Fuseki :: [1] exec/select
17:02:21 INFO Fuseki :: [1] 200 OK (11,706 s)

(without fulltext search)

17:03:45 INFO Fuseki :: [1] exec/select
17:03:45 INFO Fuseki :: [1] 200 OK (5,753 s)
```

Previously query with ORDER BY:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>


SELECT DISTINCT ?res
WHERE {
{
?res text:query (dbpprop:type 'Daily newspaper') .
?res dbpprop:type ?var_1 .
FILTER ( REGEX(STR(?var_1), "Daily newspaper" , "i") )
}
?res dbpprop:editor ?editor.
}
ORDER BY ASC(?editor)
OFFSET 0 LIMIT 10

15:19:02 INFO Fuseki :: [2] exec/select
15:19:02 INFO Fuseki :: [2] 200 OK (193,994 s)
```

Other queries similar to previously with ORDER BY:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>


SELECT DISTINCT ?res
WHERE {
{
?res dbpprop:type ?var_1X.
 ?var_1X text:query (foaf:name 'Daily newspaper').
 ?var_1X foaf:name ?var_1Y.
FILTER ( REGEX(STR(?var_1Y), "Daily newspaper" , "i") )
}
?res dbpprop:editor ?editor.
}
ORDER BY ASC(?editor)
OFFSET 0 LIMIT 10


Time > 1200 s
```

```
--

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?res
WHERE {
{
?res dbpprop:type ?var_1X.
 ?var_1X text:query (foaf:name 'Daily newspaper').
 ?var_1X foaf:name ?var_1Y.
FILTER ( REGEX(STR(?var_1Y), "Daily newspaper" , "i") )
}
UNION
{
?res text:query (dbpprop:type 'Daily newspaper') .
?res dbpprop:type ?var_1 .
FILTER ( REGEX(STR(?var_1), "Daily newspaper" , "i") )
}
?res dbpprop:editor ?editor.
}
ORDER BY ASC(?editor)
OFFSET 0 LIMIT 10

Time > 1200 s
```

In the following query using "REGEX":

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>


SELECT DISTINCT ?res ?editor
WHERE {
{
?res text:query (dbpprop:type 'Daily newspaper') .
?res dbpprop:type ?var_1 .
FILTER ( REGEX(STR(?var_1), "Daily newspaper" , "i") ).
?res dbpprop:editor ?editor.
}
UNION
{
?res dbpprop:type ?var_1X.
?var_1X text:query (foaf:name 'Daily newspaper').
?var_1X foaf:name ?var_1Y.
FILTER ( REGEX(STR(?var_1Y), "Daily newspaper" , "i") ).
?res dbpprop:editor ?editor.
}
}
#ORDER BY ASC(?editor)
OFFSET 0 LIMIT 10


16:10:48 INFO Fuseki :: [1] exec/select
16:10:49 INFO Fuseki :: [1] 200 OK (14,174 s)


With "ORDER BY" - Time > 1200
```

In the following using *"REGEX"*:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>


SELECT DISTINCT ?res ?editor
WHERE {
?res text:query (dbpprop:type 'Daily newspaper') .
?res dbpprop:type ?var_1 .
FILTER ( REGEX(STR(?var_1), "Daily newspaper" , "i") ).
?res dbpprop:editor ?editor.

}
16:31:17 INFO Fuseki :: [1] exec/select
16:34:33 INFO Fuseki :: [1] 200 OK (208,239 s)
```

An other experiment trying to find the “dbprop UNION foaf:name” which contains “Daily newspaper”. In this case the filter is always for “Daily newspaper” with ORDER BY

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>

SELECT DISTINCT ?res
WHERE {{
?res text:query (dbpprop:type 'Daily newspaper') .
?res dbpprop:type ?var_1 .
FILTER ( REGEX(STR(?var_1), "Daily newspaper" , "i") )
}
UNION {
?res dbpprop:type ?var_1X.
?var_1X text:query (foaf:name 'Daily newspaper').
?var_1X foaf:name ?var_1Y.
FILTER ( REGEX(STR(?var_1Y), "Daily newspaper" , "i") )
}
?res dbpprop:editor ?editor.
}
ORDER BY ASC(?editor)
OFFSET 0 LIMIT 10

Time > 1200
```

All of analyzed queries in this scenario with ORDER BY and without OFFSET present an execution time undefined (Time more than 1200 s). Using ”ORDER BY” and executing it without clean the cache, the time is equal to 2 seconds.

Instead cleaning the cache:

```
16:40:15 INFO Fuseki :: [1] exec/select
16:40:17 INFO Fuseki :: [1] 200 OK (222,466 s)
```

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>

SELECT DISTINCT ?res ?editor ?image ?comment ?page ?label
WHERE {
{
?res text:query (dbpprop:type 'Daily newspaper') .
?res dbpprop:type ?var_1 .
FILTER ( REGEX(STR(?var_1), "Daily newspaper" , "i") )}
UNION{
?res dbpprop:type ?var_1X.
?var_1X text:query (foaf:name 'Daily newspaper').
?var_1X foaf:name ?var_1Y.
FILTER ( REGEX(STR(?var_1Y), "Daily newspaper" , "i") ).}
?res dbpprop:editor ?editor.
?res dbpprop:editor ?editor.
OPTIONAL { ?res foaf:page ?page }
OPTIONAL { {?res dbpedia-owl:thumbnail ?image} }
OPTIONAL {
?res rdfs:comment ?comment.
        FILTER (LANG(?comment) = "en").}
OPTIONAL{
        ?res rdfs:label ?label.
        FILTER (LANG(?label) = "en") .}
}
OFFSET 0 LIMIT 10

16:53:43 INFO Fuseki :: [1] exec/select
16:53:45 INFO Fuseki :: [1] 200 OK (18,398 s)
```

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>
PREFIX dbpprop:<http://dbpedia.org/property/>
PREFIX foaf:<http://xmlns.com/foaf/0.1/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>


SELECT DISTINCT ?res ?editor ?image ?comment ?page ?label
WHERE {
{        ?res text:query (dbpprop:type 'Daily newspaper') .
?res dbpprop:type ?var_1 .
FILTER ( REGEX(STR(?var_1), "Daily newspaper" , "i") )
}
UNION
{        ?res dbpprop:type ?var_1X.
?var_1X text:query (foaf:name 'Daily newspaper').
?var_1X foaf:name ?var_1Y.
FILTER ( REGEX(STR(?var_1Y), "Daily newspaper" , "i") ).
}
UNION
{
?res dbpprop:language ?var_2X.
?var_2X text:query (foaf:name 'Italian language').
?var_2X foaf:name ?var_2Y.
FILTER ( REGEX(STR(?var_2Y),"Italian language" , "i") )
}
UNION
{
?res dbpprop:language ?var_2X.
?var_2X text:query (rdfs:label 'Italian language').
?var_2X rdfs:label ?var_2Y.
FILTER ( REGEX(STR(?var_2Y),"Italian language" , "i") )
}
?res dbpprop:editor ?editor.
OPTIONAL { ?res foaf:page ?page }
OPTIONAL { {?res dbpedia-owl:thumbnail ?image} }
OPTIONAL { ?res rdfs:comment ?comment.
       FILTER (LANG(?comment) = "en"). }
OPTIONAL { ?res rdfs:label ?label.
        FILTER (LANG(?label) = "en") .}
}
#ORDER BY ?editor
OFFSET 0 LIMIT 10

17:06:16 INFO Fuseki :: [1] exec/select
17:06:18 INFO Fuseki :: [1] 200 OK (25,375 s)
```

In the next experiment can figure out what's up searching the label which contains "ca" with wildcard (*):

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>

select distinct ?p str(?val) ?data where
{
?data text:query(rdfs:label 'ca*').
?data ?p ?val
}
LIMIT 10

16:09:28 INFO Fuseki :: [1] exec/select
16:09:28 INFO Fuseki :: [1] 200 OK (3,111 s)
```

And with sort:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>

select distinct ?p str(?val) ?data where
{
?data text:query(rdfs:label 'ca*').
?data ?p ?val
}
order by ?p
LIMIT 10

16:21:20 INFO Fuseki :: [1] exec/select
16:21:20 INFO Fuseki :: [1] 200 OK (642,267 s)
```

Experiments with sort over ?lab and filter over language always on ?lab:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>

SELECT DISTINCT ?s ?lab
WHERE{
?s text:query(rdfs:label 'ca*').
?s rdfs:label ?lab.
FILTER (lang(?lab) = "en").
}
ORDER BY ?lab
LIMIT 10

16:55:50 INFO Fuseki :: [2] exec/select
16:55:50 INFO Fuseki :: [2] 200 OK (164,488 s)
```

```
--

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>

SELECT DISTINCT ?s ?lab
WHERE{
?s text:query(rdfs:label 'ca*').
?s rdfs:label ?lab.
FILTER (lang(?lab) = "en").
}
LIMIT 10

16:54:44 INFO Fuseki :: [4] exec/select
16:54:45 INFO Fuseki :: [4] 200 OK (559 ms)
```

The instruction ORDER BY is expensive, so this is something which we also have to take into account.

Using the previously query adding the condition data ¡ 1950-01-01:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>



SELECT DISTINCT ?s ?lab ?birthDate
WHERE{ ?s text:query(rdfs:label 'ca*').
?s rdfs:label ?lab.
?s <http://dbpedia.org/ontology/birthDate> ?birthDate .
FILTER (lang(?lab) = "en").
FILTER(?birthDate < "1950-01-01").
}
LIMIT 10
19:28:20 INFO Fuseki :: [1] exec/select
19:28:20 INFO Fuseki :: [1] 200 OK (239,544 s)
```

Sorting by ?lab:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX text: <http://jena.apache.org/text#>

SELECT DISTINCT ?s ?lab ?birthDate
WHERE{
?s text:query(rdfs:label 'ca*').
?s rdfs:label ?lab.
?s <http://dbpedia.org/ontology/birthDate> ?birthDate .
FILTER (lang(?lab) = "en").
}
ORDER BY ?lab
LIMIT 10

19:37:38 INFO Fuseki :: [1] exec/select
19:37:38 INFO Fuseki :: [1] 200 OK (244,624 s)
```

In the following experiment were used other datasets:

```
PREFIX imdb: <http://data.linkedmdb.org/resource/movie/>
PREFIX dbpedia: <http://dbpedia.org/ontology/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT *
from <http://xmlns.com/foaf/0.1/>
{
SERVICE <http://data.linkedmdb.org/sparql>
{
?actor1 imdb:actor_name "Tom Hanks".
?movie imdb:actor ?actor1 ;
dcterms:title ?movieTitle .
}
SERVICE <http://dbpedia.org/sparql>
{
?actor rdfs:label "Tom Hanks"@en ;
dbpedia:birthDate ?birth_date .
}
}
```

```
Execution Time (8,634 s)
```

In the following query looks like the instruction FILTER flows all of indexes even if italian language “@IT” were not loaded.

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT *
WHERE {
?subject rdf:type <http://dbpedia.org/ontology/City>.
?subject rdfs:label ?label.
FILTER ( lang(?label) = 'it')
}
LIMIT 10
```

```
Time (56,032 s)
```

Instead in this query were exist english language en and therefore was fastest:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>


SELECT *
WHERE {
?subject rdf:type <http://dbpedia.org/ontology/City>.
?subject rdfs:label ?label.
FILTER ( lang(?label) = 'en')
}
LIMIT 10
```

```
Time (8 ms)
```

# Bibliography

[1] Sinan Al-Saffar and Gregory L. Heileman. Computing information value from rdf graph properties. In Gabriele Kotsis, David Taniar, Eric Pardede, Imad Saleh, and Ismail Khalil, editors, *iiWAS*, pages 349–356. ACM, 2010.

[2] Kemafor Anyanwu, Angela Maduko, and Amit P. Sheth. Semrank: ranking complex relationship search results on the semantic web. In Allan Ellis and Tatsuya Hagino, editors, *WWW*, pages 117–127. ACM, 2005.

[3] Kemafor Anyanwu and Amit P. Sheth. The p operator: Discovering and ranking associations on the semantic web. *SIGMOD Record*, 31(4):42–47, 2002.

[4] Maurizio Atzori and Andrea Dessi. Ranking dbpedia properties. In *23rd IEEE WETICE Conference, Web2Touch Track*, 2014.

[5] Maurizio Atzori and Carlo Zaniolo. Swipe: searching wikipedia by example. In Alain Mille, Fabien L. Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab, editors, *WWW (Companion Volume)*, pages 309–312. ACM, 2012.

[6] Maurizio Atzori and Carlo Zaniolo. Expressivity and accuracy of by-example structure queries on wikipedia. CSD Technical Report #140017, UCLA, 2014.

[7] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. Dbpedia: A nucleus for a web of open data. In Karl Aberer, Key-Sun Choi, Natasha Fridman Noy, Dean Allemang, Kyung-Il Lee, Lyndon J. B. Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *ISWC/ASWC*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007.

[8] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein.

OWL Web Ontology Language Reference. Technical report, W3C, http://www.w3.org/TR/owl-ref/, February 2004.

[9] Christian Becker and Christian Bizer. Dbpedia mobile: A location-enabled linked data browser. In Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee, editors, *LDOW*, volume 369 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[10] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[11] Abraham Bernstein and Esther Kaufmann. Gino - a guided input natural language ontology editor. In *Proceedings of the 5th International Semantic Web Conference (ISWC 2006)*, Athens, Georgia (US), November 2006.

[12] Roi Blanco, Peter Mika, and Sebastiano Vigna. Effective and efficient entity search in rdf data. In *International Semantic Web Conference (1)*, pages 83–97, 2011.

[13] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[14] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *ICML '05*, ICML '05, pages 89–96, New York, NY, USA, 2005. ACM.

[15] Elena Cabrio, Julien Cojan, Fabien Gandon, and Amine Hallili. Querying multilingual dbpedia with qakis. In Philipp Cimiano, Miriam Fernndez, Vanessa Lopez, Stefan Schlobach, and Johanna Vlker, editors, *ESWC (Satellite Events)*, volume 7955 of *Lecture Notes in Computer Science*, pages 194–198. Springer, 2013.

[16] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: From pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 129–136, New York, NY, USA, 2007. ACM.

[17] Philipp Cimiano, Peter Haase, Jörg Heizmann, Matthias Mantel, and Rudi Studer. Towards portable natural language interfaces to knowledge bases - the case of the ORAKEL system. *Data Knowl. Eng.*, 65(2):325–354, 2008.

[18] Lorand Dali, Blaz Fortuna, Duc Thanh Tran, and Dunja Mladenic. Query-independent learning to rank for rdf entity search. In Elena Simperl, Philipp Cimiano, Axel Polleres, Óscar Corcho, and Valentina Presutti, editors, *ESWC*, volume 7295 of *Lecture Notes in Computer Science*, pages 484–498. Springer, 2012.

[19] A. Dessi and M. Atzori. Computing on-the-fly dbpedia property ranking. In *Semantic Computing (ICSC), 2014 IEEE International Conference on*, pages 260–261, June 2014.

[20] Andrea Dessi, Andrea Maxia, Maurizio Atzori, and Carlo Zaniolo. Supporting semantic web search and structured queries on mobile devices. In Roberto De Virgilio, James Geller, Paolo Cappellari, and Mark Roantree, editors, *SSW@VLDB*, page 5. ACM, 2013.

[21] Li Ding, Rong Pan, Timothy W. Finin, Anupam Joshi, Yun Peng, and Pranam Kolari. Finding and ranking knowledge on the semantic web. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2005.

[22] Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, Marcin Sydow, and Gerhard Weikum. Language-model-based ranking for queries on rdf-graphs. In David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy J. Lin, editors, *CIKM*, pages 977–986. ACM, 2009.

[23] Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, and Gerhard Weikum. Searching rdf graphs with SPARQL and keywords. *IEEE Data Eng. Bull.*, 33(1):16–24, 2010.

[24] Basil Ell, Denny Vrandecic, and Elena Paslaru Bontas Simperl. Labels in the web of data. In *International Semantic Web Conference (ISWC) 2011*, 2011.

[25] Paolo Ferragina and Ugo Scaiella. TAGME: on-the-fly annotation of short text fragments (by wikipedia entities). In Jimmy Huang, Nick Koudas, Gareth J. F. Jones, Xindong Wu, Kevyn Collins-Thompson, and Aijun An, editors, *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010*, pages 1625–1628. ACM, 2010.

[26] Paolo Ferragina and Ugo Scaiella. Fast and accurate annotation of short texts with wikipedia pages. *IEEE Software*, 29(1):70–75, 2012.

[27] David A. Ferrucci, Eric W. Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John M. Prager, Nico Schlaefer, and Christopher A. Welty. Building watson: An overview of the deepqa project. *AI Magazine*, 31(3):59–79, 2010.

[28] Maximiliano Firtman. *jQuery Mobile - Up and Running: Using HTML5 to Design Web Apps for Tablets and Smartphones.* O'Reilly, 2012.

[29] Thomas Franz, Antje Schultz, Sergej Sizov, and Steffen Staab. Triplerank: Ranking semantic web data by tensor decomposition. In Abraham Bernstein, David R. Karger, Tom Heath, Lee Feigenbaum, Diana Maynard, Enrico Motta, and Krishnaprasad Thirunarayan, editors, *International Semantic Web Conference*, volume 5823 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2009.

[30] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res.*, 4:933–969, December 2003.

[31] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232, 2000.

[32] Alvaro Graves, Sibel Adali, and Jim Hendler. A method to rank nodes in an rdf graph. In Christian Bizer and Anupam Joshi, editors, *International Semantic Web Conference (Posters & Demos)*, volume 401 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[33] Rasmus Hahn, Christian Bizer, Christopher Sahnwaldt, Christian Herta, Scott Robinson, Michaela Bürgle, Holger Düwiger, and Ulrich Scheel. Faceted wikipedia search. In Witold Abramowicz and Robert Tolksdorf, editors, *BIS*, volume 47 of *Lecture Notes in Business Information Processing*, pages 1–11. Springer, 2010.

[34] Yanan Hao, Yanchun Zhang, and Jinli Cao. Web services discovery and rank: An information retrieval approach. *Future Generation Comp. Syst.*, pages 1053–1062, 2010.

[35] Xin He and Mark Baker. xhrank: Ranking entities on the semantic web. In Axel Polleres and Huajun Chen, editors, *ISWC Posters&Demos*, volume 658 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[36] Aidan Hogan, Andreas Harth, and Stefan Decker. Reconrank: A scalable ranking method for semantic web data with context. In *In 2nd Workshop on Scalable Semantic Web Knowledge Base Systems*, 2006.

[37] Zhixing Huang and Yuhui Qiu. A multiple-perspective approach to constructing and aggregating citation semantic link network. *Future Generation Comp. Syst.*, 26(3):400–407, 2010.

[38] Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. A relaxed approach to rdf querying. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2006.

[39] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3c recommendation, W3C, February 1999.

[40] Taesung Lee, Zhongyuan Wang, Haixun Wang, and Seung-won Hwang. Attribute extraction and scoring: A probabilistic approach. In Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *29th IEEE ICDE 2013*. IEEE Computer Society, 2013.

[41] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 2014.

[42] Xiaoling Li, Huaimin Wang, Bo Ding, Xiaoyong Li, and Dawei Feng. Resource allocation with multi-factor node ranking in data center networks. *Future Generation Comp. Syst.*, 32:1–12, 2014.

[43] Tie-Yan Liu. *Learning to Rank for Information Retrieval*. Springer, 2011.

[44] Vanessa Lopez, Miriam Fernández, Enrico Motta, and Nico Stieler. Poweraqua: Supporting users in querying and exploring the semantic web. *Semantic Web*, 3(3):249–265, 2012.

[45] Brian McBride. The resource description framework (rdf) and its vocabulary description language rdfs. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 51–66. Springer, 2004.

[46] Pablo N. Mendes, Max Jakob, Andrés García-Silva, and Christian Bizer. Dbpedia spotlight: shedding light on the web of documents. In Chiara Ghidini, Axel-Cyrille Ngonga Ngomo, Stefanie N. Lindstaedt, and Tassilo Pellegrini, editors, *Proceedings the 7th International Conference on Semantic Systems, I-SEMANTICS 2011, Graz, Austria, September 7-9, 2011*, ACM International Conference Proceeding Series, pages 1–8. ACM, 2011.

[47] Donald Metzler and W. Bruce Croft. Linear feature-based models for information retrieval. *Inf. Retr.*, 10(3):257–274, June 2007.

[48] Roberto Mirizzi, Azzurra Ragone, Tommaso Di Noia, and Eugenio Di Sciascio. Ranking the linked data: The case of dbpedia. In Boualem Benatallah, Fabio Casati, Gerti Kappel, and Gustavo Rossi, editors, *ICWE*, volume 6189 of *Lecture Notes in Computer Science*, pages 337–354. Springer, 2010.

[49] Hamid Mousavi, Maurizio Atzori, Shi Gao, and Carlo Zaniolo. Text-mining, structured queries, and knowledge management on web document corpora. *SIGMOD Record*, 43(3):48–54, 2014.

[50] Hamid Mousavi, Shi Gao, and Carlo Zaniolo. Ibminer: A text mining tool for constructing and populating infobox databases and knowledge bases. *PVLDB*, 6(12):1330–1333, 2013.

[51] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF, W3C recommendation. Technical report, World Wide Web Consortium, January 2008.

[52] Muhammad Saleem, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo. Feasible: A feature-based sparql benchmark generation framework. In *International Semantic Web Conference (ISWC)*, 2015.

[53] Nico Schlaefer, Jeongwoo Ko, Justin Betteridge, Manas A. Pathak, Eric Nyberg, and Guido Sautter. Semantic extensions of the ephyra QA system for TREC 2007. In Ellen M. Voorhees and Lori P. Buckland, editors, *Proceedings*

*of The Sixteenth Text REtrieval Conference, TREC 2007, Gaithersburg, Maryland, USA, November 5-9, 2007*, volume Special Publication 500-274. National Institute of Standards and Technology (NIST), 2007.

[54] C. Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:88–103, 1904.

[55] Christina Unger, Corina Forascu, Vanessa Lopez, Axel-Cyrille Ngonga Ngomo, Elena Cabrio, Philipp Cimiano, and Sebastian Walter. Question answering over linked data (QALD-4). In Linda Cappellato, Nicola Ferro, Martin Halvey, and Wessel Kraaij, editors, *Working Notes for CLEF 2014 Conference, Sheffield, UK, September 15-18, 2014.*, volume 1180 of *CEUR Workshop Proceedings*, pages 1172–1180. CEUR-WS.org, 2014.

[56] Jörg Waitelonis, Nadine Ludwig, Magnus Knuth, and Harald Sack. Whoknows? evaluating linked data heuristics with a quiz that cleans up dbpedia. *Interact. Techn. Smart Edu.*, 8(4):236–248, 2011.

[57] Chong Wang, Miao Xiong, Qi Zhou, and Yong Yu. Panto: A portable natural language interface to ontologies. In Enrico Franconi, Michael Kifer, and Wolfgang May, editors, *ESWC*, volume 4519 of *Lecture Notes in Computer Science*, pages 473–487. Springer, 2007.

[58] Haofen Wang, Thanh Tran, Chang Liu, and Linyun Fu. Lightweight integration of IR and DB for scalable hybrid search with integrated ranking support. *J. Web Sem.*, 9(4):490–503, 2011.

[59] Yufeng Wang and Akihiro Nakao. Poisonedwater: An improved approach for accurate reputation ranking in p2p networks. *Future Generation Comp. Syst.*, pages 1317–1326, 2010.

[60] Max L. Wilson, Bill Kules, M. C. Schraefel, and Ben Shneiderman. From keyword search to exploration: Designing future search interfaces for the web. *Foundations and Trends in Web Science*, 2(1):1–97, 2010.

[61] Qiang Wu, Christopher J. Burges, Krysta M. Svore, and Jianfeng Gao. Adapting boosting for information retrieval measures. *Inf. Retr.*, 13(3):254–270, June 2010.

[62] Wentao Wu, Hongsong Li, Haixun Wang, and Kenny Q. Zhu. Probase: A probabilistic taxonomy for text understanding. In *ACM SIGMOD 2012*, pages 481–492, New York, NY, USA, 2012. ACM.

[63] Fatos Xhafa and Leonard Barolli. Semantics, intelligent processing and services for big data. *Future Generation Comp. Syst.*, pages 201–202, 2014.

[64] Jun Xu and Hang Li. Adarank: A boosting algorithm for information retrieval. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 391–398, New York, NY, USA, 2007. ACM.

[65] Hugo Zaragoza, Henning Rode, Peter Mika, Jordi Atserias, Massimiliano Ciaramita, and Giuseppe Attardi. Ranking very many typed entities on wikipedia. In Mrio J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjrn Olstad, ystein Haug Olsen, and Andr O. Falco, editors, *CIKM*, pages 1015–1018. ACM, 2007.